Table of Contents

Home * Board Representation * Bitboards * Sliding Piece Attacks * **Classical Approach**

The classical approach to generate sliding piece attacks was probably first used by Chess and Kaissa.

Code samples and bitboard diagrams rely on Little endian file and rank mapping.

Ray Attacks

The classical approach works ray-wise and uses pre-calculated ray-attacks for each of the eight ray-directions and each of the 64 squares. It has to distinguish between positive and negative directions, because it has to bitscan the ray-attack intersection with the occupancy in different orders. That usually don't cares for getting line- or piece attacks, since one likely unrolls all directions needed for a particular line or piece - otherwise one may rely on a generalized bitscan.

We rely on the compass rose to enumerate ray-directions:

```
  noWe          nort          noEa
          +7    +8    +9
              \  |  /
  west    -1 <-  0 -> +1    east
              /  |  \
          -9    -8    -7
  soWe          sout          soEa
```

Positive Rays

Attacks of Positive Ray-Directions:

```
East (+1)           North (+8)          NorthEast (+9)      NorthWest
 (+7)
. . . . . . . .     . . . 1 . . . .     . . . . . . . 1     . . . . .
 . . .
. . . . . . . .     . . . 1 . . . .     . . . . . . 1 .     1 . . . .
 . . .
. . . . . . . .     . . . 1 . . . .     . . . . . 1 . .     . 1 . . .
 . . .
. . . . . . . .     . . . 1 . . . .     . . . . 1 . . .     . . 1 . .
 . . .
. . . R 1 1 1 1     . . . R . . . .     . . . B . . . .     . . . B .
 . . .
. . . . . . . .     . . . . . . . .     . . . . . . . .     . . . . .
```

```
  . . .
. . . . . . . . .     . . . . . . . . .     . . . . . . . . .     . . . . . .
  . . .
. . . . . . . . .     . . . . . . . . .     . . . . . . . . .     . . . . . .
  . . .
```

Conditional

The first step gets the ray-attack on the otherwise empty board. Potential blockers are then determined by intersection with the occupancy, the set of all pieces. The first blocker, if any, is the least significant one-bit of the intersection. A bitscan forward determines the square of the first blocker, to reset it's ray-bits from the attack set. For instance a bishop on g2:

```
occupied          &  NorthWest(g2)       {a8, c6}
1 . 1 1 1 1 1 1      1 . . . . . . .      1 . . . . . . .
1 . 1 1 1 1 1 1      . 1 . . . . . .      . . . . . . . .
. 1 1 . . . . .      . . 1 . . . . .      . . 1 . . . . .
. . . . . . . .      . . . 1 . . . .      . . . . . . . .
. . . . . . . .  &   . . . . 1 . . .  =   . . . . . . . .
. . . . . . 1 .      . . . . . 1 . .      . . . . . . . .
1 1 1 1 1 1 B 1      . . . . . . . .      . . . . . . . .
1 1 1 1 1 . 1 1      . . . . . . . .      . . . . . . . .

                     xor NorthWest(c6 = bitscanForward{a8,c6}
                     1 . . . . . . .
                     . 1 . . . . . .
                     . . . . . . . .
                     . . . . . . . .
                     . . . . . . . .
                     . . . . . . . .
                     . . . . . . . .
                     . . . . . . . .

                     =  final northWest Attacks
                     . . . . . . . .
                     . . . . . . . .
                     . . 1 . . . . .
                     . . . 1 . . . .
                     . . . . 1 . . .
                     . . . . . 1 . .
                     . . . . . . . .
                     . . . . . . . .
```

```
U64 rayAttacks[8][64];

U64 getPositiveRayAttacks(
U64 occupied, enumDir dir8, enumSquare square) {
   U64 attacks = rayAttacks[dir8][square];
   U64 blocker = attacks & occupied;
   if ( blocker ) {
      square = bitScanForward(blocker);
      attacks ^= rayAttacks[dir8][square];
   }
   return attacks;
}
```

Branchless

Branches are evil with todays super pipelined processors. Even if branch-prediction heuristics become smarter, branch-less solutions allow better scheduling and parallel speedup of independent instruction chains, like processing several directions.

Considering todays fast x86-64 bsf-instruction of Core 2 or K10, a branch-less solution may be worth a try. Due to the fact the occupancy of the outer squares doesn't affect the attack set, setting the most significant bit ensures to scan at least an outer square, which would address an empty ray set anyway, therefor not affecting the final result with no blocker or a most outer one.

```
U64 getRayAttacks(U64 occupied, enumDir dir8, unsigned long square) {
   U64 attacks    = rayAttacks[dir8][square];
   U64 blocker    =  attacks & occupied;
   _BitScanForward64 (&square, blocker | C64(0x8000000000000000));
   return attacks ^ rayAttacks[dir8][square];
}
```

Negative Rays

Attacks of Negative Ray-Directions:

```
West (-1)           South (-8)           SouthWest (-9)      SouthEast
 (-7)
. . . . . . . . .     . . . . . . . . .      . . . . . . . . .     . . . . . .
 . . .
```

```
. . . . . . . .     . . . . . . . .     . . . . . . . .     . . . . .
 . . .
. . . . . . . .     . . . . . . . .     . . . . . . . .     . . . . .
 . . .
. . . . . . . .     . . . . . . . .     . . . . . . . .     . . . . .
 . . .
1 1 1 R . . . .     . . . R . . . .     . . . B . . . .     . . . B .
 . . .
. . . . . . . .     . . . 1 . . . .     . . 1 . . . . .     . . . . 1
 . . .
. . . . . . . .     . . . 1 . . . .     . 1 . . . . . .     . . . . .
 1 . .
. . . . . . . .     . . . 1 . . . .     1 . . . . . . .     . . . . .
 . 1 .
```

Conditional

Works the same way, but needs reverse bitscan to find the first blocking piece, if any.

```
U64 getNegativeRayAttacks(
U64 occupied, enumDir dir8, enumSquare square) {
   U64 attacks = rayAttacks[dir8][square];
   U64 blocker = attacks & occupied;
   if ( blocker ) {
      square = bitScanReverse(blocker);
      attacks ^= rayAttacks[dir8][square];
   }
   return attacks;
}
```

Branchless

The branch-less solution with a fast x86-64 bsr-instruction allows better scheduling and parallel speedup of independent instruction chains, like processing several directions. Setting the least significant bit ensures to scan at least an outer square, which would address an empty ray set anyway, therefor not affecting the final result with no blocker or a most outer one.

```
U64 getRayAttacks(U64 occupied, enumDir dir8, unsigned long square) {
   U64 attacks    = rayAttacks[dir8][square];
   U64 blocker    =  attacks & occupied;
   _BitScanReverse64 (&square, blocker | 1);
```

```
   return attacks ^ rayAttacks[dir8][square];
}
```

Generalized

The idea of the generalized bitscan may be used to share the same code for all directions. The implementation of isNegative(dir8), a macro or inline-function, depends on the definition or enumeration of the directions and is likely a compare or test instruction.

Conditional

The conditional by a good predictable branch on blocker is favorable - specially for slow bitscans with some chance to skip it, e.g. in endings.

```
U64 getRayAttacks(U64 occupied, enumDir dir8, enumSquare square) {
   U64 attacks = rayAttacks[dir8][square];
   U64 blocker = attacks & occupied;
   if ( blocker ) {
      square = bitScan(blocker, isNegative(dir8));
      attacks ^= rayAttacks[dir8][square];
   }
   return attacks;
}
```

Branchless

The branch-less routine provides a dirMask as universal set for negative rays and the empty set for positive rays, to implement the generalized bitscan. The dirBit-or ensures to scan at least outer square with empty ray sets.

```
U64 getRayAttacks(U64 occupied, enumDir dir8, unsigned long square) {
   U64 attacks    = rayAttacks[dir8][square];
   U64 blocker    =  attacks & occupied;
   blocker       &= -blocker | dirMask[dir8];
   blocker       |=            dirBit [dir8]
   _BitScanReverse64 (&square, blocker | dirBit[dir8]);
   return attacks ^ rayAttacks[dir8][square];
}
```

Zero Count

If available, Leading- or Trailing Zero Count instructions may be used instead of bitscan for another branch-less solution of the classical attack getter. Since they leave 64 for empty sets, it needs to make the ray attack arrays one greater to allow index by 64 which contains an empty set - or one needs to map 64 to 63 for positive directions.

```
U64 rayAttacks[8][65];

U64 getPositiveRayAttacks(
U64 occupied, enumDir dir8, enumSquare square) {
   U64 attacks = rayAttacks[dir8][square];
   U64 blocker = attacks & occupied;
   int firstBlockingSquare = trailingZeroCount(blocker);
   attacks ^= rayAttacks[dir8][firstBlockingSquare];
   return attacks;
}
```

LeadingZeroCount instead of bitscanReverse may be done similarly, considering the reversed order.

Line Attacks

As mentioned, line attacks are the union of positive and opposite negative ray-directions - since they are disjoint one may also use 'xor' or 'add':

```
U64 diagonalAttacks(U64 occ, enumSquare sq) {
  return getPositiveRayAttacks(occ, noEa, sq)
       | getNegativeRayAttacks(occ, soWe, sq); // ^ +
}

U64 antiDiagAttacks(U64 occ, enumSquare sq) {
  return getPositiveRayAttacks(occ, noWe, sq)
       | getNegativeRayAttacks(occ, soEa, sq); // ^ +
}

U64 fileAttacks (U64 occ, enumSquare sq) {
  return getPositiveRayAttacks(occ, nort, sq)
       | getNegativeRayAttacks(occ, sout, sq); // ^ +
}

U64 rankAttacks (U64 occ, enumSquare sq) {
  return getPositiveRayAttacks(occ, east, sq)
```

```
        | getNegativeRayAttacks(occ, west, sq); // ^ +
}
```

Piece Attacks

Union of Line Attacks

Of course piece attacks are the union of the line attacks:

```
U64 rookAttacks (U64 occ, enumSquare sq) {
  return fileAttacks(occ, sq)
        | rankAttacks(occ, sq); // ^ +
}

U64 bishopAttacks (U64 occ, enumSquare sq) {
  return diagonalAttacks(occ, sq)
        | antiDiagAttacks(occ, sq); // ^ +
}

U64 queenAttacks (U64 occ, enumSquare sq) {
  return rookAttacks  (occ, sq)
        | bishopAttacks(occ, sq); // ^ +
}
```

In one Run

As mentioned by Robert Hyatt [1], instead of fetching four ray-attacks on the otherwise empty board, one may already use the rook- or bishop attacks to reset outer squares from that union set.

A further improvement was suggested by Michael Sherwin [2], to union the occupancy with the outer bits 0 and 63. Together with appropriate bits set in separate ray-masks, this yields to an efficient branchless solution with 13 64-bit operations in total and 4.5 KByte for the lookup tables for both rooks and bishops each.

```
struct {
  U64 bitsN;  // bits North, including MSB (bit 63)
  U64 bitsE;  // bits East, including MSB
  U64 bitsS;  // bits South, including LSB (bit 0 == 1)
  U64 bitsW;  // bits West, including LSB
```

```
} CACHE_ALIGN rayWstop[64];

U64 attacksEmpty[64];
U64 rayN[64];
U64 rayE[64];
U64 rayS[64];
U64 rayW[64];

U64 rookAttacks(U64 occ, unsigned int sq) {
   unsigned long ulN, ulE, ulS, ulW;
   occ |= C64(0x8000000000000001);
   _BitScanForward64(&ulN, occ & rayWstop[sq].bitsN);
   _BitScanForward64(&ulE, occ & rayWstop[sq].bitsE);
   _BitScanReverse64(&ulS, occ & rayWstop[sq].bitsS);
   _BitScanReverse64(&ulW, occ & rayWstop[sq].bitsW);
   return attacksEmpty[sq]^rayN[ulN]^rayE[ulE]^rayS[ulS]^rayW[ulW];
}
```

See also

- Bitfoot - A/B Bitboards
- Blockers and Beyond
- Obstruction Difference
- Pieces versus Directions

Publications

- Stuart Cracraft (**1984**). *Bitmap move generation in Chess*. ICCA Journal, Vol. 7, No. 3, pp. 146–153
- Fridel Fainshtein (**2006**). *An Orthodox k-Move Problem-Composer for Chess Directmates*. M.Sc. thesis, Bar-Ilan University, pdf, Appendix D - 64-bit Representation, pp. 105
- Fridel Fainshtein, Yaakov HaCohen-Kerner (**2006**). *A Chess Composer of Two-Move Mate Problems*. ICGA Journal, Vol. 29, No. 1, pdf, Appendix E: 64-bit representation

Forum Posts

- Re: Thinker output by Robert Hyatt, CCC, March 25, 2009
- Re: Yet another bitboard attack generator by Robert Hyatt, CCC, October 28, 2009
- Modified old 64 bit attack getter by Michael Sherwin, CCC, December 06, 2009

References

1. ^ Re: Yet another bitboard attack generator by Robert Hyatt, CCC, October 28, 2009
2. ^ Modified old 64 bit attack getter by Michael Sherwin, CCC, December 06, 2009

What links here?

Page	Date Edited
Andscacs	Jan 16, 2018
Bitfoot	Sep 8, 2015
BitScan	Sep 10, 2017
Blockers and Beyond	Mar 21, 2014
Classical Approach	Jan 28, 2018
DirGolem	Jun 5, 2016
Fridel Fainshtein	Oct 8, 2014
Hiding the Implementation	Feb 3, 2011
Pieces versus Directions	Oct 6, 2016
SEE - The Swap Algorithm	Jun 5, 2017
Shifted Bitboards	Jan 9, 2011
Sliding Piece Attacks	May 27, 2016
Teki	Mar 29, 2018
Tunguska	Sep 16, 2017
Yaakov HaCohen-Kerner	Aug 23, 2014

Up one Level