Home * Board Representation * Bitboards * Flipping, Mirroring and Rotating

Barbara Mittman, Forever IV [1]

Flipping, Mirroring and Rotating
might be useful to transform bitboards in various ways. Considering the fourfold symmetry of the chessboard, the first paragraph covers the whole bitboard performing bit-twiddling-techniques.

Various multiplications to flip or mirror certain subsets of the bitboard like files, ranks and even diagonals and anti-diagonals are mentioned in the second section.

Table of Contents

The whole Bitboard

Following C-routines perform parallel prefix algorithms to swap bits in various ways. They are not intended to use extensively inside a chess program - e.g. to implement rotated or reverse bitboards on the fly - but may be used for initialization purposes. They may convert sets between the eight different square mappings, considering rank-file or file-rank endianness. An exception might be the vertical flipping, which could be done by one fast x86-64 byteswap (bswap) instruction [2] . See the bswap-application of hyperbola quintessence to determine vertical or diagonal sliding attacks. Flipping, mirroring and rotating is distributive over the basic bitwise operations such as intersection, union, one's complement and xor, as demonstrated in hyperbola quintessence as well.

Beside portable routines, there are optimized routines taking advantage of compiler intrinsics, to use x86-64 processor instructions like byteswap (bswap) as mentioned and rotate (ror, rol) [3] .

Flip and Mirror

This is about

- flipping vertically
- mirroring horizontally
- flipping along the diagonal
- flipping along the antidiagonal

```
. 1 1 1 1 . . .   . 1 1 1 1 . . .   . 1 1 1 1 . . .   . 1 1 1 1 . . .
. 1 . . . 1 . .   . 1 . . . 1 . .   . 1 . . . 1 . .   . 1 . . . 1 . .
. 1 . . . 1 . .   . 1 . . . 1 . .   . 1 . . . 1 . .   . 1 . . . 1 . .
. 1 . . 1 . . .   . 1 . . 1 . . .   . 1 . . 1 . . .   . 1 . . 1 . . .
. 1 1 1 . . . .   . 1 1 1 . . . .   . 1 1 1 . . . .   . 1 1 1 . . . .
. 1 . 1 . . . .   . 1 . 1 . . . .   . 1 . 1 . . . .   . 1 . 1 . . . .
. 1 . . 1 . . .   . 1 . . 1 . . .   . 1 . . 1 . . .   . 1 . . 1 . . .
. 1 . . . 1 . .   . 1 . . . 1 . .   . 1 . . . 1 . .   . 1 . . . 1 . .
       -                 |                 /               \
 flipVertical     mirrorHorizontal  flipDiagA1H8      flipDiagA8H1
       -                 |               /                   \
. 1 . . . 1 . .   . . . 1 1 1 1 .   . . . . . . . .   . . . . . . . .
. 1 . . 1 . . .   . . 1 . . . 1 .   . . . . . . . .   1 1 1 1 1 1 1 1
. 1 . 1 . . . .   . . 1 . . . 1 .   1 . . . . 1 1 .   1 . . . 1 . . .
. 1 1 1 . . . .   . . . 1 . . 1 .   . 1 . . 1 . . 1   1 . . . 1 1 . .
. 1 . . 1 . . .   . . . . 1 1 1 .   . . 1 1 . . . 1   1 . . 1 . . 1 .
. 1 . . . 1 . .   . . . . 1 . 1 .   . . . 1 . . . 1   . 1 1 . . . . 1
. 1 . . . 1 . .   . . . 1 . . 1 .   1 1 1 1 1 1 1 1   . . . . . . . .
. 1 1 1 1 . . .   . . 1 . . . 1 .   . . . . . . . .   . . . . . . . .
```

Vertical

```
. 1 1 1 1 . . .    . 1 . . . 1 . .
. 1 . . . 1 . .    . 1 . . 1 . . .
. 1 . . . 1 . .    . 1 . 1 . . . .
. 1 . . 1 . . .    . 1 1 1 . . . .
. 1 1 1 . . . .    . 1 . . 1 . . .
. 1 . 1 . . . .    . 1 . . . 1 . .
. 1 . . 1 . . .    . 1 . . . 1 . .
. 1 . . . 1 . .    . 1 1 1 1 . . .
```

Flipping a bitboard **vertical** reverses all eight bytes - it performs a little- big-endian conversion or vice versa. A scalar square may be flipped vertically by xor 56.


```
sq' = sq ^ 56;
```

The obvious approach with 21 operations, note that the shifts by 56 don't need masks:

```
/**
 * Flip a bitboard vertically about the centre ranks.
 * Rank 1 is mapped to rank 8 and vice versa.
 * @param x any bitboard
 * @return bitboard x flipped vertically
 */
U64 flipVertical(U64 x) {
    return  ( (x << 56)                           ) |
            ( (x << 40) & C64(0x00ff000000000000) ) |
            ( (x << 24) & C64(0x0000ff0000000000) ) |
            ( (x <<  8) & C64(0x000000ff00000000) ) |
            ( (x >>  8) & C64(0x00000000ff000000) ) |
            ( (x >> 24) & C64(0x0000000000ff0000) ) |
            ( (x >> 40) & C64(0x000000000000ff00) ) |
            ( (x >> 56) );
}
```

The parallel prefix-approach takes 13 operations, to swap bytes, words and double-words in a SWAR-wise manner, performing three delta swaps:

```
/**
 * Flip a bitboard vertically about the centre ranks.
 * Rank 1 is mapped to rank 8 and vice versa.
 * @param x any bitboard
 * @return bitboard x flipped vertically
 */
U64 flipVertical(U64 x) {
   const U64 k1 = C64(0x00FF00FF00FF00FF);
   const U64 k2 = C64(0x0000FFFF0000FFFF);
   x = ((x >>  8) & k1) | ((x & k1) <<  8);
   x = ((x >> 16) & k2) | ((x & k2) << 16);
   x = ( x >> 32)       | ( x       << 32);
   return x;
}
```

Using the x86-64 _byteswap_uint64 or bswap64 intrinsics only takes one processor instruction in 64-bit mode.

```
/**
 * Flip a bitboard vertically about the centre ranks.
 * Rank 1 is mapped to rank 8 and vice versa.
 * @param x any bitboard
 * @return bitboard x flipped vertically
 */
U64 flipVertical(U64 x) {
   return _byteswap_uint64(x);
}
```

Horizontal

```
. 1 1 1|1 . . .     . . . 1 1 1 1 .
. 1 . . . 1 . .     . . 1 . . . 1 .
. 1 . . . 1 . .     . . 1 . . . 1 .
. 1 . . 1 . . .     . . . 1 . . 1 .
. 1 1 1 . . . .     . . . . 1 1 1 .
. 1 . 1 . . . .     . . . . 1 . 1 .
. 1 . . 1 . . .     . . . 1 . . 1 .
. 1 . .|. 1 . .     . . 1 . . . 1 .
```

Horizontal mirroring reverses the bits of each byte. A scalar square may be mirrored horizontally by xor 7.

```
sq' = sq ^ 7;
```

The parallel prefix-algorithm swaps bits, bit-duos and nibbles in a SWAR-wise manner, performing three delta swaps, 15 operations:

```
/**
 * Mirror a bitboard horizontally about the center files.
 * File a is mapped to file h and vice versa.
 * @param x any bitboard
 * @return bitboard x mirrored horizontally
 */
U64 mirrorHorizontal (U64 x) {
   const U64 k1 = C64(0x5555555555555555);
   const U64 k2 = C64(0x3333333333333333);
   const U64 k4 = C64(0x0f0f0f0f0f0f0f0f);
   x = ((x >> 1) & k1) | ((x & k1) << 1);
   x = ((x >> 2) & k2) | ((x & k2) << 2);
   x = ((x >> 4) & k4) | ((x & k4) << 4);
   return x;
}
```

Replacing bitwise 'or' of disjoint sets by 'add' and shift left by appropriate multiply - taking advantage of x86 lea instruction for multiplication with 2 and 4:

```
/**
 * Mirror a bitboard horizontally about the center files.
 * File a is mapped to file h and vice versa.
 * @param x any bitboard
 * @return bitboard x mirrored horizontally
 */
U64 mirrorHorizontal (U64 x) {
   const U64 k1 = C64(0x5555555555555555);
   const U64 k2 = C64(0x3333333333333333);
   const U64 k4 = C64(0x0f0f0f0f0f0f0f0f);
   x = ((x >> 1) & k1) +  2*(x & k1);
   x = ((x >> 2) & k2) +  4*(x & k2);
   x = ((x >> 4) & k4) + 16*(x & k4);
```

```
    return x;
}
```


Using rotates instead of shifts in some clever way takes 13 operations (introduced by Steffan Westcott). Disadvantage is each operation depends on the previous one, while the lea-approach gains some parallelism with same number of instructions.

```
/**
 * Mirror a bitboard horizontally about the center files.
 * File a is mapped to file h and vice versa.
 * @param x any bitboard
 * @return bitboard x mirrored horizontally
 */
U64 mirrorHorizontal (U64 x) {
   const U64 k1 = C64(0x5555555555555555);
   const U64 k2 = C64(0x3333333333333333);
   const U64 k4 = C64(0x0f0f0f0f0f0f0f0f);
   x ^= k4 & (x ^ rotateLeft(x, 8));
   x ^= k2 & (x ^ rotateLeft(x, 4));
   x ^= k1 & (x ^ rotateLeft(x, 2));
   return rotateRight(x, 7);
}
```

Generalized

In Knuth's *The Art of Computer Programming* , *Volume 4, Fascicle 1: Bitwise tricks & techniques*, page 9, Knuth interprets the *magic masks* as 2-adic fractions [4] :

```
k1 = ...1010101010101010101010101010101 = -1/3
k2 = ...1001100110011001100110011001100110011 = -1/5
k4 = ...1000011110000111100001111000011111 = -1/17
```

How the masks look on a chess board:

```
k1                 k2                 k4
-1/3               -1/5               -1/17
0x5555555555555555  0x3333333333333333  0x0F0F0F0F0F0F0F0F
1 . 1 . 1 . 1 .    1 1 . . 1 1 . .    1 1 1 1 . . . .
1 . 1 . 1 . 1 .    1 1 . . 1 1 . .    1 1 1 1 . . . .
1 . 1 . 1 . 1 .    1 1 . . 1 1 . .    1 1 1 1 . . . .
1 . 1 . 1 . 1 .    1 1 . . 1 1 . .    1 1 1 1 . . . .
1 . 1 . 1 . 1 .    1 1 . . 1 1 . .    1 1 1 1 . . . .
1 . 1 . 1 . 1 .    1 1 . . 1 1 . .    1 1 1 1 . . . .
1 . 1 . 1 . 1 .    1 1 . . 1 1 . .    1 1 1 1 . . . .
1 . 1 . 1 . 1 .    1 1 . . 1 1 . .    1 1 1 1 . . . .
```

A parametrized flip, mirror or reverse (mirror and flip) might be generalized to let the compiler produce mentioned routines with flip or mirror as constant (or template) parameter. Otherwise, without compile time constants, the division at runtime is too expensive.

```
/**
 * Flip, mirror or reverse a bitboard
 * @param x any bitboard
 * @param flip the bitboard
 * @param mirror the bitboard
 * @return bitboard x flipped, mirrored or reversed
 */
U64 flipMirrorOrReverse(U64 x, bool flip, bool mirror)
{
   for (U32 i = 3*(1-mirror); i < 3*(1+flip); i++) {
      int s(      1  << i);
      U64 f( C64( 1) << s);
      U64 k( C64(-1) / (f+1) );
      x = ((x >> s) & k) + f*(x & k);
   }
   return x;
}
```

The loop variable 'i' runs from following ranges based on the boolean (0,1) parameters 'flip' and 'mirror':

```
mirror:   for (U32 i = 0; i < 3; i++)
flip:     for (U32 i = 3; i < 6; i++)
reverse := mirror && flip
          for (U32 i = 0; i < 6; i++)
```

With following formula for the delta swaps constants ...

```
shift(i)  := s(i) := 1 << i
factor(i) := f(i) := 1 << s(i)
mask(i)   := k(i) := -1 / (f(i) + 1)
```

... and therefor following values for i = 0...5:

```
s(0)  1
f(0)  2
k(0)  0x5555555555555555 = 0xffffffffffffffff / (2+1)

s(1)  2
f(1)  4
k(1)  0x3333333333333333 = 0xffffffffffffffff / (4+1)

s(2)  4
f(2)  16
k(2)  0x0f0f0f0f0f0f0f0f = 0xffffffffffffffff / (16+1)

s(3)  8
f(3)  256
k(3)  0x00ff00ff00ff00ff = 0xffffffffffffffff / (256+1)

s(4)  16
f(4)  65536
k(4)  0x0000ffff0000ffff = 0xffffffffffffffff / (65536+1)

s(5)  32
f(5)  4294967296
k(5)  0x00000000ffffffff = 0xffffffffffffffff / (4294967296+1)
```

Diagonal

Flip about the Diagonal

```
. 1 1 1 1 . . /     . . . . . . . .
. 1 . . . 1 . .     . . . . . . . .
. 1 . . . 1 . .     1 . . . . 1 1 .
. 1 . . 1 . . .     . 1 . . 1 . . 1
. 1 1 1 . . . .     . . 1 1 . . . 1
. 1 . 1 . . . .     . . . 1 . . . 1
. 1 . . 1 . . .     1 1 1 1 1 1 1 1
/ 1 . . . 1 . .     . . . . . . . .
```

A scalar square needs to swap rank and file.

```
sq' = ((sq >> 3) | (sq << 3)) & 63;
sq' = (sq * 0x20800000) >>> 26; // unsigned 32-bit shift, zeros no sign bits from left
```

Performing three [delta swaps]:

```
/**
 * Flip a bitboard about the diagonal a1-h8.
 * Square h1 is mapped to a8 and vice versa.
 * @param x any bitboard
 * @return bitboard x flipped about diagonal a1-h8
 */
U64 flipDiagA1H8(U64 x) {
   U64 t;
   const U64 k1 = C64(0x5500550055005500);
   const U64 k2 = C64(0x3333000033330000);
   const U64 k4 = C64(0x0f0f0f0f00000000);
   t  = k4 & (x ^ (x << 28));
   x ^=       t ^ (t >> 28) ;
   t  = k2 & (x ^ (x << 14));
   x ^=       t ^ (t >> 14) ;
   t  = k1 & (x ^ (x <<  7));
   x ^=       t ^ (t >>  7) ;
   return x;
}
```

How the masks look on a chess board:

```
k1                  k2                  k4
0x5500550055005500  0x3333000033330000  0x0F0F0F0F00000000
1 . 1 . 1 . 1 .     1 1 . . 1 1 . .     1 1 1 1 . . . .
. . . . . . . .     1 1 . . 1 1 . .     1 1 1 1 . . . .
1 . 1 . 1 . 1 .     . . . . . . . .     1 1 1 1 . . . .
. . . . . . . .     . . . . . . . .     1 1 1 1 . . . .
1 . 1 . 1 . 1 .     1 1 . . 1 1 . .     . . . . . . . .
. . . . . . . .     1 1 . . 1 1 . .     . . . . . . . .
1 . 1 . 1 . 1 .     . . . . . . . .     . . . . . . . .
. . . . . . . .     . . . . . . . .     . . . . . . . .
```

Anti-Diagonal

Flip about the Anti-Diagonal

```
\ 1 1 1 1 . . .     . . . . . . . .
. 1 . . . 1 . .     1 1 1 1 1 1 1 1
. 1 . . . 1 . .     1 . . . 1 . . .
. 1 . . 1 . . .     1 . . . 1 1 . .
. 1 1 1 . . . .     1 . . 1 . . 1 .
. 1 . 1 . . . .     . 1 1 . . . . 1
. 1 . . 1 . . .     . . . . . . . .
. 1 . . . 1 . \     . . . . . . . .
```

Flip about the [Anti-Diagonal] results in the bit-reversal of flip about the Diagonal.
Thus, a scalar square needs not only swap rank and file, but also xor 63.

```
sq' = (((sq >> 3) | (sq << 3)) & 63) ^ 63;
sq' = ((sq * 0x20800000) >>> 26) ^ 63;  // unsigned 32-bit shift
```

Performing three [delta swaps]:

```
/**
 * Flip a bitboard about the antidiagonal a8-h1.
 * Square a1 is mapped to h8 and vice versa.
 * @param x any bitboard
 * @return bitboard x flipped about antidiagonal a8-h1
 */
U64 flipDiagA8H1(U64 x) {
   U64 t;
   const U64 k1 = C64(0xaa00aa00aa00aa00);
   const U64 k2 = C64(0xcccc0000cccc0000);
   const U64 k4 = C64(0xf0f0f0f00f0f0f0f);
   t  =       x ^ (x << 36) ;
   x ^= k4 & (t ^ (x >> 36));
   t  = k2 & (x ^ (x << 18));
   x ^=       t ^ (t >> 18) ;
   t  = k1 & (x ^ (x <<  9));
   x ^=       t ^ (t >>  9) ;
   return x;
}
```

How the masks look on a chess board:

```
k1                  k2                  k4
0xAA00AA00AA00AA00  0xCCCC0000CCCC0000  0xF0F0F0F00F0F0F0F
. 1 . 1 . 1 . 1     . . 1 1 . . 1 1     . . . . 1 1 1 1
. . . . . . . .     . . 1 1 . . 1 1     . . . . 1 1 1 1
. 1 . 1 . 1 . 1     . . . . . . . .     . . . . 1 1 1 1
. . . . . . . .     . . . . . . . .     . . . . 1 1 1 1
. 1 . 1 . 1 . 1     . . 1 1 . . 1 1     1 1 1 1 . . . .
. . . . . . . .     . . 1 1 . . 1 1     1 1 1 1 . . . .
```

```
. 1 . 1 . 1 . 1     . . . . . . . .     1 1 1 1 . . . .
. . . . . . . .     . . . . . . . .     1 1 1 1 . . . .
```


Rotating

This is about

- Rotation by 180 degrees
- Rotation by 90 degrees Clockwise
- Rotation by 90 degrees Anti-Clockwise

Rotation can be deduced from flipping and mirroring in various ways.

```
                    mirrorHorizontal     flipDiagA1H8        flipDiagA8H1
. 1 1 1 1 . . .     . . . 1 1 1 1 .     . . . . . . . .     . . . . . . . .
. 1 . . . 1 . .     . . 1 . . . 1 .     . . . . . . . .     1 1 1 1 1 1 1 1
. 1 . . . 1 . .     . . 1 . . . 1 .     1 . . . . 1 1 .     1 . . . 1 . . .
. 1 . . 1 . . .     . . . 1 . . 1 .     . 1 . . 1 . . 1     1 . . . 1 1 . .
. 1 1 1 . . . .     . . . . 1 1 1 .     . . 1 1 . . . 1     1 . . 1 . . 1 .
. 1 . 1 . . . .     . . . . 1 . 1 .     . . . 1 . . . 1     . 1 1 . . . . 1
. 1 . . 1 . . .     . . . 1 . . 1 .     1 1 1 1 1 1 1 1     . . . . . . . .
. 1 . . . 1 . .     . . 1 . . . 1 .     . . . . . . . .     . . . . . . . .
 flipVertical         rotate180        rotate90clockwise rotate90antiClockwise
. 1 . . . 1 . .     . . 1 . . . 1 .     . . . . . . . .     . . . . . . . .
. 1 . . 1 . . .     . . . 1 . . 1 .     1 1 1 1 1 1 1 1     . . . . . . . .
. 1 . 1 . . . .     . . . . 1 . 1 .     . . . 1 . . . 1     . 1 1 . . . . 1
. 1 1 1 . . . .     . . . . 1 1 1 .     . . 1 1 . . . 1     1 . . 1 . . 1 .
. 1 . . 1 . . .     . . . 1 . . 1 .     . 1 . . 1 . . 1     1 . . . 1 1 . .
. 1 . . . 1 . .     . . 1 . . . 1 .     1 . . . . 1 1 .     1 . . . 1 . . .
. 1 . . . 1 . .     . . 1 . . . 1 .     . . . . . . . .     1 1 1 1 1 1 1 1
. 1 1 1 1 . . .     . . . 1 1 1 1 .     . . . . . . . .     . . . . . . . .
```

By 180 degrees - Bit-Reversal

```
. 1 1 1 1 . . .     . . 1 . . . 1 .
. 1 . . . 1 . .     . . . 1 . . 1 .
. 1 . . . 1 . .     . . . . 1 . 1 .
. 1 . . 1 . . .     . . . . 1 1 1 .
. 1 1 1 . . . .     . . . 1 . . 1 .
. 1 . 1 . . . .     . . 1 . . . 1 .
. 1 . . 1 . . .     . . 1 . . . 1 .
. 1 . . . 1 . .     . . . 1 1 1 1 .
```

Rotating by 180 degrees reverses all bits in a bitboard. A scalar square may be reversed by xor 63:

```
sq' = sq ^ 63; // 63 - sq;
```

Deduced from flipping vertically and mirroring horizontally. Both operation may be applied in different orders.

```
/**
 * Rotate a bitboard by 180 degrees.
 * Square a1 is mapped to h8, and a8 is mapped to h1.
 * @param x any bitboard
 * @return bitboard x rotated 180 degrees
 */
U64 rotate180 (U64 x) {
   return mirrorHorizontal (flipVertical (x) );
}
```

The resulting code applies six delta swaps:

```
/**
 * Rotate a bitboard by 180 degrees.
 * Square a1 is mapped to h8, and a8 is mapped to h1.
 * @param x any bitboard
 * @return bitboard x rotated 180 degrees
 */
U64 rotate180 (U64 x) {
   const U64 h1 = C64 (0x5555555555555555);
   const U64 h2 = C64 (0x3333333333333333);
   const U64 h4 = C64 (0x0F0F0F0F0F0F0F0F);
   const U64 v1 = C64 (0x00FF00FF00FF00FF);
   const U64 v2 = C64 (0x0000FFFF0000FFFF);
   x = ((x >>  1) & h1) | ((x & h1) <<  1);
   x = ((x >>  2) & h2) | ((x & h2) <<  2);
   x = ((x >>  4) & h4) | ((x & h4) <<  4);
   x = ((x >>  8) & v1) | ((x & v1) <<  8);
   x = ((x >> 16) & v2) | ((x & v2) << 16);
   x = ( x >> 32)       | ( x       << 32);
   return x;
}
```

By 90 degrees Clockwise

```
. 1 1 1 1 . . .     . . . . . . . .
. 1 . . . 1 . .     1 1 1 1 1 1 1 1
. 1 . . . 1 . .     . . . 1 . . . 1
. 1 . . 1 . . .     . . 1 1 . . . 1
. 1 1 1 . . . .     . 1 . . 1 . . 1
. 1 . 1 . . . .     1 . . . . 1 1 .
. 1 . . 1 . . .     . . . . . . . .
. 1 . . . 1 . .     . . . . . . . .
```

A scalar square swaps rank and file plus xor 56:

```
sq' = (((sq >> 3) | (sq << 3)) & 63) ^ 56;
sq' = ((sq * 0x20800000) >>> 26) ^ 56; // unsigned 32-bit shift
```

Deduced from flipping vertically and flipping along the antidiagonal.

```
/**
 * Rotate a bitboard by 90 degrees clockwise.
```

```
 * @param x any bitboard
 * @return bitboard x rotated 90 degrees clockwise
```



```
U64 rotate90clockwise (U64 x) {
   return flipVertical (flipDiagA1H8 (x) );
}
```

Note that

```
flipVertical (flipDiagA1H8 (x) ) == flipDiagA8H1 (flipVertical (x) )
```

By 90 degrees Anti-Clockwise

```
. 1 1 1 1 . . .     . . . . . . . .
. 1 . . . 1 . .     . . . . . . . .
. 1 . . . 1 . .     . 1 1 . . . . 1
. 1 . . 1 . . .     1 . . 1 . . 1 .
. 1 1 1 . . . .     1 . . . 1 1 . .
. 1 . 1 . . . .     1 . . . 1 . . .
. 1 . . 1 . . .     1 1 1 1 1 1 1 1
. 1 . . . 1 . .     . . . . . . . .
```

A scalar square swaps rank and file plus xor 7:

```
sq' = (((sq >> 3) | (sq << 3)) & 63) ^ 7;
sq' = ((sq * 0x20800000) >>> 26) ^ 7;  // unsigned 32-bit shift
```

Deduced from flipping vertically and flipping along the diagonal.

```
/**
 * Rotate a bitboard by 90 degrees anticlockwise.
 * @param x any bitboard
 * @return bitboard x rotated 90 degrees anticlockwise
 */
U64 rotate90antiClockwise (U64 x) {
   return flipDiagA1H8 (flipVertical (x) );
}
```

Note that

```
flipDiagA1H8 (flipVertical (x) ) == flipVertical (flipDiagA8H1 (x) )
```

Pseudo-Rotation by 45 degrees

The chess-board is four-fold symmetry - thus there is no real 45 degree rotation in the mathematical sense. Anyway we may map diagonals and anti-diagonals to ranks, similar to rotated bitboards.

Clockwise

The 15 diagonals are enumerated by (file - rank) - nibble wise two's complement F == (16) -1, 9 == (16) -7. Two shorter diagonals are file-aligned packed into one rank each. Note that the three lower bits are equal in each rank and bit three (the sign-bit inside a signed nibble) indicates a "negative" diagonal north the main diagonal.

```
9 A B C D E F 0     9 1 1 1 1 1 1 1
A B C D E F 0 1     A A 2 2 2 2 2 2
B C D E F 0 1 2     B B B 3 3 3 3 3
C D E F 0 1 2 3     C C C C 4 4 4 4
D E F 0 1 2 3 4     D D D D D 5 5 5
E F 0 1 2 3 4 5     E E E E E E 6 6
F 0 1 2 3 4 5 6     F F F F F F F 7
0 1 2 3 4 5 6 7     0 0 0 0 0 0 0 0
```

We need to rotate the A-H files vertically by 0 to 7 ranks - done parallel prefix wise. One square is therefor mapped by:

```
sq' = (sq + 8*(sq&7)) & 63;
```

The main-diagonal is rotated clockwise to the first rank - thus 45 degrees.

On using xor, see bits from two sources by a mask. See rotate for the intrinsic routines.

```
/**
 * Pseudo rotate a bitboard 45 degree clockwise.
 * Main Diagonal is mapped to 1st rank
 * @param x any bitboard
 * @return bitboard x rotated
 */
U64 pseudoRotate45clockwise (U64 x) {
   const U64 k1 = C64(0xAAAAAAAAAAAAAAAA);
   const U64 k2 = C64(0xCCCCCCCCCCCCCCCC);
   const U64 k4 = C64(0xF0F0F0F0F0F0F0F0);
   x ^= k1 & (x ^ rotateRight(x,  8));
   x ^= k2 & (x ^ rotateRight(x, 16));
   x ^= k4 & (x ^ rotateRight(x, 32));
   return x;
}
```

Another pseudo rotation scheme was introduced by Robert Hyatt's rotated bitboards approach. While the 45 degree rotation looks more natural at the first glance, the calculations of this mapping is more complicated.

```
normal chess board bitmap          45 clockwise               45 clockwise crafty

A8 B8 C8 D8 E8 F8 G8 H8                                       C7 D8|A6 B7 C8|A7 B8|A8|
A7 B7 C7 D7 E7 F7 G7 H7                  A8                   F8|A4 B5 C6 D7 E8|A5 B6
A6 B6 C6 D6 E6 F6 G6 H6                A7  B8                 E6 F7 G8|A3 B4 C5 D6 E7
A5 B5 C5 D5 E5 F5 G5 H5              A6  B7  C8               E5 F6 G7 H8|A2 B3 C4 D5
A4 B4 C4 D4 E4 F4 G4 H4            A5  B6  C7  D8             E4 F5 G6 H7|A1 B2 C3 D4
A3 B3 C3 D3 E3 F3 G3 H3          A4  B5  C6  D7  E8           D2 E3 F4 G5 H6|B1 C2 D3
A2 B2 C2 D2 E2 F2 G2 H2        A3  B4  C5  D6  E7  F8        |G3 H4|D1 E2 F3 G4 H5|C1
A1 B1 C1 D1 E1 F1 G1 H1      A2  B3  C4  D5  E6  F7  G8       H1|G1 H2|F1 G2 H3|E1 F2
                           A1  B2  C3  D4  E5  F6  G7  H8
A8|B1 C2 D3 E4 F5 G6 H7      B1  C2  D3  E4  F5  G6  H7       A8|B1 C2 D3 E4 F5 G6 H7
A7 B8|C1 D2 E3 F4 G5 H6        C1  D2  E3  F4  G5  H6         A7 B8|C1 D2 E3 F4 G5 H6
A6 B7 C8|D1 E2 F3 G4 H5          D1  E2  F3  G4  H5           A6 B7 C8|D1 E2 F3 G4 H5
A5 B6 C7 D8|E1 F2 G3 H4            E1  F2  G3  H4             A5 B6 C7 D8|E1 F2 G3 H4
```

```
A4 B5 C6 D7 E8|F1 G2 H3             F1  G2  H3              A4 B5 C6 D7 E8|F1 G2 H3
A3 B4 C5 D6 E7 F8|G1 H2               G1  H2                A3 B4 C5 D6 E7 F8|G1 H2
A2 B3 C4 D5 E6 F7 G8|H1                  H1                 A2 B3 C4 D5 E6 F7 G8|H1
A1 B2 C3 D4 E5 F6 G7 H8                                     A1 B2 C3 D4 E5 F6 G7 H8

45 clockwise                                                45 clockwise
```

A parallel prefix solution to map the normal occupancy to the visual rotated approach is left to the ambitious reader.

Anti-Clockwise

We enumerate the 15 anti-diagonals by 7 ^ (file + rank), a nibble wise two's complement F == -1.

```
0 F E D C B A 9     1 1 1 1 1 1 1 9
1 0 F E D C B A     2 2 2 2 2 2 A A
2 1 0 F E D C B     3 3 3 3 3 B B B
3 2 1 0 F E D C     4 4 4 4 C C C C
4 3 2 1 0 F E D     5 5 5 D D D D D
5 4 3 2 1 0 F E     6 6 E E E E E E
6 5 4 3 2 1 0 F     7 F F F F F F F
7 6 5 4 3 2 1 0     0 0 0 0 0 0 0 0
```

We need to rotate the A-H files vertically by 7 to 0 ranks - done parallel prefix wise.
One square is therefor mapped by:

```
sq' = (sq + 8*((sq&7)^7)) & 63;
```

The main anti-diagonal is rotated anti-clockwise to the first rank - thus 45 degrees. The shorter anti-diagonals are pairwise and properly file aligned packed into further ranks.

On using xor see bits from two sources by a mask. See rotate for the intrinsic routines.

```
/**
 * Pseudo rotate a bitboard 45 degree anti clockwise.
 * Main AntiDiagonal is mapped to 1st rank
 * @param x any bitboard
 * @return bitboard x rotated
 */
U64 pseudoRotate45antiClockwise (U64 x) {
   const U64 k1 = C64(0x5555555555555555);
   const U64 k2 = C64(0x3333333333333333);
   const U64 k4 = C64(0x0f0f0f0f0f0f0f0f);
   x ^= k1 & (x ^ rotateRight(x,  8));
   x ^= k2 & (x ^ rotateRight(x, 16));
   x ^= k4 & (x ^ rotateRight(x, 32));
   return x;
}
```

Rank, File and Diagonal

Those subsets may be flipped or mirrored in a more efficient way by multiplication-techniques with disjoint intermediate sums and no internal overflows. Unlike the whole board techniques, multiplication doesn't swap bits, but maps file to a rank or vice versa in different steps, which can not be combined in one step. Main application is to map file- or diagonal occupancies to ranks, to dense the line-occupancy to consecutive bits, further elaborated in Occupancy of any Line or Kindergarten Bitboards.

Flip about the Anti-Diagonal

Flipping about the anti-diagonal multiplies with the main-diagonal. Note that the set bits in the main-diagonal have a distance of 9 (2^0, 2^9,2^18,...,2^54, 2^63). We can therefor safely multiply it with a rank-pattern with 8 consecutive neighboring bits without overflows.

File to a Rank

Multiplying the masked A-file with the main-diagonal, maps the file-bits to the 8th rank, similar to a flip about the anti-diagonal A8-H1. Shifting down to the 1st rank, leaves the bits like a 90-degree anti clockwise rotation.

```
masked bits                         mapped to 8th rank
bits on A-file   *  main-diagonal   =  with garbage     -> 1st rank
A . . . . . . .     . . . . . . . 1    A B C D E F G H     . . . . . . . .
B . . . . . . .     . . . . . . 1 .    B C D E F G H .     . . . . . . . .
C . . . . . . .     . . . . . 1 . .    C D E F G H . .     . . . . . . . .
D . . . . . . .     . . . . 1 . . .    D E F G H . . .  >> . . . . . . . .
E . . . . . . .  *  . . . 1 . . . . =  E F G H . . . .  56 . . . . . . . .
F . . . . . . .     . . 1 . . . . .    F G H . . . . .     . . . . . . . .
G . . . . . . .     . 1 . . . . . .    G H . . . . . .     . . . . . . . .
H . . . . . . .     1 . . . . . . .    H . . . . . . .     A B C D E F G H
```

Rank to File

Multiplying the masked 1st rank with the main-diagonal, maps the rank-bits to the H-file, similar to a flip about the anti-diagonal A8-H1. Shifting the H-file to the A-file plus mask acts like a 90-degree clockwise rotation.

```
masked bits                         mapped to H-file
bits on 1st rank *  main-diagonal   =  with garbage     -> masked A-file
. . . . . . . .     . . . . . . . 1    C D E F G H . A     A . . . . . . .
. . . . . . . .     . . . . . . 1 .    D E F G H . A B     B . . . . . . .
. . . . . . . .     . . . . . 1 . .    E F G H . A B C  >> C . . . . . . .
. . . . . . . .     . . . . 1 . . .    F G H . A B C D  7  D . . . . . . .
. . . . . . . .  *  . . . 1 . . . . =  G H . A B C D E  &  E . . . . . . .
. . . . . . . .     . . 1 . . . . .    H . A B C D E F  A  F . . . . . . .
. . . . . . . .     . 1 . . . . . .    . A B C D E F G     G . . . . . . .
A B C D E F G H     1 . . . . . . .    A B C D E F G H     H . . . . . . .
```

Flip about the Diagonal

Flipping about the Diagonal is a bit more tricky. Since the Anti-Diagonal pattern as factor has the set bits with distance of 7 only (2^0,2^7, 2^14,...2^49, 2^56) with possible overflows, if multiplied with rank pattern of eight neighboring bits. Thus it can only be used to flip the H-file about the diagonal.

File to a Rank

Multiplying the masked H-file with the 7-bit right shifted (board left shifted) anti-diagonal, maps the file-bits to the 8th rank, similar to a flip about the diagonal A1-H8. Shifting it down to the 1st rank, leaves the bits like flip about the diagonal. Shifting down to the 1st rank, leaves the bits like a 90-degree clockwise rotation.

```
masked bits         Shifted             mapped to 8th rank
bits on H-file   *  anti-diagonal   =  with garbage     -> 1st rank
. . . . . . . A     . . . . . . . .    H G F E D C B A     . . . . . . . .
. . . . . . . B     . 1 . . . . . .    . H G F E D C B     . . . . . . . .
. . . . . . . C     . . 1 . . . . .    . . H G F E D C     . . . . . . . .
```

```
. . . . . . . D     . . . 1 . . . .     . . . H G F E D  >> . . . . . . . .
. . . . . . . E  *  . . . . 1 . . .  =  . . . . H G F E  56 . . . . . . . .
. . . . . . . F     . . . . . 1 . .     . . . . . H G F     . . . . . . . .
. . . . . . . G     . . . . . . 1 .     . . . . . . H G     . . . . . . . .
/ . . . . . . H     1 . . . . . . 1     . . . . . . . H     H G F E D C B A
```


Diagonals to Ranks
That is straight forward multiplication of a masked diagonal or anti-diagonal with the A-file.
To mask the garbage off, we further shift down by 7 ranks.

```
masked diagonal  *  A-file              mapped
                                        to 8th rank      ->  1st rank
. . . . . . . H     1 . . . . . . .     A B C D E F G H     . . . . . . . .
. . . . . . G .     1 . . . . . . .     A B C D E F G .     . . . . . . . .
. . . . . F . .     1 . . . . . . .     A B C D E F . .  >> . . . . . . . .
. . . . E . . .     1 . . . . . . .     A B C D E . . .  56 . . . . . . . .
. . . D . . . .  *  1 . . . . . . .  =  A B C D . . . .     . . . . . . . .
. . C . . . . .     1 . . . . . . .     A B C . . . . .     . . . . . . . .
. B . . . . . .     1 . . . . . . .     A B . . . . . .     . . . . . . . .
A . . . . . . .     1 . . . . . . .     A . . . . . . .     A B C D E F G H
```

Mirror Horizontally
This is about bit-reversal of a byte.

```
rank1mirrored = (((rank1 * 0x80200802) & 0x0884422110) * 0x0101010101010101) >> 56;
```

This is how it works on a chessboard:

```
rank1            *  0x80200802          flipped
. . . . . . . .     . . . . . . . .     . . . . . . . .
. . . . . . . .     . . . . . . . .     . . . . . . . .
. . . . . . . .     . . . . . . . .     . . . . . . . .
. . . . . . . .     . . . . . . . .     B C D E F G H .
. . . . . . . .  *  . . . . . . . 1  =  D E F G H . . A
. . . . . . . .     . . . . . 1 . .     F G H . . A B C
. . . . . . . .     . . . 1 . . . .     H . . A B C D E
A B C D E F G H     . 1 . . . . . .     . A B C D E F G

flipped          &  0x0884422110        reversedOnFiles
. . . . . . . .     . . . . . . . .     . . . . . . . .
. . . . . . . .     . . . . . . . .     . . . . . . . .
. . . . . . . .     . . . . . . . .     . . . . . . . .
B C D E F G H .     . . . 1 . . . .     . . . E . . . .
D E F G H . . A  &  . . 1 . . . . 1  =  . . F . . . . A
F G H . . A B C     . 1 . . . . 1 .     . G . . . . B .
H . . A B C D E     1 . . . . 1 . .     H . . . . C . .
. A B C D E F G     . . . . 1 . . .     . . . . D . . .

reversedOnFiles  *  A-file              reversedOn8         reversed
. . . . . . . .     1 . . . . . . .     H G F E D C B A     . . . . . . . .
. . . . . . . .     1 . . . . . . .     H G F E D C B A     . . . . . . . .
. . . . . . . .     1 . . . . . . .     H G F E D C B A     . . . . . . . .
. . . E . . . .     1 . . . . . . .     H G F E D C B A  >> . . . . . . . .
. . F . . . . A  *  1 . . . . . . .  =  H G F . D C B A  56 . . . . . . . .
. G . . . . B .     1 . . . . . . .     H G . . D C B .     . . . . . . . .
H . . . . C . .     1 . . . . . . .     H . . . D C . .     . . . . . . . .
. . . . D . . .     1 . . . . . . .     . . . . D . . .     H G F E D C B A
```

A 32 bit solution:

```
rank1mirrored = ( (rank1 * 0x0802 & 0x22110)
                 |(rank1 * 0x8020 & 0x88440) )
                * 0x10101000 >> 24;
```

Or a simple lookup:

```
rank1mirrored = reverseByteLookup256[rank1];
```

See also
- BMI2 PEXT instruction
- Color Flipping
- Diagonal Mirroring
- Horizontal Mirroring
- Kindergarten Bitboards
- Occupancy of any Line
- Vertical Flipping
- XOP VPPERM instruction

Publications
- Christopher Strachey (**1961**). *Bitwise operations*. Communications of the ACM, Vol. 4, No. 3 [5]
- Henry S. Warren, Jr. (**2002, 2012**). *Hacker's Delight*. Addison-Wesley
- Donald Knuth (**2009**). *The Art of Computer Programming*, *Volume 4, Fascicle 1: Bitwise tricks & techniques*, as Pre-Fascicle 1a postscript

Forum Posts
- Reflection of a bitboard by Harm Geert Muller, CCC, April 22, 2016

External Links
- Bit Gather Via Multiplication by Vlad Petric, Dr. Petric's Technical Blog, September 17, 2013 [6]

Flipping
- Flip from Wikipedia
- Flip (mathematics) from Wikipedia
- Flip from iCoachMath
- Flip, Slide, Turn - Alphabet Geometry from misterteacher.com

Mirroring
- Mirror from Wikipedia
- Mirroring (psychology) from Wikipedia
- Reflection (mathematics) from Wikipedia

- Reflection (physics) from Wikipedia
- Reflection symmetry from Wikipedia
- Venus effect from Wikipedia

- Why do Mirrors Reverse Left and Right?

Rotation

- Rotation from Wikipedia
- Rotation (mathematics) from Wikipedia
- Rotation matrix from Wikipedia
- Rotation formalisms in three dimensions - Wikipedia
- Rotation from iCoachMath
- Oliver Vornberger's German lecture on Graphic Rotation:
  - 2D Rotation
  - 3D Rotation

Reflection

- Reflection (mathematics) from Wikipedia
- Reflection (physics) from Wikipedia
- Reflection symmetry from Wikipedia

Smoke 'n' Mirrors

- Lee Ritenour, Mike Stern, Simon Phillips, John Beasley, Melvin Davis - Smoke 'n' Mirrors [7], Blue Note Tokyo, 2011, YouTube Video [8]

References

1. ^ Figurative Paintings by Barbara Mittman
2. ^ _byteswap_uint64 Visual C++ Developer Center - Run-Time Library Reference
3. ^ _rotl, _rotl64, _rotr_rotr64 Visual C++ Developer Center - Run-Time Library Reference
4. ^ Donald Knuth (**2009**). *The Art of Computer Programming*, *Volume 4, Fascicle 1: Bitwise tricks & techniques*, as Pre-Fascicle 1a postscript
5. ^ reverse.c from C code for most of the programs that appear in HD by Henry S. Warren, Jr.
6. ^ Demystifying the Magic Multiplier?
7. ^ Smoke and mirrors (disambiguation)
8. ^ Shorter front view version with interviews

What links here?

Page	Date Edited
Aleks Peshkov	Jan 13, 2016
Algorithms	May 5, 2017
Anti-Diagonals	Sep 2, 2012
Big-endian	Jan 25, 2015
Bit-Twiddling	Nov 6, 2017
Bitboard Serialization	Dec 24, 2014
Bitboards	Nov 14, 2017
Chessboard	May 10, 2017
Christopher Strachey	May 8, 2017
Color Flipping	May 17, 2017
Crafty	Jan 28, 2018
Diagonal Mirroring	Jun 29, 2013
Diagonals	Jan 16, 2013
Direction	Oct 6, 2016
Efficient Generation of Sliding Piece Attacks	Nov 5, 2016
Endianness	Jan 25, 2015
Flipping Mirroring and Rotating	Oct 14, 2016
General Setwise Operations	Feb 25, 2018
Gk	Oct 9, 2017
Horizontal Mirroring	Jun 29, 2013
Hyperbola Quintessence	Mar 25, 2017
KBNK Endgame	Nov 26, 2016
Kindergarten Bitboards	Aug 1, 2017
Kurt	Apr 20, 2014
Little-endian	Jan 25, 2015
Occupancy	Sep 19, 2016
Occupancy of any Line	Sep 16, 2016
Parallel Prefix Algorithms	Jun 22, 2016
Population Count	Sep 3, 2017
Reverse Bitboards	Aug 29, 2015
Rotated Bitboards	Mar 7, 2017
Rotated Indices	Oct 9, 2017
Ryan Mack	Jan 18, 2011
SIMD and SWAR Techniques	Jun 27, 2017
Squares	Feb 15, 2015
SSSE3	Aug 8, 2017
Vertical Flipping	Oct 10, 2013

Page	Date Edited
Vlad Petric	Oct 25, 2013
Wing	Oct 26, 2017
XOP	Aug 8, 2017

Up one Level

(https://www.wikispaces.com/user/view/workingonchess)

Idea to avoid rotating
workingonchess (https://www.wikispaces.com/user/view/workingonchess) Jan 23, 2012

Hello,

It's seems too easy to be true, but I have an idea to avoid rotating and calculate good moves for sliding pieces. We could put a mask to a sliding piece, this mask will be retrieved with the piece position. Applying this mask (with most 0 a few 1) on the entire chessboard with all pieces, we retrieve 64*64 possibilities. Putting this new bitboard into a hashmap we could retrieve all the moves for the current piece.

As example, we got a rook at c3 square. We put the bitboard with only one 1 at c3 to an hashmap. This hashmap will give a bitboard with four rectangles of 0 and two lines of 1. The lines will be c8 to c1 and a3 to h3. We can now do an AND for all pieces of the chessboard. We will retrieve a bitboard with a almost only 0 and some 0 on the two lines. Now we can put the result into another hashmap which will bring the final moves for the rook. There will be 64*64 items into this hashmap. We could do the same for bishop, and add theses two techniques to compute queen moves. Using this we do not need to rotate bitmap. Where is the problem with this method ? Sorry for bad english and thanks beforehand.

Bretwa

(https://www.wikispaces.com/user/view/GerdIsenberg) GerdIsenberg (https://www.wikispaces.com/user/view/GerdIsenberg) *Jan 23, 2012*
Hi Bretwa,
this page "Flipping Mirroring and Rotating" has nothing to do with move generation. Intersection of occupancy with the one or two lines to lookup attack sets is topic of Sliding Piece Attacks (/Sliding%20Piece%20Attacks), specially the none rotated lookup techniques Kindergarten Bitboards (/Kindergarten%20Bitboards), Hashing Dictionaries (/Hashing%20Dictionaries), Congruent Modulo Bitboards (/Congruent%20Modulo%20Bitboards) and Magic Bitboards (/Magic%20Bitboards).

Gerd

(https://www.wikispaces.com/user/view/workingonchess) workingonchess (https://www.wikispaces.com/user/view/workingonchess) *Jan 25, 2012*
Hi Gerd,

Sorry for that, I have believed it was the right place to deal with rotating efficiency because this article deals with rotating bitboard. Actually the method I have given already exists and is fully described here http://code.google.com/p/shatranjpy/downloads/detail?name=avoiding-rotations-final.pdf&can=2&q= (http://code.google.com/p/shatranjpy/downloads/detail?name=avoiding-rotations-final.pdf&can=2&q=). Probably it could be a good idea to compare on this wiki the good and worse ideas about chess programming in order to give a conspicuous level on what works effectively and what is not.

Bretwa

(https://www.wikispaces.com/user/view/GerdIsenberg) GerdIsenberg (https://www.wikispaces.com/user/view/GerdIsenberg) *Jan 25, 2012*
Hi Bretwa,

No problem, I was even a bit wrong since this page also covers in its second part *Rank, File and Diagonal* the basics of kindergarten bitboard index calculation. Sam Tannous (/Sam%20Tannous) approach and paper is mentioned in Hashing Dictionaries (/Hashing%20Dictionaries). At least in compiled C/C++ a none rotated "and/mul/shift" perfect hashing function ala kindergarten or magic lookups seems hard to beat.

Gerd

(https://www.wikispaces.com/user/view/AGN1964)

45, 135 degree rotations
AGN1964 (https://www.wikispaces.com/user/view/AGN1964) Aug 31, 2011

This wiki and this bitboard rotation page contain some amazing ideas and a lot of low level code that I could not recreate myself. Kudos to all involved.
I am using the Pseudo-Rotation by 45 degrees routine to convert a bitboard to a “horizontal” representation, where I can do some fast processing. Once complete, I need to rotate back to the original position. Due to the different diagonal packing mechanism:
pseudoRotate45antiClockwise(pseudoRotate45clockwise(x) != x
Of course, I'll need the undoPseudoRotate45antiClockwise() next, too.
I am not good enough with binary math to figure out my own undoPseudoRotate45clockwise() function. I've searched, looked at the references, look at What Links Here, but I can only see the two forward rotation functions.
Can anyone point me to the two undo functions I need? Or give a link to tools that might help me devise my own? Thanks.

(https://www.wikispaces.com/user/view/GerdIsenberg) GerdIsenberg (https://www.wikispaces.com/user/view/GerdIsenberg) *Aug 31, 2011*
Using the same routines with rotateLeft inside should do the trick, I guess.

Gerd

(https://www.wikispaces.com/user/view/AGN1964) AGN1964 (https://www.wikispaces.com/user/view/AGN1964) *Sep 2, 2011*
After 5 mins of testing, I'm sure you are correct. I am surprised the answer was so simple!

Thanks very much.

(https://www.wikispaces.com/user/view/GerdIsenberg) GerdIsenberg (https://www.wikispaces.com/user/view/GerdIsenberg) *Sep 2, 2011*
You are welcome!

Curious about your fast processing of the “horizontal” representation, with such a prologue and epilogue to map and re-map the bits. Isn't it cheaper to perform the processing directly in the "diagonal" world?

(https://www.wikispaces.com/user/view/AGN1964) AGN1964 (https://www.wikispaces.com/user/view/AGN1964) *Sep 2, 2011*
I am re-writing my Othello program, for the fourth time; the original is over 30 years old now.
This is the first version using bitboards. This particular question came up during move generation.
Using 2 8-bit integers, representing black and white positions in one row, I have precalculated all the legal moves and stored them in an array; it's quite small [256][256][2]. Move list generation (for a single row) is now a quick look up. For a full board, I need to or the results from all possible rows together, i.e.

from all ranks, files and diagonals. The 8 ranks can be extracted from the bitboards easily, and making the result bit board is simple, too. Then I flip the board along the diagonal, to swap ranks and files and repeat the process; I just flip the result to put the bits back in the correct position. Finally I do something similar for the 45 and 135 diagonals; extracting the diagonals is slightly harder since two diagonals are packed into each rank. Currently, this is 80 times faster than a naive 2D indexing approach.
I probably could extract the 8-bit row integers directly. But while I am using an 8-bit number for the look up, I think I will always need something equivalent to the rotation, masking and shifting to get the bits in need. Perhaps it could be done more efficiently than currently, where 64-bit rotation is a separate operation from 8-bit extraction, but I think that would be a small gain. I think I would need to abandon my look-up table if I stay in the “diagonal world”; perhaps there are efficient algorithms in that space. As I become more experienced with the bitboards, I might re-examine this.
Thanks for the help and the interest.