bitwise

Claude Heiland-Allen

2012–2019

Contents

1 bitwise.cabal

```
Name:                bitwise
Version:             1.0.0.1
Synopsis:            fast multi-dimensional unboxed bit packed Bool arrays
Description:
  Unboxed multidimensional bit packed Bool arrays with fast aggregate
  operations based on lifting Bool operations to bitwise operations.
  .
  There are many other bit packed structures out there, but none met
  all of these requirements:
  .
    (1) unboxed bit packed Bool array,
  .
    (2) multi-dimensional indexing,
  .
    (3) fast (de)serialization, or interoperable with foreign code,
  .
    (4) fast aggregate operations (fold, map, zip).
  .
  Quick tour of the bitwise library:
  .
  [Data.Bits.Bitwise] Lift boolean operations on 'Bool' to bitwise
  operations on 'Data.Bits.Bits'.
  .
  [Data.Array.BitArray] Immutable bit arrays.
  .
  [Data.Array.BitArray.ST] Mutable bit arrays in 'Control.Monad.ST.ST'.
  .
```

```
    [Data.Array.BitArray.IO] Mutable bit arrays in 'IO'.
    .
    [Data.Array.BitArray.ByteString] (De)serialization.
    .
    [Codec.Image.PBM] Portable bitmap monochrome 2D image format.
    .
    Very rough performance benchmarks:
    .
      * immutable random access single bit reads:
        @BitArray ix@ is about 40% slower than @UArray ix Bool@,
    .
      * 'Control.Monad.ST.ST' mutable random access single bit reads:
        @STBitArray s ix@ is about the same as @STUArray s ix Bool@,
    .
      * immutable map @Bool -> Bool@:
        @BitArray ix@ is about 85x faster than @UArray ix Bool@,
    .
      * immutable zipWith @Bool -> Bool -> Bool@:
        @BitArray ix@ is about 1300x faster than @UArray ix Bool@.

Homepage:            https://code.mathr.co.uk/bitwise
License:             BSD3
License-file:        LICENSE
Author:              Claude Heiland-Allen
Maintainer:          claude@mathr.co.uk
Copyright:           (c) 2012,2016,2018 Claude Heiland-Allen
Category:            Data, Data Structures, Bit Vectors
Build-type:          Simple

Cabal-version:       >= 1.10

Library
  Exposed-modules:
    Data.Bits.Bitwise
    Data.Array.BitArray
    Data.Array.BitArray.IO
    Data.Array.BitArray.ST
    Data.Array.BitArray.ByteString
    Codec.Image.PBM

  Other-modules:
    Data.Array.BitArray.Internal

  Build-depends:
    base >= 4.7 && < 4.14,
    bytestring < 0.11,
    array < 0.6

  HS-source-dirs: src

  Default-Language: Haskell2010
  Other-Extensions: PatternGuards

  GHC-Options:         -Wall

Test-Suite bitwise-testsuite
  type:     exitcode-stdio-1.0
```

```
    main-is: extra/testsuite.hs
    build-depends:
      bitwise,
      base,
      QuickCheck >= 2.4 && < 2.14
    Default-Language: Haskell2010

Benchmark bitwise-benchmark
    type:     exitcode-stdio-1.0
    main-is: extra/benchmark.hs
    build-depends:
      bitwise,
      base,
      array,
      bytestring,
      criterion >= 0.6 && < 1.6
    Default-Language: Haskell2010

source-repository head
    type:     git
    location: https://code.mathr.co.uk/bitwise.git

source-repository this
    type:     git
    location: https://code.mathr.co.uk/bitwise.git
    tag:      v1.0.0.1
```

2 extra/benchmark.hs

```
{-# LANGUAGE BangPatterns #-}
module Main(main) where

import Control.Exception (evaluate)
import Control.Monad.ST (runST, ST)
import Data.Bits (shiftR)
import Data.ByteString (ByteString, pack, unpack)
import Data.Ix (range)
import Data.List (foldl1')
import Data.Word (Word8)
import System.Environment (getArgs)

import Data.Bits.Bitwise (packWord8LE, unpackWord8LE)
import qualified Data.Array.Unboxed  as A
import qualified Data.Array.BitArray as B
import qualified Data.Array.ST as STA (STUArray, readArray)
import qualified Data.Array.Unsafe as STA (unsafeThaw)
import qualified Data.Array.BitArray.ST as STB

import qualified Data.Array.BitArray.ByteString as BSB

import Criterion.Main

type I = (Int, Int, Int, Int)
type A = A.UArray I Bool
type B = B.BitArray I

bs :: (I, I)
```

```
    bs = ((0, 0, 0, 0), (na - 1, nb - 1, nc - 1, nd - 1))
30
    na, nb, nc, nd, n :: Int
    na = 13
    nb = 17
    nc = 19
35  nd = 11
    n = na * nb * nc * nd

    next :: I -> I
    next !(!a, !b, !c, !d) = ((a + 1) `mod` na, (b + 1) `mod` nb, (c + 1) `mod` nc, ↙
        ↳ (d + 1) `mod` nd)
40
    aToB :: A -> B
    aToB a = {-# SCC "aToB" #-} B.listArray (A.bounds a) (A.elems a)

    bToA :: B -> A
45  bToA b = {-# SCC "bToA" #-} A.listArray (B.bounds b) (B.elems b)

    aToBS :: A -> ByteString
    aToBS a = {-# SCC "aToBS" #-} pack . toWord8sLE . A.elems $ a

50  bToBS :: B -> ByteString
    bToBS b = {-# SCC "bToBS" #-} BSB.toByteString b

    aFromBS :: ((I, I), ByteString) -> A
    aFromBS (i, a) = {-# SCC "aFromBS" #-} A.listArray i . fromWord8sLE . unpack $ a
55
    bFromBS :: ((I, I), ByteString) -> B
    bFromBS (i, b) = {-# SCC "bFromBS" #-} BSB.fromByteString i b

    aListArray :: [Bool] -> A
60  aListArray a = {-# SCC "aListArray" #-} A.listArray bs a

    bListArray :: [Bool] -> B
    bListArray b = {-# SCC "bListArray" #-} B.listArray bs b

65  aIndex :: A -> ()
    aIndex !a = {-# SCC "aIndex" #-} loop (1, 1, 1, 1) (n `div` 3)
      where
        loop _ 0 = ()
        loop !i !m = (a A.! i) `seq` loop (next i) (m - 1)
70
    bIndex :: B -> ()
    bIndex !b = {-# SCC "bIndex" #-} loop (1, 1, 1, 1) (n `div` 3)
      where
        loop _ 0 = ()
75      loop !i !m = (b B.! i) `seq` loop (next i) (m - 1)

    bIndex' :: B -> ()
    bIndex' !b = {-# SCC "bIndex'" #-} loop (1, 1, 1, 1) (n `div` 3)
      where
80      loop _ 0 = ()
        loop !i !m = (b B.!!! i) `seq` loop (next i) (m - 1)

    aIndexST :: A -> ()
    aIndexST !a = {-# SCC "aIndexST" #-} runST $ do
```

```
    !a' <- STA.unsafeThaw a :: ST s (STA.STUArray s I Bool)
    loop a' (1, 1, 1, 1) (n `div` 3)
    where
      loop _ _ 0 = return ()
      loop !a' !i !m = do
        !_ <- STA.readArray a' i
        loop a' (next i) (m - 1)

bIndexST :: B -> ()
bIndexST !b = {-# SCC "bIndexST" #-} runST $ do
  !b' <- STB.unsafeThaw b
  loop b' (1, 1, 1, 1) (n `div` 3)
  where
    loop _ _ 0 = return ()
    loop !b' !i !m = do
      !_ <- STB.readArray b' i
      loop b' (next i) (m - 1)

bIndexST' :: B -> ()
bIndexST' !b = {-# SCC "bIndexST'" #-} runST $ do
  !b' <- STB.unsafeThaw b
  loop b' (1, 1, 1, 1) (n `div` 3)
  where
    loop _ _ 0 = return ()
    loop !b' !i !m = do
      !_ <- STB.unsafeReadArray b' i
      loop b' (next i) (m - 1)

aUpdate :: A -> A
aUpdate !a = {-# SCC "aUpdate" #-} a A.// take (n `div` 3) (loop (1, 1, 1, 1) ↙
    ↳ False)
  where
    loop (0, 0, 0, 0) _ = []
    loop !i !c = (i, c) : loop (next i) (not c)

bUpdate :: B -> B
bUpdate !b = {-# SCC "bUpdate" #-} b B.// take (n `div` 3) (loop (1, 1, 1, 1) ↙
    ↳ False)
  where
    loop (0, 0, 0, 0) _ = []
    loop !i !c = (i, c) : loop (next i) (not c)

aElems :: A -> ()
aElems !a = {-# SCC "aElems" #-} seqList (A.elems a) ()

seqList :: [a] -> b -> b
seqList [] b = b
seqList (x:xs) b = x `seq` seqList xs b

bElems :: B -> ()
bElems b = {-# SCC "bElems" #-} seqList (B.elems b) ()

aMap :: A -> A
aMap !a = {-# SCC "aMap" #-} A.amap not a

bMap :: B -> B
bMap !b = {-# SCC "bMap" #-} B.amap not b
```

```
140
    aZipWith :: (A, A) -> A
    aZipWith (!a, !a') = {-# SCC "aZipWith" #-} A.listArray bs (map (\ !i -> a A.! i↙
        ↪ == a' A.! i) (range bs)) :: A

    bZipWith :: (B, B) -> B
145 bZipWith (!b, !b') = {-# SCC "bZipWith" #-} B.zipWith (==) b b' :: B

    aFold :: A -> Bool
    aFold !a = {-# SCC "aFold" #-} foldl1' (/=) (A.elems a)

150 bFold :: B -> Bool
    bFold !b = {-# SCC "bFold" #-} case B.fold (/=) b of
      Just !c -> c
      Nothing -> error "fold"

155 main :: IO ()
    main = do
      let a  = A.listArray bs (take n          lorem ) :: A
          a' = A.listArray bs (take n (drop n lorem)) :: A
          b  = B.listArray bs (take n          lorem ) :: B
160       b' = B.listArray bs (take n (drop n lorem)) :: B
          l  = take n lorem
      evaluate (l `seqList` a `seq` a' `seq` b `seq` b' `seq` ())
      defaultMain
        [ bgroup "listArray"
165         [ bench "UArray"   $ whnf aListArray l
            , bench "BitArray" $ whnf bListArray l
            ]
        , bgroup "elems"
            [ bench "UArray"   $ whnf aElems a
170         , bench "BitArray" $ whnf bElems b
            ]
        , bgroup "index I"
            [ bench "UArray"   $ whnf aIndex a
            , bench "BitArray" $ whnf bIndex b
175         , bench "BitArray'"$ whnf bIndex' b
            ]
        , bgroup "index ST"
            [ bench "UArray"   $ whnf aIndexST a
            , bench "BitArray" $ whnf bIndexST b
180         , bench "BitArray'"$ whnf bIndexST' b
            ]
        , bgroup "update"
            [ bench "UArray"   $ whnf aUpdate a
            , bench "BitArray" $ whnf bUpdate b
185         ]
        , bgroup "map"
            [ bench "UArray"   $ whnf aMap a
            , bench "BitArray" $ whnf bMap b
            ]
190     , bgroup "zipWith"
            [ bench "UArray"   $ whnf aZipWith (a, a')
            , bench "BitArray" $ whnf bZipWith (b, b')
            ]
        , bgroup "fold"
195         [ bench "UArray"   $ whnf aFold a
```

```
            , bench "BitArray" $ whnf bFold b
            ]
        , bgroup "conversion"
            [ bench "U to Bit" $ whnf aToB a
            , bench "Bit to U" $ whnf bToA b
            ]
        , bgroup "serialize"
            [ bench "UArray"   $ whnf aToBS loremA
            , bench "BitArray" $ whnf bToBS loremB
            ]
        , bgroup "deserialize"
            [ bench "UArray"   $ whnf aFromBS (bs, loremBS)
            , bench "BitArray" $ whnf bFromBS (bs, loremBS)
            ]
        ]

loremA :: A
loremA = A.listArray bs (take n lorem)

loremB :: B
loremB = B.listArray bs (take n lorem)

lorem :: [Bool]
lorem = cycle . fromWord8sLE $ loremWS

loremBS :: ByteString
loremBS = pack . take ((n + 7) `shiftR` 3) . cycle $ loremWS

loremWS :: [Word8]
loremWS = map (toEnum . fromEnum) . unlines $
  [ "Lorem ipsum dolor sit amet, consectetur adipisicing elit, "
  , "sed do eiusmod tempor incididunt ut labore et dolore magna "
  , "aliqua. Ut enim ad minim veniam, quis nostrud exercitation "
  , "ullamco laboris nisi ut aliquip ex ea commodo consequat. "
  , "Duis aute irure dolor in reprehenderit in voluptate velit "
  , "esse cillum dolore eu fugiat nulla pariatur. Excepteur sint "
  , "occaecat cupidatat non proident, sunt in culpa qui officia "
  , "deserunt mollit anim id est laborum."
  ]

un8 :: (a, a, a, a, a, a, a, a) -> [a]
un8 (a, b, c, d, e, f, g, h) = [a, b, c, d, e, f, g, h]

fromWord8sLE :: [Word8] -> [Bool]
fromWord8sLE = concatMap (un8 . unpackWord8LE)

toWord8sLE :: [Bool] -> [Word8]
toWord8sLE [] = []
toWord8sLE (a:b:c:d:e:f:g:h:rest) = packWord8LE a b c d e f g h : toWord8sLE ↙
    ↳ rest
toWord8sLE rest = toWord8sLE (take 8 (rest ++ repeat False))
```

3 extra/testsuite.hs

```
{-# LANGUAGE TemplateHaskell #-}
module Main(main) where
```

```
import Prelude hiding (any, all, and, or, map, zipWith)
import qualified Prelude as P
import qualified Data.List as P

import Data.Ix (inRange, range)
import Data.Function (on)
import Data.Word (Word8, Word16)
import System.Exit (exitSuccess, exitFailure)

import Data.Array.BitArray

import Test.QuickCheck
import Test.QuickCheck.All(quickCheckAll)

fromW :: Word16 -> Int
fromW = fromIntegral

fromW8 :: Word8 -> Int
fromW8 = fromIntegral

prop_bounds1 o w = let n = fromW w in (o, o + n) == bounds (listArray (o, o + n) (take (n + 1) (cycle [False, True, True]))) 

prop_bounds2 o1 w1 o2 w2 = let n1 = fromW8 w1 ; n2 = fromW8 w2 ; bs = ((o1, o2), (o1 + n1, o2 + n2)) in bs == bounds (listArray bs (take ((n1 + 1) * (n2 + 1)) (cycle [False, True, True]))) 

prop_index1 o es = let n = length es in n > 0 ==> P.and [es !! i == listArray (o, o + n - 1) es ! (o + i) | i <- [0 .. n - 1]]

prop_index2 o1 o2 es1 = let n2 = ceiling . sqrt . fromIntegral . length $ es1 in n2 > 0 ==>
  let es = init (chunk n2 es1)
      n1 = length es
      bs = ((o1, o2), (o1 + n1 - 1,o2 + n2 - 1))
  in  n1 > 0 ==> P.and [ es !! (i - o1) !! (j - o2) == listArray bs (concat es) ! (i, j) | (i, j) <- range bs ]

prop_indices1 o w = let n = fromW w ; bs = (o, o + n) in range bs == indices (listArray bs (cycle [False, True, True]))

prop_indices2 o1 w1 o2 w2 =
  let n1 = fromW8 w1
      n2 = fromW8 w2
      bs = ((o1, o2), (o1 + n1, o2 + n2))
  in  range bs == indices (listArray bs (cycle [False, True, True]))

prop_elems1 o es = es == (elems . listArray (o, o + length es - 1)) es

prop_assocs1 o es = zip [o..] es == (assocs . listArray (o, o + length es - 1)) es

prop_map1 (Blind f) o es = P.map f es == (elems . map f . listArray (o, o + length es - 1)) es

prop_zipWith1 (Blind f) o ees = P.map (uncurry f) ees == (elems . uncurry (zipWith f `on` listArray (o, o + length ees - 1)) . unzip) ees
```

```
prop_or1 o es = P.or es == (or . listArray (o, o + length es - 1)) es

prop_and1 o es = P.and es == (and . listArray (o, o + length es - 1)) es

prop_isUniform1 o es = not (null es) ==> listUniform es == (isUniform . ↙
    ↳ listArray (o, o + length es - 1)) es

prop_fill1 o w b = let n = fromW w in Just b == isUniform (fill (o, o + n) b)

prop_true1 o w = let n = fromW w in Just True == isUniform (true (o, o + n))

prop_false1 o w = let n = fromW w in Just False == isUniform (false (o, o + n))

prop_elemIndex b o es = (fmap (+ o) . P.elemIndex b) es == (elemIndex b . ↙
    ↳ listArray (o, o + length es - 1)) es

prop_popCount o es = (P.length . P.filter id) es == (popCount . listArray (o, o ↙
    ↳ + length es - 1)) es

listUniform l
  | null l = Nothing
  | P.and l = Just True
  | not (P.or l) = Just False
  | otherwise = Nothing

chunk _ [] = []
chunk n xs = let (ys, zs) = splitAt n xs in ys : chunk n zs

{-
  , accumArray
  , (//)
  , accum
  , ixmap
  , (!?)
-}

return []
main :: IO ()
main = do
  r <- $quickCheckAll
  if r then exitSuccess else exitFailure
```

4 .gitignore

```
.cabal-sandbox
cabal.sandbox.config
dist
dist-newstyle
_
```

5 LICENSE

```
Copyright (c)2012,2016,2018, Claude Heiland-Allen

All rights reserved.
```

6 Setup.hs

```
import Distribution.Simple
main = defaultMain
```

7 src/Codec/Image/PBM.hs

```
{-# OPTIONS_GHC -fno-warn-name-shadowing #-}
-- avoid a flood of warnings
{-|

Module      :  Codec.Image.PBM
Copyright   :  (c) Claude Heiland-Allen 2012,2018
License     :  BSD3

Maintainer  :  claude@mathr.co.uk
Stability   :  unstable
Portability :  portable

Encode and decode both versions (binary P4 and plain P1) of PBM: the
portable bitmap lowest common denominator monochrome image file format.

References:

  * pbm(5)

  * The PBM Format <http://netpbm.sourceforge.net/doc/pbm.html>

Bugs:
```

```
    * This implementation is not fully compliant with the PBM specification,
      with respect to point 8 in the second reference above which states that
      /a comment can actually be in the middle of what you might consider a token/
      Such a pathological PBM file might be rejected by 'decodePBM', but
      may instead be wrongly decoded if (for example) the comment were in
      the middle of the image width token, leading to it being interpreted
      as a (smaller) width and height.

-}
module Codec.Image.PBM
  ( PBM(..)
  -- * Encoding PBM images.
  , encodePBM
  , encodePlainPBM
  , EncodeError(..)
  , encodePBM'
  -- * Decoding PBM images.
  , DecodeError(..)
  , decodePBM
  , decodePlainPBM
  , decodePBMs
  -- * Padding and trimming PBM images.
  , padPBM
  , trimPBM
  , repadPBM
  ) where

import Data.Bits (shiftL, shiftR, (.&.))
import Data.Ix (range)
import Data.Word (Word8)
import qualified Data.Array.Unboxed as U
import qualified Data.ByteString as BS

import Data.Bits.Bitwise (fromListBE, toListLE)
import Data.Array.BitArray (BitArray, bounds, elems, listArray, false, (//), ↙
    ↳ assocs, ixmap)
import Data.Array.BitArray.ByteString (toByteString, fromByteString)

-- | A decoded PBM image.  'pbmWidth' must be less or equal to the
--   width of the 'pbmPixels' array (which has its first index in Y
--   and the second in X, with lowest coordinates at the top left).
--
--   False pixels are white, True pixels are black.  Pixels to the
--   right of 'pbmWidth' are don't care padding bits.  However, these
--   padding bits are likely to invalidate aggregrate 'BitArray.fold'
--   operations.  See 'trimPBM'.
--
data PBM = PBM{ pbmWidth :: !Int, pbmPixels :: !(BitArray (Int, Int)) }

-- | Encode a binary PBM (P4) image, padding rows to multiples of 8
--   bits as necessary.
--
encodePBM :: BitArray (Int, Int) {- ^ pixels -} -> BS.ByteString
encodePBM pixels = case encodePBM' pbm of
  Right string -> string
  _ -> error "Codec.Image.PBM.encodePBM: internal error"
```

```
    where
      ((_, xlo), (_, xhi)) = bounds pixels
      width  = xhi - xlo + 1
      pbm = padPBM PBM{ pbmWidth = width, pbmPixels = pixels }

-- | Possible reasons for encoding to fail.
data EncodeError
  = BadPixelWidth{ encErrPBM :: PBM } -- ^ array width is not a multiple of 8 ↙
      ↳ bits
  | BadSmallWidth{ encErrPBM :: PBM } -- ^ image width is too smaller than array ↙
      ↳ width
  | BadLargeWidth{ encErrPBM :: PBM } -- ^ image width is larger than array ↙
      ↳ width

-- | Encode a plain PBM (P1) image.
--
--  No restrictions on pixels array size, but the file format is
--  exceedingly wasteful of space.
--
encodePlainPBM :: BitArray (Int, Int) {- ^ pixels -} -> String
encodePlainPBM pixels = unlines (header : raster)
  where
    ((ylo, xlo), (yhi, xhi)) = bounds pixels
    width  = xhi - xlo + 1
    height = yhi - ylo + 1
    header = "P1\n" ++ show width ++ " " ++ show height
    raster = concatMap (chunk 64) . chunk width . map char . elems $ pixels
    char False = '0'
    char True  = '1'
    chunk n _ | n <= 0 = error "Codec.Image.PBM.encodePlainPBM: internal error"
    chunk _ [] = []
    chunk n xs = let (ys, zs) = splitAt n xs in ys : chunk n zs

-- | Encode a pre-padded 'PBM' to a binary PBM (P4) image.
--
--  The pixels array must have a multiple of 8 bits per row.  The image
--  width may be less than the pixel array width, with up to 7 padding
--  bits at the end of each row.
--
encodePBM' :: PBM -> Either EncodeError BS.ByteString
encodePBM' pbm
  | (pixelWidth .&. 7) /= 0 = Left (BadPixelWidth pbm)
  | width <= pixelWidth - 8 = Left (BadSmallWidth pbm)
  | width >  pixelWidth     = Left (BadLargeWidth pbm)
  | otherwise = Right (header `BS.append` raster)
  where
    width = pbmWidth pbm
    pixels = pbmPixels pbm
    ((ylo, xlo), (yhi, xhi)) = bounds pixels
    pixelWidth  = xhi - xlo + 1
    pixelHeight = yhi - ylo + 1
    height = pixelHeight
    header = BS.pack $ map (toEnum . fromEnum) headerStr
    headerStr = "P4\n" ++ show width ++ " " ++ show height ++ "\n"
    raster = reverseByteBits (toByteString pixels)

-- | Possible reasons for decoding to fail, with the input that failed.
```

```
data DecodeError a
  = BadMagicP a -- ^ First character was not P.
  | BadMagicN a -- ^ Second character was not 4 (binary) or 1 (plain).
  | BadWidth  a -- ^ The width could not be parsed, or was non-positive.
  | BadHeight a -- ^ The height could not be parsed, or was non-positive.
  | BadSpace  a -- ^ Parsing failed at the space before the pixel data.
  | BadPixels a -- ^ There weren't enough bytes of pixel data.
  deriving (Eq, Ord, Read, Show)

-- | Decode a binary PBM (P4) image.
decodePBM :: BS.ByteString -> Either (DecodeError BS.ByteString) (PBM, BS.↙
    ↳ ByteString)
decodePBM s =                       case BS.uncons s of
  Just (cP, s) | cP == char 'P' -> case BS.uncons s of
    Just (c4, s) | c4 == char '4' -> case int (skipSpaceComment s) of
      Just (iw, s) | iw > 0         -> case int (skipSpaceComment s) of
        Just (ih, s) | ih > 0         -> case skipSingleSpace s of
          Just s                        ->
            let rowBytes = (iw + 7) `shiftR` 3
                imgBytes = ih * rowBytes
            in                                  case BS.splitAt imgBytes s of
            (raster, s) | BS.length raster == imgBytes ->
              let ibs = ((0, 0), (ih - 1, (rowBytes `shiftL` 3) - 1))
              in  Right (PBM{ pbmWidth = iw, pbmPixels = fromByteString ibs (↙
                  ↳ reverseByteBits raster) }, s)
            _ -> Left (BadPixels s)
          _ -> Left (BadSpace s)
        _ -> Left (BadHeight s)
      _ -> Left (BadWidth s)
    _ -> Left (BadMagicN s)
  _ -> Left (BadMagicP s)
  where
    skipSpaceComment t = case (\t -> (t, BS.uncons t)) (BS.dropWhile isSpace t) ↙
        ↳ of
      (_, Just (cH, t)) | cH == char '#' -> case BS.uncons (BS.dropWhile (/= ↙
          ↳ char '\n') t) of
        Just (cL, t) | cL == char '\n' -> skipSpaceComment t
        _ -> Left (BadSpace t)
      (t, _) -> Right t
    skipSingleSpace t = case BS.uncons t of
      Just (cS, t) | isSpace cS -> Just t
      _ -> Nothing
    int (Left _) = Nothing
    int (Right t) = case BS.span isDigit t of
      (d, t)
        | BS.length d > 0 &&
          fmap ((/= char '0') . fst) (BS.uncons d) == Just True -> case reads (↙
              ↳ map unchar $ BS.unpack d) of
            [(d, "")] -> Just (d, t)
            _ -> Nothing
      _ -> Nothing
    isSpace c = c `elem` map char pbmSpace
    isDigit c = c `elem` map char "0123456789"
    char = toEnum . fromEnum
    unchar = toEnum . fromEnum

-- | Decode a sequence of binary PBM (P4) images.
```

```
185 --
    --   Keeps decoding until end of input (in which case the 'snd' of the
    --   result is 'Nothing') or an error occurred.
    --
    decodePBMs :: BS.ByteString -> ([PBM], Maybe (DecodeError BS.ByteString))
190 decodePBMs s
      | BS.null s = ([], Nothing)
      | otherwise = case decodePBM s of
          Left err -> ([], Just err)
          Right (pbm, s) -> prepend pbm (decodePBMs s)
195   where
        prepend pbm (pbms, merr) = (pbm:pbms, merr)

    -- | Decode a plain PBM (P1) image.
    --
200 --   Note that the pixel array size is kept as-is (with the width not
    --   necessarily a multiple of 8 bits).
    --
    decodePlainPBM :: String -> Either (DecodeError String) (PBM, String)
    decodePlainPBM s = case s of
205   ('P':s) -> case s of
        ('1':s) -> case int (skipSpaceComment s) of
          Just (iw, s) | iw > 0 -> case int (skipSpaceComment s) of
            Just (ih, s) | ih > 0 -> case collapseRaster (iw * ih) s of
              Just (raster, s) ->
210             let ibs = ((0, 0), (ih - 1, iw - 1))
                in  Right (PBM{ pbmWidth = iw, pbmPixels = listArray ibs raster }, s↙
                    ↳ )
              _ -> Left (BadPixels s)
            _ -> Left (BadHeight s)
          _ -> Left (BadWidth s)
215     _ -> Left (BadMagicN s)
      _ -> Left (BadMagicP s)
      where
        skipSpaceComment t = case dropWhile isSpace t of
          ('#':t) -> case dropWhile (/= '\n') t of
220         ('\n':t) -> skipSpaceComment t
            _ -> Left (BadSpace t)
          t -> Right t
        int (Left _) = Nothing
        int (Right t) = case span isDigit t of
225       (d@(d0:_), t) | d0 /= '0' -> case reads d of
            [(d, "")] -> Just (d, t)
            _ -> Nothing
          _ -> Nothing
        collapseRaster 0 t = Just ([], t)
230     collapseRaster n t = case dropWhile isSpace t of
          ('0':t) -> prepend False (collapseRaster (n - 1) t)
          ('1':t) -> prepend True  (collapseRaster (n - 1) t)
          _ -> Nothing
        prepend _ Nothing = Nothing
235     prepend b (Just (bs, t)) = Just (b:bs, t)
        isSpace c = c `elem` pbmSpace
        isDigit c = c `elem` "0123456789"

    -- | Add padding bits at the end of each row to make the array width a
240 --   multiple of 8 bits, required for binary PBM (P4) encoding.
```

```
--
padPBM :: PBM -> PBM
padPBM pbm
  | (pixelWidth .&. 7) == 0 = pbm
  | otherwise = pbm{ pbmPixels = false paddedBounds // assocs (pbmPixels pbm) }
  where
    ((ylo, xlo), (yhi, xhi)) = bounds (pbmPixels pbm)
    pixelWidth = xhi - xlo + 1
    rowBytes = (pixelWidth + 7) `shiftR` 3
    paddedWidth = rowBytes `shiftL` 3
    paddedBounds = ((ylo, xlo), (yhi, xhi'))
    xhi' = paddedWidth + xlo - 1

-- | Trim any padding bits, required for 'fold' operations to give
--   meaningful results.
--
--   Fails for invalid 'PBM' with image width greater than array width.
--
trimPBM :: PBM -> Maybe PBM
trimPBM pbm
  | pbmWidth pbm > pixelWidth = Nothing
  | pbmWidth pbm == pixelWidth = Just pbm
  | otherwise = Just pbm{ pbmPixels = ixmap trimmedBounds id (pbmPixels pbm) }
  where
    ((ylo, xlo), (yhi, xhi)) = bounds (pbmPixels pbm)
    pixelWidth = xhi - xlo + 1
    trimmedBounds = ((ylo, xlo), (yhi, xhi'))
    xhi' = pbmWidth pbm + xlo - 1

-- | Trim then pad.  The resulting 'PBM' (if any) is suitable for
--   encoding to binary PBM (P4), moreover its padding bits will
--   be cleared.
repadPBM :: PBM -> Maybe PBM
repadPBM pbm = padPBM `fmap` trimPBM pbm

-- | Reverse the bit order of all bytes.
--
--  PBM specifies that the most significant bit is leftmost, which is
--   opposite to the convention used by BitArray.
--
reverseByteBits :: BS.ByteString -> BS.ByteString
reverseByteBits = BS.map reverseBits

-- | Fast reversal of the bit order of a byte using a lookup table.
reverseBits :: Word8 -> Word8
reverseBits w = bitReversed U.! w

-- | A lookup table for bit order reversal.
bitReversed :: U.UArray Word8 Word8
bitReversed = U.listArray bs [ bitReverse w | w <- range bs ]
  where bs = (minBound, maxBound)

-- | A slow way to reverse bit order.
bitReverse :: Word8 -> Word8
bitReverse = fromListBE . toListLE

-- | White space characters as defined by the PBM specification.
```

```
pbmSpace :: String
pbmSpace = " \t\n\v\f\r"
```

8 src/Data/Array/BitArray/ByteString.hs

```
{-|

Module      :  Data.Array.BitArray.ByteString
Copyright   :  (c) Claude Heiland-Allen 2012,2018
License     :  BSD3

Maintainer  :  claude@mathr.co.uk
Stability   :  unstable
Portability :  portable

Copy bit array data to and from ByteStrings.

-}
module Data.Array.BitArray.ByteString
  (
  -- * Immutable copying.
    toByteString
  , fromByteString
  -- * Mutable copying.
  , toByteStringIO
  , fromByteStringIO
  ) where

import Data.Bits (shiftR, (.&.))
import Data.ByteString (ByteString, packCStringLen)
import Data.ByteString.Unsafe (unsafeUseAsCStringLen)
import Data.Ix (Ix, rangeSize)
import Data.Word (Word8)
import Control.Monad (when)
import Foreign.ForeignPtr (withForeignPtr)
import Foreign.Marshal.Utils (copyBytes)
import Foreign.Ptr (castPtr)
import Foreign.Storable (peekByteOff, pokeByteOff)
import System.IO.Unsafe (unsafePerformIO)

import Data.Bits.Bitwise (mask)
import Data.Array.BitArray (BitArray)
import Data.Array.BitArray.IO (IOBitArray)
import qualified Data.Array.BitArray.IO as IO
import Data.Array.BitArray.Internal (iobData)

-- | Copy to a ByteString.  The most significant bits of the last byte
--   are padded with 0 unless the array was a multiple of 8 bits in size.
toByteString :: Ix i => BitArray i -> ByteString
toByteString a = unsafePerformIO $ toByteStringIO =<< IO.unsafeThaw a

-- | Copy from a ByteString.  Much like 'listArray' but with packed bits.
fromByteString :: Ix i => (i, i) {- ^ bounds -} -> ByteString {- ^ packed elems ↙
    ↳ -} -> BitArray i
fromByteString bs s = unsafePerformIO $ IO.unsafeFreeze =<< fromByteStringIO bs ↙
    ↳ s
```

```
-- | Copy to a ByteString.  The most significant bits of the last byte
--   are padded with 0 unless the array was a multiple of 8 bits in size.
toByteStringIO :: Ix i => IOBitArray i -> IO ByteString
toByteStringIO a = do
  bs <- IO.getBounds a
  let rs = rangeSize bs
      bytes = (rs + 7) `shiftR` 3
      bits = rs .&. 7
      lastByte = bytes - 1
  withForeignPtr (iobData a) $ \p -> do
    when (bits /= 0) $ do
      b <- peekByteOff p lastByte
      pokeByteOff p lastByte (b .&. mask bits :: Word8)
    packCStringLen (castPtr p, bytes)

-- | Copy from a ByteString.  Much like 'newListArray' but with packed bits.
fromByteStringIO :: Ix i => (i, i) {- ^ bounds -} -> ByteString {- ^ packed ↙
    ↳ elems -} -> IO (IOBitArray i)
fromByteStringIO bs s = do
  a <- IO.newArray bs False
  let rs = rangeSize bs
      bytes = (rs + 7) `shiftR` 3
  unsafeUseAsCStringLen s $ \(src, len) ->
    withForeignPtr (iobData a) $ \dst ->
      copyBytes dst (castPtr src) (bytes `min` len)
  return a
```

9 src/Data/Array/BitArray.hs

```
{-|

Module      :  Data.Array.BitArray
Copyright   :  (c) Claude Heiland-Allen 2012
License     :  BSD3

Maintainer  :  claude@mathr.co.uk
Stability   :  unstable
Portability :  portable

Immutable unboxed packed bit arrays using bitwise operations to
manipulate large chunks at a time much more quickly than individually
unpacking and repacking bits would allow.

-}
-- almost all is implemented with runST and the ST-based implementation
module Data.Array.BitArray
  ( BitArray()
  -- * IArray-like interface.
  , bounds
  , array
  , listArray
  , accumArray
  , (!)
  , indices
  , elems
  , assocs
  , (//)
```

```
    , accum
    , amap
    , ixmap
    -- * Constant arrays.
    , fill
    , false
    , true
    -- * Short-circuiting reductions.
    , or
    , and
    , isUniform
    , elemIndex
    -- * Aggregate operations.
    , fold
    , map
    , zipWith
    , popCount
    -- * Bounds-checked indexing.
    , (!?)
    -- * Unsafe.
    , (!!!)
    ) where

import Prelude hiding (and, or, map, zipWith)
import qualified Prelude as P

import Control.Monad (forM_)
import Control.Monad.ST (runST)
import Data.Ix (Ix, range, inRange)

import Data.Array.BitArray.Internal (BitArray)
import qualified Data.Array.BitArray.ST as ST

-- | The bounds of an array.
{-# INLINE bounds #-}
bounds :: Ix i => BitArray i -> (i, i)
bounds a = runST (ST.getBounds =<< ST.unsafeThaw a)

-- | Create an array from a list of (index, element) pairs.
{-# INLINE array #-}
array :: Ix i => (i, i) {- ^ bounds -} -> [(i, Bool)] {- ^ assocs -} -> BitArray i
array bs ies = false bs // ies

-- | Create an array from a list of elements.
{-# INLINE listArray #-}
listArray :: Ix i => (i, i) {- ^ bounds -} -> [Bool] {- ^ elems -} -> BitArray i
listArray bs es = runST (ST.unsafeFreeze =<< ST.newListArray bs es)

-- | Create an array by accumulating a list of (index, operand) pairs
--   from a default seed with an operation.
{-# INLINE accumArray #-}
accumArray :: Ix i => (Bool -> a -> Bool) {- ^ operation -} -> Bool {- ^ default -} -> (i, i) {- ^ bounds -} -> [(i, a)] {- ^ assocs -} -> BitArray i
accumArray f d bs = accum f (fill bs d)

-- | Bit array indexing.
```

```
{-# INLINE (!) #-}
(!) :: Ix i => BitArray i -> i -> Bool
a ! i = runST (do
  a' <- ST.unsafeThaw a
  ST.readArray a' i)

-- | Bit array indexing without bounds checking.  Unsafe.
{-# INLINE (!!!) #-}
(!!!) :: Ix i => BitArray i -> i -> Bool
a !!! i = runST (do
  a' <- ST.unsafeThaw a
  ST.unsafeReadArray a' i)

-- | A list of all the valid indices for this array.
{-# INLINE indices #-}
indices :: Ix i => BitArray i -> [i]
indices = range . bounds

-- | A list of the elements in this array.
{-# INLINE elems #-}
elems :: Ix i => BitArray i -> [Bool]
elems a = runST (ST.unsafeGetElems =<< ST.unsafeThaw a)
  -- P.map (a !!!) (indices a) -- very slow!

-- | A list of the (index, element) pairs in this array.
{-# INLINE assocs #-}
assocs :: Ix i => BitArray i -> [(i, Bool)]
assocs ba = P.map (\i -> (i, ba ! i)) (indices ba)

-- | A new array with updated values at the supplied indices.
{-# INLINE (//) #-}
(//) :: Ix i => BitArray i -> [(i, Bool)] {- ^ new assocs -} -> BitArray i
ba // ies = accum (\_ a -> a) ba ies

-- | Accumulate with an operation and a list of (index, operand).
{-# INLINE accum #-}
accum :: Ix i => (Bool -> a -> Bool) {- ^ operation -} -> BitArray i {- ^ source↙
    ↳ -} -> [(i, a)] {- ^ assocs -} -> BitArray i
accum f a ies = runST (do
  a' <- ST.thaw a
  forM_ ies $ \(i, x) -> do
    b <- ST.readArray a' i
    ST.writeArray a' i (f b x)
  ST.unsafeFreeze a')

-- | Alias for 'map'.
{-# INLINE amap #-}
amap :: Ix i => (Bool -> Bool) -> BitArray i -> BitArray i
amap = map

-- | Create a new array by mapping indices into a source array..
{-# INLINE ixmap #-}
ixmap :: (Ix i, Ix j) => (i, i) {- ^ new bounds -} -> (i -> j) {- ^ index ↙
    ↳ transformation -} -> BitArray j {- ^ source array -} -> BitArray i
ixmap bs h ba = array bs (P.map (\i -> (i, ba ! h i)) (range bs))

-- | A uniform array of bits.
```

```
    {-# INLINE fill #-}
140 fill :: Ix i => (i, i) {- ^ bounds -} -> Bool -> BitArray i
    fill bs b = runST (ST.unsafeFreeze =<< ST.newArray bs b)

    -- | A uniform array of 'False'.
    {-# INLINE false #-}
145 false :: Ix i => (i, i) {- ^ bounds -} -> BitArray i
    false bs = fill bs False

    -- | A uniform array of 'True'.
    {-# INLINE true #-}
150 true :: Ix i => (i, i) {- ^ bounds -} -> BitArray i
    true bs = fill bs True

    -- | Bounds checking combined with array indexing.
    {-# INLINE (!?) #-}
155 (!?) :: Ix i => BitArray i -> i -> Maybe Bool
    b !? i
      | inRange (bounds b) i = Just (b ! i)
      | otherwise = Nothing

160 -- | Short-circuit bitwise reduction: True if any bit is True.
    {-# INLINE or #-}
    or :: Ix i => BitArray i -> Bool
    or a = runST (ST.or =<< ST.unsafeThaw a)

165 -- | Short-circuit bitwise reduction: False if any bit is False.
    {-# INLINE and #-}
    and :: Ix i => BitArray i -> Bool
    and a = runST (ST.and =<< ST.unsafeThaw a)

170 -- | Short-circuit bitwise reduction: Nothing if any bits differ.
    {-# INLINE isUniform #-}
    isUniform :: Ix i => BitArray i -> Maybe Bool
    isUniform a = runST (ST.isUniform =<< ST.unsafeThaw a)

175 -- | Look up index of first matching bit.
    --
    --   Note that the index type is limited to Int because there
    --   is no 'unindex' method in the 'Ix' class.
    {-# INLINE elemIndex #-}
180 elemIndex :: Bool -> BitArray Int -> Maybe Int
    elemIndex b a = runST (ST.elemIndex b =<< ST.unsafeThaw a)

    -- | Bitwise reduction with an associative commutative boolean operator.
    --   Implementation lifts from 'Bool' to 'Bits' and folds large chunks
185 --   at a time.  Each bit is used as a source exactly once.
    {-# INLINE fold #-}
    fold :: Ix i => (Bool -> Bool -> Bool) -> BitArray i -> Maybe Bool
    fold f a = runST (ST.fold f =<< ST.unsafeThaw a)

190 -- | Bitwise map.  Implementation lifts from 'Bool' to 'Bits' and maps
    --   large chunks at a time.
    {-# INLINE map #-}
    map :: Ix i => (Bool -> Bool) -> BitArray i -> BitArray i
    map f a = runST (ST.unsafeFreeze =<< ST.map f =<< ST.unsafeThaw a)
195
```

```
-- | Bitwise zipWith.  Implementation lifts from 'Bool' to 'Bits' and
--   combines large chunks at a time.
--
--   The bounds of the source arrays must be identical.
{-# INLINE zipWith #-}
zipWith :: Ix i => (Bool -> Bool -> Bool) -> BitArray i -> BitArray i -> ↙
    ↳ BitArray i
zipWith f a b
  | bounds a == bounds b = runST (do
      a' <- ST.unsafeThaw a
      b' <- ST.unsafeThaw b
      ST.unsafeFreeze =<< ST.zipWith f a' b')
  | otherwise = error "zipWith bounds mismatch"

-- | Count set bits.
{-# INLINE popCount #-}
popCount :: Ix i => BitArray i -> Int
popCount a = runST (ST.popCount =<< ST.unsafeThaw a)
```

10 src/Data/Array/BitArray/Internal.hs

```
{-# OPTIONS_HADDOCK hide #-}
{-|

Module      :  Data.Array.BitArray.Internal
Copyright   :  (c) Claude Heiland-Allen 2012
License     :  BSD3

Maintainer  :  claude@mathr.co.uk
Stability   :  unstable
Portability :  portable

Bit arrays internals.  Not exposed.

-}
module Data.Array.BitArray.Internal
  ( BitArray(..)
  , IOBitArray(..)
  , getBounds
  , newArray_
  , freeze
  , thaw
  , copy
  , unsafeFreeze
  , unsafeThaw
  ) where

import Data.Bits (shiftL, shiftR)
import Data.Ix (Ix, rangeSize)
import Data.Word (Word64)
import Foreign.Marshal.Utils (copyBytes)
import Foreign.ForeignPtr (ForeignPtr, mallocForeignPtrBytes, withForeignPtr)

-- | The type of immutable bit arrays.
newtype BitArray i = B (IOBitArray i)

-- | The type of mutable bit arrays in the 'IO' monad.
```

```
data IOBitArray i = IOB{ iobBoundLo :: !i, iobBoundHi :: !i, iobBytes :: {-# ↙
    ↳ UNPACK #-} !Int, iobData :: {-# UNPACK #-} !(ForeignPtr Word64) }

-- | Create a new array filled with unspecified initial values.
{-# INLINE newArray_ #-}
newArray_ :: Ix i => (i, i) {- ^ bounds -} -> IO (IOBitArray i)
newArray_ bs@(bl, bh) = do
  let bits = rangeSize bs
      nwords = (bits + 63) `shiftR` 6
      bytes = nwords `shiftL` 3
  p <- mallocForeignPtrBytes bytes
  return IOB{ iobBoundLo = bl, iobBoundHi = bh, iobBytes = bytes, iobData = p }

-- | Get the bounds of a bit array.
{-# INLINE getBounds #-}
getBounds :: Ix i => IOBitArray i -> IO (i, i)
getBounds a = return (iobBoundLo a, iobBoundHi a)

-- | Snapshot the array into an immutable form.
{-# INLINE freeze #-}
freeze :: Ix i => IOBitArray i -> IO (BitArray i)
freeze a = B `fmap` copy a

-- | Snapshot the array into an immutable form.  Unsafe when the source
--   array can be modified later.
{-# INLINE unsafeFreeze #-}
unsafeFreeze :: Ix i => IOBitArray i -> IO (BitArray i)
unsafeFreeze a = B `fmap` return a

-- | Convert an array from immutable form.
{-# INLINE thaw #-}
thaw :: Ix i => BitArray i -> IO (IOBitArray i)
thaw (B a) = copy a

-- | Convert an array from immutable form.  Unsafe to modify the result
--   unless the source array is never used later.
{-# INLINE unsafeThaw #-}
unsafeThaw :: Ix i => BitArray i -> IO (IOBitArray i)
unsafeThaw (B a) = return a

-- | Copy an array.
{-# INLINE copy #-}
copy :: Ix i => IOBitArray i -> IO (IOBitArray i)
copy a = do
  b <- newArray_ =<< getBounds a
  withForeignPtr (iobData a) $ \ap ->
    withForeignPtr (iobData b) $ \bp ->
      copyBytes bp ap (iobBytes b)
  return b
```

11 src/Data/Array/BitArray/IO.hs

```
{-|

Module      :  Data.Array.BitArray.IO
Copyright   :  (c) Claude Heiland-Allen 2012,2018
License     :  BSD3
```

```
    Maintainer  :  claude@mathr.co.uk
    Stability   :  unstable
    Portability :  portable

    Unboxed mutable bit arrays in the 'IO' monad.

    -}
    module Data.Array.BitArray.IO
      ( IOBitArray()
      -- * MArray-like interface.
      , getBounds
      , newArray
      , newArray_
      , newListArray
      , readArray
      , writeArray
      , mapArray
      , mapIndices
      , getElems
      , getAssocs
      -- * Conversion to/from immutable bit arrays.
      , freeze
      , thaw
      -- * Construction
      , copy
      , fill
      -- * Short-circuiting reductions.
      , or
      , and
      , isUniform
      , elemIndex
      -- * Aggregate operations.
      , fold
      , map
      , zipWith
      , popCount
      -- * Unsafe.
      , unsafeReadArray
      , unsafeGetElems
      , unsafeFreeze
      , unsafeThaw
      ) where

    import Prelude hiding (and, or, map, zipWith)

    import Control.Monad (forM_, when)
    import Data.Bits (shiftL, shiftR, testBit, setBit, clearBit, (.&.), complement)
    import qualified Data.Bits
    import Data.Ix (Ix, index, inRange, range, rangeSize)
    import Data.List (foldl1')
    import Data.Word (Word8, Word64)
    import Foreign.ForeignPtr (withForeignPtr, touchForeignPtr)
    import Foreign.Ptr (Ptr, plusPtr, castPtr)
    import Foreign.Storable (poke, pokeByteOff, pokeElemOff, peekByteOff, ↙
        ↳ peekElemOff)
    import System.IO.Unsafe (unsafeInterleaveIO)
```

```
import Data.Bits.Bitwise (packWord8LE, mask)
import qualified Data.Bits.Bitwise as Bitwise

import Data.Array.BitArray.Internal
  ( IOBitArray(..)
  , getBounds
  , newArray_
  , freeze
  , unsafeFreeze
  , thaw
  , unsafeThaw
  , copy
  )

-- | Create a new array filled with an initial value.
{-# INLINE newArray #-}
newArray :: Ix i => (i, i) {- ^ bounds -} -> Bool {- ^ initial value -} -> IO (↙
    ↳ IOBitArray i)
newArray bs b = do
  a <- newArray_ bs
  fill a b
  return a

-- | Create a new array filled with values from a list.
{-# INLINE newListArray #-}
newListArray :: Ix i => (i, i) {- ^ bounds -} -> [Bool] {- ^ elems -} -> IO (↙
    ↳ IOBitArray i)
newListArray bs es = do
  a <- newArray_ bs
  let byteBits = 8
      writeBytes :: Ptr Word8 -> [Bool] -> IO ()
      writeBytes p (b0:b1:b2:b3:b4:b5:b6:b7:rest) = do
        poke p (packWord8LE b0 b1 b2 b3 b4 b5 b6 b7)
        writeBytes (plusPtr p 1) rest
      writeBytes _ [] = return ()
      writeBytes p rest = writeBytes p (take byteBits (rest ++ repeat False))
  withForeignPtr (iobData a) $ \p -> do
    writeBytes (castPtr p) (take (byteBits * iobBytes a) es)
  return a

-- | Read from an array at an index.
{-# INLINE readArray #-}
readArray :: Ix i => IOBitArray i -> i -> IO Bool
readArray a i = do
  bs <- getBounds a
  when (not (inRange bs i)) $ error "array index out of bounds"
  readArrayRaw a (index bs i)

-- | Read from an array at an index without bounds checking.  Unsafe.
{-# INLINE unsafeReadArray #-}
unsafeReadArray :: Ix i => IOBitArray i -> i -> IO Bool
unsafeReadArray a i = do
  bs <- getBounds a
  readArrayRaw a (index bs i)

{-# INLINE readArrayRaw #-}
```

```
readArrayRaw :: Ix i => IOBitArray i -> Int -> IO Bool
readArrayRaw a n = do
  let byte = n `shiftR` 3
      bit = n .&. 7
  withForeignPtr (iobData a) $ \p -> do
    b0 <- peekByteOff p byte
    return (testBit (b0 :: Word8) bit)

-- | Write to an array at an index.
{-# INLINE writeArray #-}
writeArray :: Ix i => IOBitArray i -> i -> Bool -> IO ()
writeArray a i b = do
  bs <- getBounds a
  when (not (inRange bs i)) $ error "array index out of bounds"
  let n = index bs i
      byte = n `shiftR` 3
      bit = n .&. 7
  withForeignPtr (iobData a) $ \p -> do
    b0 <- peekByteOff p byte
    let b1 = (if b then setBit else clearBit) (b0 :: Word8) bit
    pokeByteOff p byte b1

-- | Alias for 'map'.
{-# INLINE mapArray #-}
mapArray :: Ix i => (Bool -> Bool) -> IOBitArray i -> IO (IOBitArray i)
mapArray = map

-- unsafeInterleaveIO is used to avoid having to create the whole list in
-- memory before the function can return, but need to keep the ForeignPtr
-- alive to avoid GC stealing our data.
interleavedMapMThenTouch :: Ix i => IOBitArray i -> (a -> IO b) -> [a] -> IO [b]
interleavedMapMThenTouch a _ [] = touchForeignPtr (iobData a) >> return []
interleavedMapMThenTouch a f (x:xs) = unsafeInterleaveIO $ do
  y <- f x
  ys <- interleavedMapMThenTouch a f xs
  return (y:ys)

-- | Create a new array by reading from another.
{-# INLINE mapIndices #-}
mapIndices :: (Ix i, Ix j) => (i, i) {- ^ new bounds -} -> (i -> j) {- ^ index ↙
    ↳ transformation -} -> IOBitArray j {- ^ source array -} -> IO (IOBitArray i↙
    ↳ )
mapIndices bs h a = newListArray bs =<< interleavedMapMThenTouch a (readArray a ↙
    ↳ . h) (range bs)

-- | Get a list of all elements of an array.
{-# INLINE getElems #-}
getElems :: Ix i => IOBitArray i -> IO [Bool]
getElems a = unsafeGetElems =<< copy a

-- | Get a list of all elements of an array.  Unsafe when the source
--   array can be modified later.
{-# INLINE unsafeGetElems #-}
unsafeGetElems :: Ix i => IOBitArray i -> IO [Bool]
unsafeGetElems a' = do
  bs <- getBounds a'
  let r = rangeSize bs
```

```
      count = (r + 7) `shiftR` 3
    p <- withForeignPtr (iobData a') $ return
    bytes <- interleavedMapMThenTouch a' (peekByteOff p) [0 .. count - 1]
    return . take r . concatMap Bitwise.toListLE $ (bytes :: [Word8])

-- | Get a list of all (index, element) pairs.
{-# INLINE getAssocs #-}
getAssocs :: Ix i => IOBitArray i -> IO [(i, Bool)]
getAssocs a = do
  bs <- getBounds a
  zip (range bs) `fmap` getElems a

-- | Fill an array with a uniform value.
{-# INLINE fill #-}
fill :: Ix i => IOBitArray i -> Bool -> IO ()
fill a b = do
  let count = iobBytes a `shiftR` 3
      word :: Word64
      word = if b then complement 0 else 0
  withForeignPtr (iobData a) $ \p ->
    forM_ [0 .. count - 1] $ \i ->
      pokeElemOff p i word

-- | Short-circuit bitwise reduction: True when any bit is True.
{-# INLINE or #-}
or :: Ix i => IOBitArray i -> IO Bool
or a = do
  bs <- getBounds a
  let total = rangeSize bs
      full = total .&. complement (mask 6)
      count = full `shiftR` 6
      loop :: Ptr Word64 -> Int -> IO Bool
      loop p n
        | n < count = do
            w <- peekElemOff p n
            if w /= (0 :: Word64) then return True else loop p (n + 1)
        | otherwise = rest full
      rest m
        | m < total = do
            b <- readArrayRaw a m
            if b then return True else rest (m + 1)
        | otherwise = return False
  withForeignPtr (iobData a) $ \p -> loop p 0

-- | Short-circuit bitwise reduction: False when any bit is False.
{-# INLINE and #-}
and :: Ix i => IOBitArray i -> IO Bool
and a = do
  bs <- getBounds a
  let total = rangeSize bs
      full = total .&. complement (mask 6)
      count = full `shiftR` 6
      loop :: Ptr Word64 -> Int -> IO Bool
      loop p n
        | n < count = do
            w <- peekElemOff p n
            if w /= (complement 0 :: Word64) then return False else loop p (n + ↙
```

```
                ↳ 1)
        | otherwise = rest full
      rest m
        | m < total = do
            b <- readArrayRaw a m
            if not b then return False else rest (m + 1)
        | otherwise = return True
  withForeignPtr (iobData a) $ \p -> loop p 0

-- | Short-circuit bitwise reduction: 'Nothing' when any bits differ,
--   'Just' when all bits are the same.
{-# INLINE isUniform #-}
isUniform :: Ix i => IOBitArray i -> IO (Maybe Bool)
isUniform a = do
  bs <- getBounds a
  let total = rangeSize bs
      full = total .&. complement (mask 6)
      count = full `shiftR` 6
      loop :: Ptr Word64 -> Int -> Bool -> Bool -> IO (Maybe Bool)
      loop p n st sf
        | n < count = do
            w <- peekElemOff p n
            let t = w /= (0 :: Word64)  || st
                f = w /= (complement 0) || sf
            if t && f then return Nothing else loop p (n + 1) t f
        | otherwise = rest full st sf
      rest m st sf
        | m < total = do
            b <- readArrayRaw a m
            let t =     b || st
                f = not b || sf
            if t && f then return Nothing else rest (m + 1) t f
        | st && not sf = return (Just True)
        | not st && sf = return (Just False)
        | otherwise = return Nothing
  withForeignPtr (iobData a) $ \p -> loop p 0 False False

-- | Look up index of first matching bit.
--
--   Note that the index type is limited to Int because there
--   is no 'unindex' method in the 'Ix' class.
{-# INLINE elemIndex #-}
elemIndex :: Bool -> IOBitArray Int -> IO (Maybe Int)
elemIndex which a = do
  bs <- getBounds a
  let skip :: Word64
      skip | which = 0
           | otherwise = complement 0
      total = rangeSize bs
      full = total .&. complement (mask 6)
      count = full `shiftR` 6
      loop :: Ptr Word64 -> Int -> IO (Maybe Int)
      loop p n
        | n < count = do
            w <- peekElemOff p n
            if w /= skip then rest (n `shiftL` 6) else loop p (n + 1)
        | otherwise = rest full
```

```
        rest m
          | m < total = do
              b <- readArrayRaw a m
              if b == which then return (Just (fst bs + m)) else rest (m + 1)
          | otherwise = return Nothing
    withForeignPtr (iobData a) $ \p -> loop p 0

-- | Bitwise reduction with an associative commutative boolean operator.
--   Implementation lifts from 'Bool' to 'Bits' and folds large chunks
--   at a time.  Each bit is used as a source exactly once.
{-# INLINE fold #-}
fold :: Ix i => (Bool -> Bool -> Bool) {- ^ operator -} -> IOBitArray i -> IO (↙
    ↳ Maybe Bool)
fold f a = do
  bs <- getBounds a
  let g = Bitwise.zipWith f
      total = rangeSize bs
      full = total .&. complement (mask 6)
      count = full `shiftR` 6
      loop :: Ptr Word64 -> Int -> Maybe Word64 -> IO (Maybe Bool)
      loop p n mw
        | n < count = do
            w <- peekElemOff p n
            case mw of
              Nothing -> loop p (n + 1) (Just $!      w)
              Just w0 -> loop p (n + 1) (Just $! g w0 w)
        | otherwise =
            case mw of
              Nothing -> rest full Nothing
              Just w0 -> rest full (Just $! foldl1' f (Bitwise.toListLE w0))
      rest m mb
        | m < total = do
            b <- readArrayRaw a m
            case mb of
              Nothing -> rest (m + 1) (Just $!      b)
              Just b0 -> rest (m + 1) (Just $! f b0 b)
        | otherwise = return mb
  withForeignPtr (iobData a) $ \p -> loop p 0 Nothing

-- | Bitwise map.  Implementation lifts from 'Bool' to 'Bits' and maps
--   large chunks at a time.
{-# INLINE map #-}
map :: Ix i => (Bool -> Bool) -> IOBitArray i -> IO (IOBitArray i)
map f a = do
  bs <- getBounds a
  b <- newArray_ bs
  mapTo b f a
  return b

{-# INLINE mapTo #-}
mapTo :: Ix i => IOBitArray i -> (Bool -> Bool) -> IOBitArray i -> IO ()
mapTo dst f src = do
  -- {
  sbs <- getBounds src
  dbs <- getBounds dst
  when (sbs /= dbs) $ error "mapTo mismatched bounds"
  -- }
```

```
340     let count = iobBytes dst `shiftR` 3
            g :: Word64 -> Word64
            g = Bitwise.map f
        withForeignPtr (iobData src) $ \sp ->
          withForeignPtr (iobData dst) $ \dp ->
345         forM_ [0 .. count - 1] $ \n -> do
              pokeElemOff dp n . g =<< peekElemOff sp n

    -- | Bitwise zipWith.  Implementation lifts from 'Bool' to 'Bits' and
    --   combines large chunks at a time.
350 --
    --   The bounds of the source arrays must be identical.
    {-# INLINE zipWith #-}
    zipWith :: Ix i => (Bool -> Bool -> Bool) -> IOBitArray i -> IOBitArray i -> IO ↙
        ↳ (IOBitArray i)
    zipWith f l r = do
355   lbs <- getBounds l
      rbs <- getBounds r
      when (lbs /= rbs) $ error "zipWith mismatched bounds"
      c <- newArray_ lbs
      zipWithTo c f l r
360   return c

    {-# INLINE zipWithTo #-}
    zipWithTo :: Ix i => IOBitArray i -> (Bool -> Bool -> Bool) -> IOBitArray i -> ↙
        ↳ IOBitArray i -> IO ()
    zipWithTo dst f l r = do
365   lbs <- getBounds l
      rbs <- getBounds r
      dbs <- getBounds dst
      when (lbs /= rbs || dbs /= lbs || dbs /= rbs) $ error "zipWithTo mismatched ↙
          ↳ bounds"
      let count = iobBytes dst `shiftR` 3
370       g :: Word64 -> Word64 -> Word64
          g = Bitwise.zipWith f
      withForeignPtr (iobData l) $ \lp ->
        withForeignPtr (iobData r) $ \rp ->
          withForeignPtr (iobData dst) $ \dp ->
375           forM_ [0 .. count - 1] $ \n -> do
                p <- peekElemOff lp n
                q <- peekElemOff rp n
                pokeElemOff dp n (g p q)

380 -- | Count set bits.
    {-# INLINE popCount #-}
    popCount :: Ix i => IOBitArray i -> IO Int
    popCount a = do
      bs <- getBounds a
385   let total = rangeSize bs
          full = total .&. complement (mask 6)
          count = full `shiftR` 6
          loop :: Ptr Word64 -> Int -> Int -> IO Int
          loop p n acc
390         | n < count = acc `seq` do
                w <- peekElemOff p n
                loop p (n + 1) (acc + Data.Bits.popCount w)
            | otherwise = rest full acc
```

```
      rest m acc
395     | m < total = acc `seq` do
            b <- readArrayRaw a m
            rest (m + 1) (acc + fromEnum b)
        | otherwise = return acc
  withForeignPtr (iobData a) $ \p -> loop p 0 0
```

12 src/Data/Array/BitArray/ST.hs

```
{-|

Module      :  Data.Array.BitArray.ST
Copyright   :  (c) Claude Heiland-Allen 2012,2018
License     :  BSD3

Maintainer  :  claude@mathr.co.uk
Stability   :  unstable
Portability :  uses ST

Unboxed mutable bit arrays in the 'ST' monad.

-}
-- almost all is implemented with unsafeIOToST and the IO-based implementation
module Data.Array.BitArray.ST
  ( STBitArray()
  -- * MArray-like interface.
  , getBounds
  , newArray
  , newArray_
  , newListArray
  , readArray
  , writeArray
  , mapArray
  , mapIndices
  , getElems
  , getAssocs
  -- * Conversion to/from immutable bit arrays.
  , freeze
  , thaw
  -- * Construction.
  , copy
  , fill
  -- * Short-circuiting reductions.
  , or
  , and
  , isUniform
  , elemIndex
  -- * Aggregate operations.
  , fold
  , map
  , zipWith
  , popCount
  -- * Unsafe.
  , unsafeReadArray
  , unsafeGetElems
  , unsafeFreeze
  , unsafeThaw
```

```
  ) where

import Prelude hiding (and, or, map, zipWith)
import Control.Monad.ST (ST)
import Data.Ix (Ix)

import Control.Monad.ST.Unsafe (unsafeIOToST)
import Data.Array.BitArray.Internal (BitArray)
import Data.Array.BitArray.IO (IOBitArray)
import qualified Data.Array.BitArray.IO as IO

-- | The type of mutable bit arrays.
newtype STBitArray s i = STB (IOBitArray i)

-- | Get the bounds of a bit array.
{-# INLINE getBounds #-}
getBounds :: Ix i => STBitArray s i -> ST s (i, i)
getBounds (STB a) = unsafeIOToST (IO.getBounds a)

-- | Create a new array filled with an initial value.
{-# INLINE newArray #-}
newArray :: Ix i => (i, i) {- ^ bounds -} -> Bool {- ^ initial value -} -> ST s ↙
    ↳ (STBitArray s i)
newArray bs b = STB `fmap` unsafeIOToST (IO.newArray bs b)

-- | Create a new array filled with a default initial value ('False').
{-# INLINE newArray_ #-}
newArray_ :: Ix i => (i, i) {- ^ bounds -} -> ST s (STBitArray s i)
newArray_ bs = STB `fmap` unsafeIOToST (IO.newArray bs False)

-- | Create a new array filled with values from a list.
{-# INLINE newListArray #-}
newListArray :: Ix i => (i, i) {- ^ bounds -} -> [Bool] {- ^ elems -} -> ST s (↙
    ↳ STBitArray s i)
newListArray bs es = STB `fmap` unsafeIOToST (IO.newListArray bs es)

-- | Read from an array at an index.
{-# INLINE readArray #-}
readArray :: Ix i => STBitArray s i -> i -> ST s Bool
readArray (STB a) i = unsafeIOToST (IO.readArray a i)

-- | Read from an array at an index without bounds checking.  Unsafe.
{-# INLINE unsafeReadArray #-}
unsafeReadArray :: Ix i => STBitArray s i -> i -> ST s Bool
unsafeReadArray (STB a) i = unsafeIOToST (IO.unsafeReadArray a i)

-- | Write to an array at an index.
{-# INLINE writeArray #-}
writeArray :: Ix i => STBitArray s i -> i -> Bool -> ST s ()
writeArray (STB a) i b = unsafeIOToST (IO.writeArray a i b)

-- | Alias for 'map'.
{-# INLINE mapArray #-}
mapArray :: Ix i => (Bool -> Bool) -> STBitArray s i -> ST s (STBitArray s i)
mapArray = map

-- | Create a new array by reading from another.
```

```
    {-# INLINE mapIndices #-}
105 mapIndices :: (Ix i, Ix j) => (i, i) {- ^ new bounds -} -> (i -> j) {- ^ index ↙
        ↳ transformation -} -> STBitArray s j {- ^ source array -} -> ST s (↙
        ↳ STBitArray s i)
    mapIndices bs h (STB a) = STB `fmap` unsafeIOToST (IO.mapIndices bs h a)

    -- | Get a list of all elements of an array.
    {-# INLINE getElems #-}
110 getElems :: Ix i => STBitArray s i -> ST s [Bool]
    getElems (STB a) = unsafeIOToST (IO.getElems a)

    -- | Get a list of all elements of an array without copying.  Unsafe when
    --   the source array can be modified later.
115 {-# INLINE unsafeGetElems #-}
    unsafeGetElems :: Ix i => STBitArray s i -> ST s [Bool]
    unsafeGetElems (STB a) = unsafeIOToST (IO.unsafeGetElems a)

    -- | Get a list of all (index, element) pairs.
120 {-# INLINE getAssocs #-}
    getAssocs :: Ix i => STBitArray s i -> ST s [(i, Bool)]
    getAssocs (STB a) = unsafeIOToST (IO.getAssocs a)

    -- | Snapshot the array into an immutable form.
125 {-# INLINE freeze #-}
    freeze :: Ix i => STBitArray s i -> ST s (BitArray i)
    freeze (STB a) = unsafeIOToST (IO.freeze a)

    -- | Snapshot the array into an immutable form.  Unsafe when the source
130 --   array can be modified later.
    {-# INLINE unsafeFreeze #-}
    unsafeFreeze :: Ix i => STBitArray s i -> ST s (BitArray i)
    unsafeFreeze (STB a) = unsafeIOToST (IO.unsafeFreeze a)

135 -- | Convert an array from immutable form.
    {-# INLINE thaw #-}
    thaw :: Ix i => BitArray i -> ST s (STBitArray s i)
    thaw a = STB `fmap` unsafeIOToST (IO.thaw a)

140 -- | Convert an array from immutable form.  Unsafe to modify the result
    --   unless the source array is never used later.
    {-# INLINE unsafeThaw #-}
    unsafeThaw :: Ix i => BitArray i -> ST s (STBitArray s i)
    unsafeThaw a = STB `fmap` unsafeIOToST (IO.unsafeThaw a)
145
    -- | Copy an array.
    {-# INLINE copy #-}
    copy :: Ix i => STBitArray s i -> ST s (STBitArray s i)
    copy (STB a) = STB `fmap` unsafeIOToST (IO.copy a)
150
    -- | Fill an array with a uniform value.
    {-# INLINE fill #-}
    fill :: Ix i => STBitArray s i -> Bool -> ST s ()
    fill (STB a) b = unsafeIOToST (IO.fill a b)
155
    -- | Short-circuit bitwise reduction: True when any bit is True.
    {-# INLINE or #-}
    or :: Ix i => STBitArray s i -> ST s Bool
```

```
or (STB a) = unsafeIOToST (IO.or a)

-- | Short-circuit bitwise reduction: False when any bit is False.
{-# INLINE and #-}
and :: Ix i => STBitArray s i -> ST s Bool
and (STB a) = unsafeIOToST (IO.and a)

-- | Short-circuit bitwise reduction: 'Nothing' when any bits differ,
--   'Just' when all bits are the same.
{-# INLINE isUniform #-}
isUniform :: Ix i => STBitArray s i -> ST s (Maybe Bool)
isUniform (STB a) = unsafeIOToST (IO.isUniform a)

-- | Look up index of first matching bit.
--
--   Note that the index type is limited to Int because there
--   is no 'unindex' method in the 'Ix' class.
{-# INLINE elemIndex #-}
elemIndex :: Bool -> STBitArray s Int -> ST s (Maybe Int)
elemIndex b (STB a) = unsafeIOToST (IO.elemIndex b a)

-- | Bitwise reduction with an associative commutative boolean operator.
--   Implementation lifts from 'Bool' to 'Bits' and folds large chunks
--   at a time.  Each bit is used as a source exactly once.
{-# INLINE fold #-}
fold :: Ix i => (Bool -> Bool -> Bool) {- ^ operator -} -> STBitArray s i -> ST ↙
    ↳ s (Maybe Bool)
fold f (STB a) = unsafeIOToST (IO.fold f a)

-- | Bitwise map.  Implementation lifts from 'Bool' to 'Bits' and maps
--   large chunks at a time.
{-# INLINE map #-}
map :: Ix i => (Bool -> Bool) -> STBitArray s i -> ST s (STBitArray s i)
map f (STB a) = STB `fmap` unsafeIOToST (IO.map f a)

-- | Bitwise zipWith.  Implementation lifts from 'Bool' to 'Bits' and
--   combines large chunks at a time.
--
--   The bounds of the source arrays must be identical.
{-# INLINE zipWith #-}
zipWith :: Ix i => (Bool -> Bool -> Bool) -> STBitArray s i -> STBitArray s i ->↙
    ↳ ST s (STBitArray s i)
zipWith f (STB a) (STB b) = STB `fmap` unsafeIOToST (IO.zipWith f a b)

-- | Count set bits.
{-# INLINE popCount #-}
popCount :: Ix i => STBitArray s i -> ST s Int
popCount (STB a) = unsafeIOToST (IO.popCount a)
```

13 src/Data/Bits/Bitwise.hs

```
{-# LANGUAGE PatternGuards #-}
{-|

Module      :  Data.Bits.Bitwise
Copyright   :  (c) Claude Heiland-Allen 2012
License     :  BSD3
```

```
    Maintainer  :  claude@mathr.co.uk
    Stability   :  unstable
    Portability :  portable

    Lifting boolean operations on 'Bool' to bitwise operations on 'Bits'.

    Packing bits into words, and unpacking words into bits.

    -}
    module Data.Bits.Bitwise
      (
      -- * Boolean operations lifted to bitwise operations.
        repeat
      , map
      , zipWith
      , or
      , and
      , any
      , all
      , isUniform
      -- * Splitting\/joining 'Bits' to\/from (lsb, msb).
      , mask
      , splitAt
      , joinAt
      , fromBool
      -- * (Un)packing 'Bits' to\/from lists of 'Bool'.
      , fromListLE
      , toListLE
      , fromListBE
      , toListBE
      -- * (Un)packing 'Word8' to\/from 8-tuples of 'Bool'.
      , packWord8LE
      , unpackWord8LE
      , packWord8BE
      , unpackWord8BE
      ) where

    import Prelude hiding (repeat, map, zipWith, any, all, or, and, splitAt)
    import qualified Prelude as P

    import Data.Bits (Bits(complement, (.&.), (.|.), xor, bit, shiftL, shiftR, ↙
        ↳ testBit, bitSizeMaybe, zeroBits),
                      FiniteBits(finiteBitSize))
    import Data.List (foldl')
    import Data.Word (Word8)

    -- | Lift a boolean constant to a bitwise constant.
    {-# INLINE repeat #-}
    repeat :: (Bits b) => Bool -> b
    repeat False = zeroBits
    repeat True = complement zeroBits

    -- | Lift a unary boolean operation to a bitwise operation.
    --
    --   The implementation is by exhaustive input\/output case analysis:
    --   thus the operation provided must be total.
```

```
--
{-# INLINE map #-}
map :: (Bits b) => (Bool -> Bool) {- ^ operation -} -> b -> b
map f = case (f False, f True) of
  (False, False) -> \_ -> zeroBits
  (False, True ) -> id
  (True,  False) -> complement
  (True,  True ) -> \_ -> complement zeroBits

-- | Lift a binary boolean operation to a bitwise operation.
--
--   The implementation is by exhaustive input\/output case analysis:
--   thus the operation provided must be total.
--
{-# INLINE zipWith #-}
zipWith :: (Bits b) => (Bool -> Bool -> Bool) {- ^ operation -} -> b -> b -> b
zipWith f = case (f False False, f False True, f True False, f True True) of
  (False, False, False, False) -> \_ _ -> zeroBits
  (False, False, False, True ) -> (.&.)
  (False, False, True,  False) -> \x y -> x .&. complement y
  (False, False, True,  True ) -> \x _ -> x
  (False, True,  False, False) -> \x y -> complement x .&. y
  (False, True,  False, True ) -> \_ y -> y
  (False, True,  True,  False) -> xor
  (False, True,  True,  True ) -> (.|.)
  (True,  False, False, False) -> \x y -> complement (x .|. y)
  (True,  False, False, True ) -> \x y -> complement (x `xor` y)
  (True,  False, True,  False) -> \_ y -> complement y
  (True,  False, True,  True ) -> \x y -> x .|. complement y
  (True,  True,  False, False) -> \x _ -> complement x
  (True,  True,  False, True ) -> \x y -> complement x .|. y
  (True,  True,  True,  False) -> \x y -> complement (x .&. y)
  (True,  True,  True,  True ) -> \_ _ -> complement zeroBits

-- zipWith3 would have 256 cases? not sure..

-- | True when any bit is set.
{-# INLINE or #-}
or  :: (Bits b) => b -> Bool
or  b = b /= zeroBits

-- | True when all bits are set.
{-# INLINE and #-}
and :: (Bits b) => b -> Bool
and b = b == complement zeroBits

-- | True when the predicate is true for any bit.
{-# INLINE any #-}
any :: (Bits b) => (Bool -> Bool) {- ^ predicate -} -> b -> Bool
any f = or  . map f

-- | True when the predicate is true for all bits.
{-# INLINE all #-}
all :: (Bits b) => (Bool -> Bool) {- ^ predicate -} -> b -> Bool
all f = and . map f

-- | Determine if a 'Bits' is all 1s, all 0s, or neither.
```

```
{-# INLINE isUniform #-}
isUniform :: (Bits b) => b -> Maybe Bool
isUniform b
  | b == zeroBits            = Just False
  | b == complement zeroBits = Just True
  | otherwise                = Nothing

-- | A mask with count least significant bits set.
{-# INLINE mask #-}
mask :: (Num b, Bits b) => Int {- ^ count -} -> b
mask n = bit n - bit 0

-- | Split a word into (lsb, msb).  Ensures lsb has no set bits
--   above the split point.
{-# INLINE splitAt #-}
splitAt :: (Num b, Bits b) => Int {- ^ split point -} -> b {- ^ word -} -> (b, b↙
    ↳ ) {- ^ (lsb, msb) -}
splitAt n b = (b .&. mask n, b `shiftR` n)

-- | Join lsb with msb to make a word.  Assumes lsb has no set bits
--   above the join point.
{-# INLINE joinAt #-}
joinAt :: (Bits b) => Int {- ^ join point -} -> b {- ^ least significant bits -}↙
    ↳  -> b {- ^ most significant bits -} -> b {- ^ word -}
joinAt n lsb msb = lsb .|. (msb `shiftL` n)

-- | Pack bits into a byte in little-endian order.
{-# INLINE packWord8LE #-}
packWord8LE :: Bool {- ^ least significant bit -} -> Bool -> Bool -> Bool -> ↙
    ↳ Bool -> Bool -> Bool -> Bool {- ^ most significant bit -} -> Word8
packWord8LE  a b c d e f g h = z a 1 .|. z b 2 .|. z c 4 .|. z d 8 .|. z e 16 ↙
    ↳ .|. z f 32 .|. z g 64 .|. z h 128
  where z False _ = 0
        z True  n = n

-- | Pack bits into a byte in big-endian order.
{-# INLINE packWord8BE #-}
packWord8BE :: Bool {- ^ most significant bit -} -> Bool -> Bool -> Bool -> Bool↙
    ↳  -> Bool -> Bool -> Bool {- ^ least significant bit -} -> Word8
packWord8BE a b c d e f g h = packWord8LE h g f e d c b a

-- | Extract the bits from a byte in little-endian order.
{-# INLINE unpackWord8LE #-}
unpackWord8LE :: Word8 -> (Bool, Bool, Bool, Bool, Bool, Bool, Bool, Bool) {- ^ ↙
    ↳ (least significant bit, ..., most significant bit) -}
unpackWord8LE w = (b 1, b 2, b 4, b 8, b 16, b 32, b 64, b 128)
  where b z = w .&. z /= 0

-- | Extract the bits from a byte in big-endian order.
{-# INLINE unpackWord8BE #-}
unpackWord8BE :: Word8 -> (Bool, Bool, Bool, Bool, Bool, Bool, Bool, Bool) {- ^ ↙
    ↳ (most significant bit, ..., least significant bit) -}
unpackWord8BE w = (b 128, b 64, b 32, b 16, b 8, b 4, b 2, b 1)
  where b z = w .&. z /= 0

-- | The least significant bit.
{-# INLINE fromBool #-}
```

```
fromBool :: (Bits b) => Bool -> b
fromBool False = zeroBits
fromBool True  = bit 0

-- | Convert a little-endian list of bits to 'Bits'.
{-# INLINE fromListLE #-}
fromListLE :: (Bits b) => [Bool] {- ^ \[least significant bit, ..., most ↙
    ↳ significant bit\] -} -> b
fromListLE = foldr f zeroBits
  where
    f b i = fromBool b .|. (i `shiftL` 1)

-- | Convert a 'Bits' to a list of bits, in
--   little-endian order.
{-# INLINE toListLE #-}
toListLE :: (Bits b) => b -> [Bool] {- ^ \[least significant bit, ..., most ↙
    ↳ significant bit\] -}
toListLE b0 | Just n <- bitSizeMaybe b0 = P.map (testBit b0) [0..n-1]
            | otherwise = go b0
  where go b | zeroBits == b = []
             | otherwise = testBit b 0 : go (b `shiftR` 1)

-- | Convert a big-endian list of bits to 'Bits'.
{-# INLINE fromListBE #-}
fromListBE :: (Bits b) => [Bool] {- ^ \[most significant bit, ..., least ↙
    ↳ significant bit\] -} -> b
fromListBE = foldl' f zeroBits
  where
    f i b = (i `shiftL` 1) .|. fromBool b

-- | Convert a 'FiniteBits' to a list of bits, in
--   big-endian order.
{-# INLINE toListBE #-}
toListBE :: (FiniteBits b) => b -> [Bool] {- ^ \[most significant bit, ..., ↙
    ↳ least significant bit\] -}
toListBE b = P.map (testBit b) [finiteBitSize b - 1, finiteBitSize b - 2 .. 0]
```