

Bitboard Serialization (/Bitboard+Serialization)

✖ It's time for us to say farewell... Regretfully, we've made the tough decision to close Wikispaces. Find out why, and what will happen, here (http://blog.wikispaces.com/). 55 (/page/history/Bitboard+Serialization)

... (/page/menu/Bitboard+Serialization)

[Home](#) * [Board Representation](#) * [Bitboards](#) * [Bitboard Serialization](#)



Bitboard Serialization refers to the transformation of a bitboard with up to 64 one-bits set into a list of up to 64 bit-indices aka [square indices](#) of a [8x8 board](#) - for instance to process [move-target](#) sets for [move generation](#). This is done in two phases, isolating none-empty [subsets](#) and then transforming those more versatile subsets into lists, either bit by bit, by applying a [bisection](#) scheme, where finally [words](#) or [bytes](#) may act as index of a pre-calculated database, or by [perfect hashing](#) of square lists by subsets with a limited maximum popularity, for instance move-target sets of a [king](#) or [knight](#) even with [minimal perfect hashing](#).

Table of Contents

[Isolating Subsets](#)

[Single Bits](#)

[Multiple Bits](#)

[Converting Sets to Lists](#)

[Square Index Serialization](#)

[Scanning Forward](#)

[Scanning Reverse](#)

[Scanning with Reset](#)

[Intrinsic Version](#)

[Black or White](#)

[STL Iterator](#)

[Hashing Multiple Bits](#)

[See also](#)

[Forum Posts](#)

[External Links](#)

[References](#)

[What links here?](#)

Isolating Subsets

The process of isolating subsets is performed by [intersection](#), for single populated subsets with the [two's complement](#).

Single Bits

This obvious loop approach is similar to [Brian Kernighan's way](#) to count the number of one-bits by consecutively isolating and clearing the LS1B:

```
while ( x ) {
    U64 ls1b = x & -x; // isolate LS1B
    ...
    x &= x-1; // reset LS1B
}
```

or with likely the same generated assembly:

```
if ( x ) do {
    U64 ls1b = x & -x; // isolate LS1B
    ...
} while ( x &= x-1 );
```

Of course we may also reset the LS1B by

```
x ^= ls1b; // reset LS1B
```

but todays processors like to gain more parallelism to calculate independent expressions.

Multiple Bits

Isolating none-empty subsets with possibly multiple one-bits can be applied by [divide and conquer](#) , that is to divide the bitboard or [quad word recursively](#) into smaller items, two [double words](#), consisting of two [words](#) and those again of two [bytes](#). Word and byte-wide isolated subsets may then act as an index of a pre-calculated lookup table to convert those subsets to adequate data-structures, most likely lists with up to sixteen or eight elements.

Converting Sets to Lists

❗ x It's time for us to say farewell... Regretfully, we've made the tough decision to close Wikispaces. Find out why, and what will happen next. [Wikispaces.com](#)

For most applications LS1B-isolation alone is not appropriate, but the conversion from the exponential bitboard centric world to the scalar square centric world, also called [bit-scanning](#).

Scanning Forward

```
if ( x ) do {
    int idx = bitScanForward(x); // square index from 0..63
    *list++ = foo(idx, ...);
} while (x &= x-1); // reset LS1B
```

Per definition bitScanForward reveals the index of LS1B.

Scanning Reverse

If - for some reason - we like to traverse the sets in reverse or unknown order anyway, we can not (or don't want to) rely on the independent LS1B reset.

```
if ( x ) do {
    int idx = bitScanReverse(x); // square index from 0..63
    *list++ = foo(idx, ...);
} while (x ^= powOf2[idx]); // or 1ULL << idx -> reset found bit
```

Scanning with Reset

A win of abstraction is to use a combined [bitscan with reset](#) found bit routine. This is fine. But probably harder for compilers to generate optimal code in the if-do-while-sense, where reset last bit already sets the zero-flag. If you don't care on such micro-optimizations, this is the preferred control structure.

```
while ( x ) {
    int idx = bitScanAndReset(&x); // square index from 0..63
    *list++ = foo(idx, ...);
}
```

One may even don't care about the order.

Intrinsic Version

If bitscan is able to properly handle empty sets - leaving an value outside the 0..63 range (like leading or trailing zero count), we may think about to skip the leading while condition and to break on bitscan(x) > 63 for instance. That was not recommend - since the reset leaves the condition en-passant, and the computational cost of an additional bitscan or zero count was higher. If you like to play the optimization game, it might be fine for [x86-64 Core 2 duo](#) thought - using [bitscan](#) and [bittestandreset](#) intrinsics or wrappers - if kept all in registers of course.

```
while (_BitScanForward64(&idx, x)) { // or reverse
    *list++ = foo(idx, ...);
    _bittestandreset64(&x, idx);
}
```

The loop is intended to look like this in [x86-64 assembly](#):

```
; input rdx - move target set
;   ecx - move from aspects
;   rdi - pointer to movelist
std     ; set direction flag
bsf rax, rdx ; scan first to-bit
jz over  ; jump if no more moves
loop:
    btr rdx, rax ; reset found bit
    or  eax, ecx ; combine to- with from-square
    stosw ; store 16-bit move *rdi++ = move
    bsf rax, rdx ; scan next to-bit
    jnz loop ; jump if more moves
over:
```

Black or White

With bitboard serialization one minor problem is the relative order in [move generation](#) considering [side to move](#). Bsf scans the board in a1..h1, a2..h2, a3..h3 order, assuming [little-endian rank-file mapping](#), which might be the desired order for an attacking black player. Traversing white pieces and target squares the same way say a few bit-arrays, and different search behavior of [color flipped positions](#). Despite other features considered in [move ordering](#), the initial order in generation has more or less influence. Therefore, it is desired to traverse the "white" bitboards with priority for the black back-rank as well [\[2\]](#). This might be done by bitscan reverse, which covers the rank symmetry, but also mirrors the files. Another alternative is to traverse a [vertically flipped](#) "white" bitboard, which can be done outside the do-while loop by a "conditional" [x86-64 byte swap](#), and requires one further register and xor per loop cycle, which might be combined with other stuff, f.i. the [from square](#) of a move:

```
if ( x ) {
    U64 m = (U64)color - 1; // e.g. -1 if white, 0 for black
    int o = (int)m & 56;
    x = x ^ ((x ^ flipVertical(x)) & m); // conditional flip
    do {
        int idx = bitScanForward(x) ^ o; // square index from 0..63
        *list++ = foo(idx , ...);
    } while (x &= x-1); // reset LSB
}
```

STL Iterator

[Rein Halbersma](#) has written a prototype of a [generic C++11](#) bitset [template](#) that can be used to traverse a set in [STL iterator](#) style, hiding bitscan and reset [\[3\]](#) [\[4\]](#) ...

```
typedef bit_set<int64_t, 1> bitset;

void testLoop(int* p, const bitset & x) {
    for (auto it = x.begin(); it != x.end(); ++it)
        *p++ = *it;
}
/* or std::copy */
void testCopy(int* p, const bitset & x) {
    std::copy(x.begin(), x.end(), p);
}
```

... which yields in following [X86-64 assembly](#), almost identical for the iterator loop and std::copy [\[5\]](#):

```
; testLoop(int*, bit_set<long, 1ul>):
    test    rsi, rsi
    jne     .L13
.L7:
    ret
.L13:
    bsf     rax, rsi
    mov     DWORD PTR [rdi], eax
    lea     rax, [rsi-1]
    add     rdi, 4
    and     rsi, rax
    jne     .L13
    jmp     .L7
```

Hashing Multiple Bits

Similar to the idea to hash occupancies in [kindergarten bitboards](#) and [magic bitboards](#), one may hash certain move-target subsets of one piece in one run, to lookup tables with pre-calculated moves-lists [\[6\]](#). Sounds like doing up to eight bitscans in parallel. The idea is to multiply-shift-lookup a move-target bitboard, to do an almost branch-less [move-generation](#) with pre-calculated moves inside [move-lists](#). For instance king- and knight-moves [\[7\]](#) as well as moves of sliding pieces per line:

```
moveTarget = KingAttacks[sq] & ~(ownPieces | attackedSquares);
idx = (moveTarget * kingMagic[sq]) >> kingShift[sq];
movelists = kingMoveLists[sq];
movelist = movelists[idx];
moveCpy(target, movelist, movelist->n);
```

A king in a corner may have up to three moves. Thus there are $2^3 == 8$ possible move-lists. For instance for a king on a1 (number of moves: vector of moves):

```
0:{empty}
1:{a1-b1}
1:{a1-a2}
1:{a1-b2}
```

2:{a1-b1,a1-a2}

2:{a1-b1,a1-b2}

2:{a1-a1,a1-b2}

3:{a1-b1,a1-a1,a1-b2}

It's time for us to say farewell... Regretfully, we've made the tough decision to close Wikispaces. Find out why, and what will happen, [here](http://blog.wikispaces.com) (http://blog.wikispaces.com)

Kings on edges have 5 potential target squares, thus there are 32 possible move-lists. All other kings have 8 all the 8 neighbors with up to 256 move-lists. Similar move-list enumeration is possible with knights and others. All possible move-target subsets of kings and knights for all 64 from-squares are [perfectly minimal hashtable](#) with a magic factor of four one-bits set. 10016 possible king-move-lists and 5520 knight-move-lists. To reduce memory one may offset the sets to a "normalized" source square per king, knight and sliding piece line, implying some vector arithmetic in the board centric world considering the offset.

The less populated move-target subsets are, the less efficient this hashing technique. This might become a problem since bitboard move-generation is essentially about subsets of moves with certain properties, like most importantly fast winning captures at [Cut-nodes](#).

See also

- [Move Generation](#)
- [Belle | Hardware Move Generation](#)
- [Move Ordering](#)
- [Pieces versus Directions](#)
- [Table-driven Move Generation](#)
- [Traversing Subsets of a Set](#)

Forum Posts

- [Subject: sliding move generation idea with bitboards](#) by [Gerd Isenberg](#), [CCC](#), February 19, 2006
- [Magic Knight- and King-Move Generation](#) by [Gerd Isenberg](#), [Winboard Forum](#), Januar 11, 2007
- [Re: C vs ASM](#) by [Rein Halbersma](#), [CCC](#), March 05, 2013
- [Symmetric move generation using bitboards](#) by [Lasse Hansen](#), [CCC](#), December 20, 2014 » [BitScan](#)

External Links

- [Serialization from Wikipedia](#)

References

- [^](#) [Picture gallery "Back in Holland 1941 - 1954"](#) from [The Official M.C. Escher Website](#)
- [^](#) [Re: Alternatives to History Heuristics](#) by [Robert Hyatt](#), [CCC](#), September 02, 2009
- [^](#) [LiveWorkSpace\(IDE online\): C++-3.2 \(clang++\)](#): [41EaZl](#)
- [^](#) [Re: C vs ASM](#) by [Rein Halbersma](#), [CCC](#), March 05, 2013
- [^](#) [GCC Explorer](#) with g++ 4.7 compiler, options -std=c++11 -O3 -march=k8-sse3 -fverbose-asm
- [^](#) [Subject: sliding move generation idea with bitboards](#) by [Gerd Isenberg](#), [CCC](#), February 19, 2006
- [^](#) [Magic Knight- and King-Move Generation](#) by [Gerd Isenberg](#), [Winboard Forum](#), Januar 11, 2007

What links here?


| Page | Date Edited |
|---|--------------|
| Backtracking | Dec 16, 2017 |
| Bison | Sep 29, 2016 |
| Bitboard Serialization | Dec 24, 2014 |
| Bitboards | Nov 14, 2017 |
| BitScan | Sep 10, 2017 |
| Blockers and Beyond | Mar 21, 2014 |
| Brainless | Jun 24, 2017 |
| Checks and Pinned Pieces (Bitboards) | Aug 14, 2013 |
| CHEOPS | Apr 18, 2015 |
| Color Flipping | May 17, 2017 |
| Crafty | Jan 28, 2018 |
| Double and Triple (Bitboards) | May 12, 2016 |
| Efficient Generation of Sliding Piece Attacks | Nov 5, 2016 |
| General Setwise Operations | Feb 25, 2018 |
| Gibbon | Dec 23, 2014 |
| Gk | Oct 9, 2017 |

[More Links](#)

| Page | Date Edited |
|--|--------------|
| Hash Table | Jan 1, 2018 |
| Iteration ✖ It's time for us to say farewell... Regretfully, we've made the tough decision to close Wikispaces. Find out why, and what will happen, here (http://blog.wikispaces.com) | May 5, 2017 |
| Joker iT | Sep 16, 2017 |
| Knight Pattern | Feb 23, 2015 |
| Lasse Hansen | Sep 1, 2015 |
| Linked List | Oct 11, 2016 |
| Move Generation | Jan 29, 2018 |
| NoraGrace | Nov 23, 2014 |
| Open Pawns (Bitboards) | Aug 24, 2017 |
| Passed Pawns (Bitboards) | Aug 23, 2010 |
| Pawn Pushes (Bitboards) | Apr 5, 2013 |
| Piece-Sets | Jun 9, 2017 |
| Pieces versus Directions | Oct 6, 2016 |
| Population Count | Sep 3, 2017 |
| Praetorian | Feb 21, 2015 |
| Rein Halbersma | Dec 22, 2015 |
| Reverse Bitboards | Aug 29, 2015 |
| Ryan Mack | Jan 18, 2011 |
| Rybka | Mar 27, 2017 |
| Sliding Piece Attacks | May 27, 2016 |
| Sliding Pieces | Aug 3, 2017 |
| Spector | Nov 11, 2016 |
| Tinker | Aug 29, 2015 |
| Traversing Subsets of a Set | Oct 14, 2016 |
| More Links | |

[Up one Level](#)

[Hilfe](#) · [Über](#) · [Preisliste](#) · [Privatsphäre](#) · [Bedingungen](#) · [Unterstützung](#) · [Höherstufen](#)

Contributions to <https://chessprogramming.wikispaces.com/> are licensed under a [Creative Commons Attribution Share-Alike 3.0 License](#). 

Portions not contributed by visitors are Copyright 2018 Tangient LLC

[TES: The largest network of teachers in the world](#)