Home * Board Representation * Bitboards * Sliding Piece Attacks * **Magic Bitboards**

Samuel Bak - Auspicious Moon, 2001 [3]

Magic bitboards, a multiply-right-shift perfect hashing algorithm to index an attack bitboard database - which leaves both line-attacks of bishop or rook in one run. Thanks to the fast 64-bit multiplication and fast and huge caches of recent processors, Magic Bitboards has become a de facto standard of modern bitboard engines, as used for instance in Crafty, Arasan, Stockfish and Houdini. While Robert Hyatt reported no immediate speed gain over Rotated Bitboards in Crafty [1] , Jon Dart mentioned a 20-25% speedup [2] in Arasan over rotated.

Table of Contents

History

The magic bitboard approach was motivated by Gerd Isenberg's multi-direction hashing technique kindergarten bitboards and probably by Gerd's and Tony van Roon-Werten's early trials to map line-wise occupancies by De Bruijn- or random number multiplication [4] . Lasse Hansen had the idea to hash the up

to twelve relevant occupied bits of **both directions** of a rook- or bishop movement simultaneously [5] .

Pradu Kannan's improvements to Lasse Hansen's initial approach was to introduce a Java-like, two-dimensional array with individual size for each square and all it's relevant occupancies [6] . Big savings in table-size - since many squares on either orthogonal or diagonal lines require less bits than others, especially considering the inner six bits. While center squares are more dense for rooks, it is the opposite for bishops [7] .

Recently, Robert Purves coined the names Plain and Fancy Magics [8] , and found Hansen's initial Plain Magics with 2 MiB table for rooks and 256 KiB for bishops nearly indistinguishable from Fancy (about 800 KiB and 38 KiB) on his Intel i5 with huge L3 smart cache, see plain versus fancy source code. In the same CCC forum thread, Robert Houdart proposed a byte lookup per square for further table reductions.

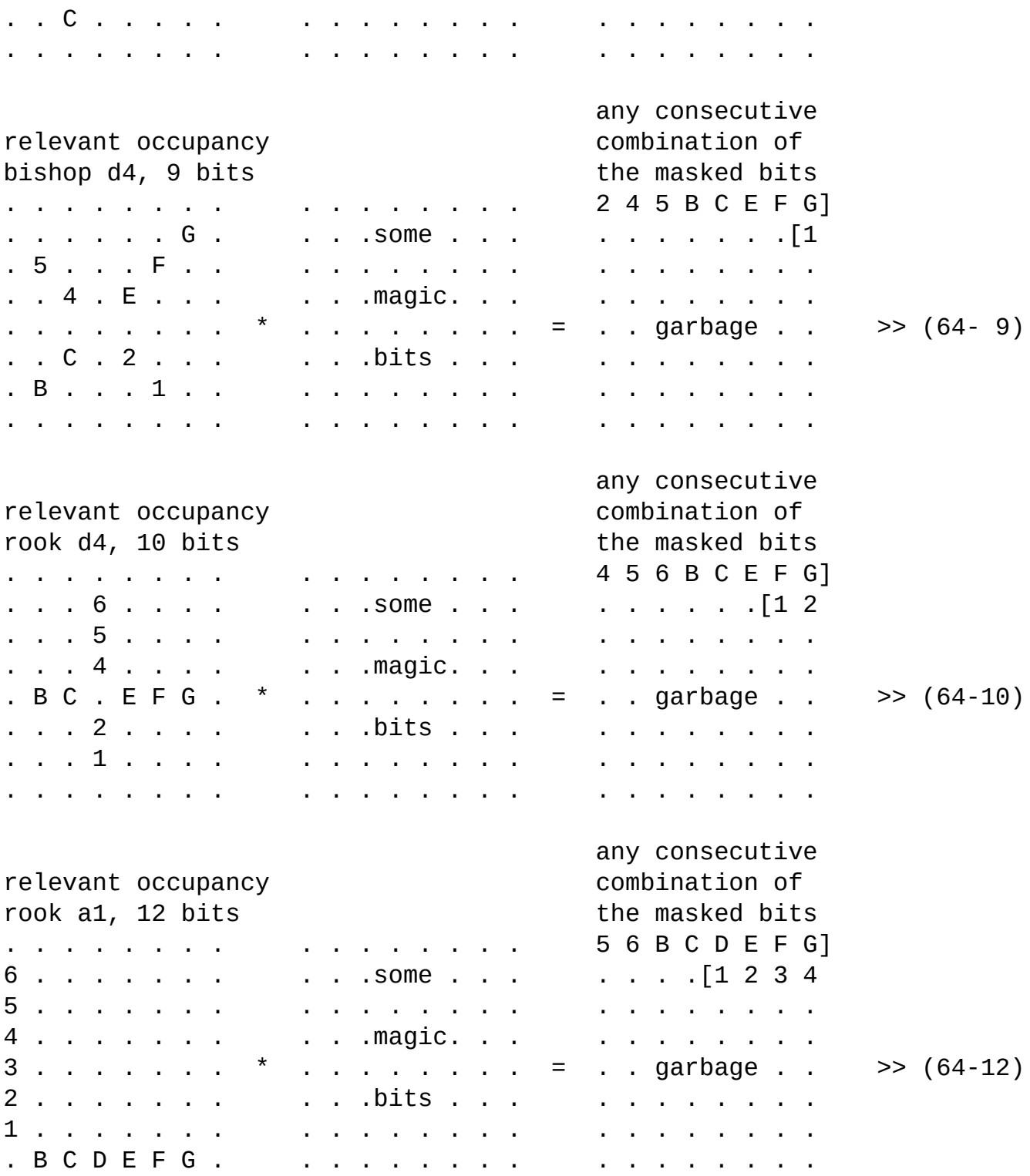
How it works

A magic move-bitboard generation technique consists of four key steps:

1. Mask the relevant occupancy bits to form a key. For example if you had a rook on a1, the relevant occupancy bits will be from a2-a7 and b1-g1.
2. Multiply the key by a "magic number" to obtain an index mapping. This magic number can be generated by brute-force trial and error quite easily although it isn't 100% certain that the magic number is the best possible (see step 3).
3. Right shift the index mapping by 64-n bits to create an index, where n is the number of bits in the index. A better magic number will have less bits required in the index.
4. Use the index to reference a preinitialized move database.

The following illustration should give an impression, how magic bitboards work. All masked relevant occupied bits are perfectly hashed to the consecutive occupied state to index the pre-calculated attack-sets. Constructive collisions, where different occupancies map same attack-sets - since different bits are outer redundant bits "behind" the first blocker, are desired and even necessary to apply a perfect hashing with N bits.

```
                                        any consecutive
relevant occupancy                      combination of
bishop b1, 5 bits                       the masked bits
. . . . . . . .     . . . . . . . .     . . .[C D E F G]
. . . . . . . .     . 1 . . . . . .     . . . . . . . .
. . . . . . G .     . 1 . . . . . .     . . . . . . . .
. . . . . F . .     . 1 . . . . . .     . . . . . . . .
. . . . E . . .  *  . 1 . . . . . .  =  . . garbage . .    >> (64- 5)
. . . D . . . .     . 1 . . . . . .     . . . . . . . .
```



The above illustration is correct for the b1 bishop, since it has only one ray and one bit per file and works kindergarten like. In general a one to one mapping of N scattered occupied bits to N consecutive bits is not always possible due to overflows. A perfect mapping of N scattered bits to N consecutive bits is likely not minimal for most squares. It requires one or two gaps inside the consecutive N bits, to avoid collisions, blowing up the table size.

But the purpose is to perfectly hash attack-sets rather than consecutive occupied bits. The number of

distinct attack-sets is much smaller than the relevant occupancies. Thus, with the help of constructive collisions, some initial guess how to map the bits, and/or trial and error, using exactly N bits is always possible. If one tries hard enough to maximize constructive collisions - even less. There are some N-1 ones reported at Best Magics so far, which will half the individual table size for some squares in the widespread Fancy Magic Bitboards approach.

Perfect Hashing

Magic bitboards applies perfect hashing. A surjective function, to map the vector of all relevant occupancies to a range of attack-sets per square. The less bits the attack-set - the closer the blockers, the more those attack-sets are shared by occupancies with different, but redundant outer squares.

- The **cardinality** of all **relevant occupancies** is determined by the number of bits to map, varying from five to twelve - thus, the cardinality is the power of two the number of bits, varying from 32 to 4096.
- The **cardinality** of **distinct attack-sets** is determined by the product of the length of each of the max four direction rays greater than zero (or one). The rook on d4 has 3*4*3*4 = 144 distinct attack-sets, a bishop on a8 has only 7.

The **ratio** of both cardinalities, that is all **relevant occupancies** versus the all **distinct attack-sets** is illustrated below: As a quarter of a board - due to the symmetry, the other squares may deduced by flipping and mirroring. Noticeable is the huge 4096/49 ratio of 2^12 occupied states versus 7 times 7 attack-sets of the edge rooks - 12 bits instead of 6. Those "expensive" squares make constructive collisions very desirable. To become more "minimal" by saving an index bit - to halve down the table for one square or the other.

	bishop on square				#occs/ #attset	rook on square			
	A	B	C	D		A	B	C	
8	64/7	32/6	32/10	32/ 12		8	4096/49	2048/42	2048/ 70
7	32/6	32/6	32/10	32/ 12		7	2048/42	1024/36	1024/ 60

6	32/10	32/10	128/40	128/ 48	6	2048/70	1024/60	1024/100
5	32/12	32/12	128/48	512/108	5	2048/84	1024/72	1024/120

	bishop on square				rook on square			
	A	B	C	D		A	B	C
8	9.14	5.33	3.20	2.67	8	83.59	48.76	29.26
7	5.33	5.33	3.20	2.67	7	48.76	28.44	17.07
6	3.20	3.20	3.20	2.67	6	29.26	17.07	10.24
5	2.67	2.67	2.67	4.74	5	24.38	14.22	8.53

The idea to implement minimal perfect hashing by an additional 16-bit indirection turned out to be slower (see conditional compiles in Pradu Kannan's source [9]).

Recent table sizes were about 38 KiB for the bishop attacks, but still about 800 KiB for rook attacks (Fancy). That sounds huge, considering L1 and L2 (L3) cache-sizes and number of cachelines and pages needed - we likely fetch distinct cachelines for each different square or occupancy. On the other hand caches and pages become larger in future processors. And occupancy and squares of the lookups don't change that randomly inside a search that we can still expect a lot of L1-hits. Unfortunately changes in

occupancy outside the blockers and therefor not affecting the attack-set will introduce some more cache misses.

Wishing Dreams

Since there are 1428/4900 **distinct** attack sets for the bishop/rook attacks, we can use 11(2048 > 1428)/13(8196 > 4900) index bits for all bishop/rook attack bitboards. The table size will be 16KB for the bishop attacks and 64KB for rook attacks. To make it possible, we must find a group of magics (one per square and dependent each other). All the magics in the group must hash its all **relevant occupancies** into the same 11/13 bit index without any collision with others (anyhow for each square, different occupancies can hash to the same attack sets).

```
U64 bishopAttacks[2048]; // 16KB
U64 rookAttacks[8196]; // 64KB

struct GMagic {
   U64 mask;
// to mask relevant squares of both lines (no outer squares) and inclu
de sliding piece itself
   U64 magic; // magic 64-bit factor
};

GMagic mBishopTbl[64]; //1KB
GMagic mRookTbl[64]; //1KB

U64 bishopAttacks(U64 occ, enumSquare sq) {
   occ &= mBishopTbl[sq].mask;
   occ *= mBishopTbl[sq].magic;
   occ >>= 64-11;  //fixed shift
   return bishopAttacks[occ]; //no offset
}

U64 rookAttacks(U64 occ, enumSquare sq) {
   occ &= mRookTbl[sq].mask;
   occ *= mRookTbl[sq].magic;
   occ >>= 64-13;
   return rookAttacks[occ];
}
```

But the idea seems like a wishing dream. Can we find ONE of the **magic group** of magics? Does it exist in theory? If not, can we use more bits to reach the condition? If yes, can we find a good magic group? A good magic group means that all the hash value