Magic Move-Bitboard Generation in Computer Chess

Pradyumna Kannan

April 30, 2007

Abstract

This ***casual*** paper is written to provide programmers of chess engines a discussion of one of the fastest and most versatile move-bitboard generators for individual sliding pieces. It is assumed that the reader has prior knowledge of the fundamentals of bitboards and the C programming language.

Contents

Listings

1 Acknowledgements

I would like to express my sincere gratitude to users of the Winboard Forum[1] for their constant support, guidance, and motivation. I am espically grateful to Lasse Hansen[2] and Gerd Isenberg[3] for providing interesting ideas and valuable feedback. I take this opportunity to also thank Andrew Fan and Sune Fischer for reviewing this paper.

2 Bitboard Representation and Orientation

In chess there are 64 squares on the board, therefore 64-bit numbers are required to be used as bitboards as shown in listing 1.

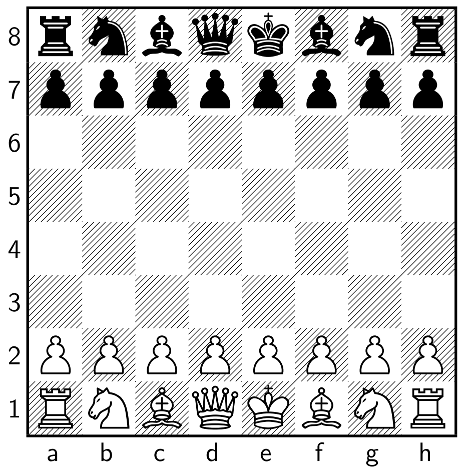
Listing 1: Definition of a 64 bit number in C

```
/*The following definition of an unsigned 64-bit number that will be used
 *throughout the rest of this paper assumes a C99 compiler conformance.
 *The current Microsoft, Intel, and GNU compilers will be able to compile
 *this code.
 */

typedef unsigned long long U64;

/*The following macro is used to append the appropriate suffix to unsigned
 *64-bit constants.
 */
#define C64(x) x##ULL
#define U64FULL C64(0xFFFFFFFFFFFFFFFF)
```

For this text our bitboard representation has A1 being the least significant bit[4], H8 being the most significant bit[5], and the squares in between represented by counting up through ranks. To show the representation graphically, here is a chessboard as we see it from white's side:



[1]http://www.vpittlik.org/wbforum/
[2]Inventer of multi-direction magic move-bitboard generation
[3]Inventer of seperated-direction magic move-bitboard generation
[4]For the rest of this text LSB will be used as an acronym for least significant bit.
[5]For the rest of this text MSB will be used as an acronym for most significant bit.

Here are the respective indices for each bit in the bitboard for the above chessboard[6].

$$\begin{bmatrix} 56 & 57 & 58 & 59 & 60 & 61 & 62 & 63 \\ 48 & 49 & 50 & 51 & 52 & 53 & 54 & 55 \\ 40 & 41 & 42 & 43 & 44 & 45 & 46 & 47 \\ 32 & 33 & 34 & 35 & 36 & 37 & 38 & 39 \\ 24 & 25 & 26 & 27 & 28 & 29 & 30 & 31 \\ 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 \\ 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{bmatrix}$$

In binary the same chessboard can be represented like this[7]:

$$H8G8F8E8D8C8B8A8H7G7F7E7D7C7B7A7\ldots H2G2F2E2D2C2B2A2H1G1F1E1D1C1B1A1$$

Although this definition of the bitboard will be used throughout the rest of the text, any particular orientation and geometry of the bitboard may be hashed for move-bitboard generation by magic hashing techniques.

3 The Magic Hashing Function

3.1 Introduction

Initially, the magic hashing function is best seen in an abstract manner. If one has a sparsely populated key a and needs to hash it to an index in the database, then the hashing scheme in listing 2 can be used.

Listing 2: The 64-bit Magic Hashing Function

```
/*a - a sparsely populated input key
 *b - a magic constant used to hash the key a
 *s - the number of bits in the index
 */
index = (a*b)>>(64-s);
info = database[index];
```

The bits in a will get shifted up by multiplication according to a pattern in the magic key b. The result[8] will be shifted down to create an index. We will talk more about the mechanics of this hashing function later and will now move on to the magic bitscan as a concrete and simple example of the usage of this magic hashing scheme.

3.2 The Magic Bitscan

A bitscan will get the distance of a single bit from either the MSB or the LSB. We will concentrate on an LSB bitscan because there are some tricks to isolate the LSB. An implementation of a 64-bit LSB bitscan is shown in listing 3. This idea of a magic bitscan was apparently first introduced in July 1998 by Charles E. Lieserson, Harald Prokop, and Keith H. Randall from the MIT Laboratory for Computer Science[9].

Right now the code and the magic key in listing 3 might seem all random and working by "magic", but after we run through an example of an 8-bit bitscan, understanding and writing this 64-bit bitscan will be easier. Let us first begin by simplifying the multiplication a little. We can note that if we have only a single

[6]Bitboards from now on will be represented as matrices like the one shown on this page.
[7]Note that the LSB is to the right and the MSB is to the left and that each square represents a 0 or a 1
[8]from now on the product will be referred to as an index mapping
[9]http://supertech.csail.mit.edu/papers/debruijn.pdf

Listing 3: 64-bit Magic Bitscan

```
#define BITSCAN_MAGIC C64(0x07EDD5E59A4E28C2)
/*X&-X gives the least significant bit because of two's complement encoding
 *X&(~X+1) is equivalent to X&-X
 *The bit is then run through the magic hashing scheme to index a database
 */
#define FirstOne(X) BitScanDatabase[(((X)&-(X))*BITSCAN_MAGIC)>>58]
int BitScanDatabase[64];
/*The initialization function runs all possible inputs through the
 *hashing scheme and places the correct data in the database.
 */
void initializeFistOne()
{
    U64 bit=1;
    int i=0;
    do
    {
        BitScanDatabase[(bit*BITSCAN_MAGIC)>>58]=i;
        i++;
        bit<<=1;
    }while(bit);
}
```

isolated bit then it is a power of 2. We can also note that $b*2^n$ is equivalent to $b << n$. Here are examples[10] to make it clear that $b*2^n$ is equivalent to $b << n$:

$$010110 * 10 = 101100 = 010110 << 1$$

$$001101 * 100 = 110100 = 001101 << 2$$

It can be seen that the multiplication left-shifts the magic b by an amount depending on the input key. The next step is to note that the right-shift in the magic hashing function will make the upper bits of the product the index. We will now present a graphical display of the newly introduced concepts.

```
magic = abcdefgh (each letter corresponds to a bit)
```

The indices are computed by running all possible inputs through the magic hashing function. Note that for this 8-bit bitscan our index will be 3-bits wide because there will be $8 = 2^3$ elements in our database.

```
When applying the magic hashing scheme (key*magic)>>5
for key = 00000001 the index will become abc
for key = 00000010 the index will become bcd
for key = 00000100 the index will become cde
for key = 00001000 the index will become def
for key = 00010000 the index will become efg
for key = 00100000 the index will become fgh
for key = 01000000 the index will become gh0
for key = 10000000 the index will become h00
```

[10]The numbers except the shifts are in binary

To generate the magic, we have to make sure that the index will be unique for every input key. A trial-error approach works well to generate the magic. First try zeros in our magic then try ones if any indices repeat. In the following trial-error run, indices are shown graphically as parts of the magic.

```
  |abcdefgh|
 1|       0|00 //Putting in a zero for h
 2|      00|0  //Putting in a zero for g - collision with 1
 3|      10|0  //Putting in a one  for g
 4|     010|   //Putting in a zero for f
 5|    001 |   //Putting in a zero for e - all future variations collide
 6|   000  |   //Putting in a zero for d - collision with 1
 7|   100  |   //Putting in a one  for d - collision with 3
 8|    101 |   //Putting in a one  for e
 9|   010  |   //Putting in a zero for d - collision with 4
10|   110  |   //Putting in a one  for d
11|  011   |   //Putting in a zero for c - all future variations collide
12| 001    |   //Putting in a zero for b - all future variations collide
13|000     |   //Putting in a zero for a - collision with 1
14|100     |   //Putting in a one  for a - collision with 3
15| 101    |   //Putting in a one  for b - collision with 8
16|  111   |   //Putting in a one  for c
17| 011    |   //Putting in a zero for b
18|001     |   //Putting in a zero for a
```

All the unique indices can be put together to make our magic.

```
 1|       0|00
 3|      10|0
 4|     010|
 8|    101 |
10|   110  |
16|  111   |
17| 011    |
18|001     |
  |00111010| = 0x3A
```

The magic can now be used in an 8-bit bitscan as shown in listing 4.

Listing 4: 8-bit Magic Bitscan

```
#define BITSCAN_MAGIC 0x3A
#define FirstOne(X) BitScanDatabase[(((X)&-(X))*BITSCAN_MAGIC)>>5]
char BitScanDatabase[8];
void initializeFistOne()
{
   unsigned char bit=1;
   char i=0;
   do
   {
      BitScanDatabase[(bit*BITSCAN_MAGIC)>>5]=i;
      i++;
      bit<<=1;
   }while(bit);
}
```

Although it would be rather tedious by hand[11], the same technique of generation used to generate magics for the 8-bit bitscan can be applied to the 64-bit bitscan. Hopefully this section on the magic bitscan gave a solid foundation for the use of the magic hashing function. The magic bitscan is also referenced when magic generation techniques for move-bitboard generation are discussed.

3.3 Source Code for Move-Bitboard Generation

Here C source code is developed for magic move-bitboard generation but we will leave a discussion of generating magics for a later section. To generate the move-bitboard for a particular piece we will require the relevant occupancy information from the occupancy bitboard[12]. When isolating the relevant occupancy bits we can make the simplification that the occupancy at the board edges do not affect the moves of a sliding piece. For example, for a slider on D4 we will only need to consider the following occupancy bits[13] in our key for the magic hashing function:

$$BishopMask[D4] = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & X & \cdot \\ \cdot & X & \cdot & \cdot & \cdot & X & \cdot & \cdot \\ \cdot & \cdot & X & \cdot & X & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & X & \cdot & X & \cdot & \cdot & \cdot \\ \cdot & X & \cdot & \cdot & \cdot & X & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix}; RookMask[D4] = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & X & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & X & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & X & \cdot & \cdot & \cdot & \cdot \\ \cdot & X & X & \cdot & X & X & X & \cdot \\ \cdot & \cdot & \cdot & X & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & X & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

To isolate the relevant occupancy bits, we will & the mask of the relevant bits with the occupancy bitboard. The next step is to run the relevant occupancy bits through the magic hashing scheme to index a move-bitboard database. The generic procedure for both rooks and bishops is shown in listing 5.

Listing 5: Homogeneous Array Access Magic Move-Bitboard Generator

```
/*Initialize the move database in the same manner that we initialized the
 *magic bitscan. Run through all possible inputs and use the pre-computed
 *magic to find the index. Compute the move-bitboard by any conventional
 *method and place it at the indexed location in the move database.
 */
U64 moveDB[64][1<<s]; //s is the number of bits in the index and depends
                      //on the quality of the magics being used
//This mask is used to get the relevant occupancy bits
U64 mask[64];
U64 magics[64];

inline U64 move(const unsigned int square, const U64 occupancy)
{
  return moveDB[square][((occupancy&mask[square])*magic[square])>>(64-s)];
}
```

3.3.1 Platform-Dependant Optimizations

It is possible to make memory optimizations by noting that different squares on the board have a different number of bits in the index. We can then reduce the size of the database as shown in listing 6. This can

[11]That's why we have computers!
[12]An occupancy bitboard has active bits on squares where pieces are placed and has inactive bits everywhere else
[13]X signifies the active bits

sometimes improve speed on platforms that suffer cache problems from the larger homogeneous access array databases.

Listing 6: Reduced Array Access Magic Move-Bitboard Generator

```
U64 moveDB[DB_SIZE];
//DBindex points to the beginning of moves for a square in moveDB
U64* DBindex[64];
/*Since the bits in the indices are different for each square
 *we will require different shifts for each square in the magic
 *hashing function.
 */
U64 shift[64];
U64 mask[64];
U64 magics[64];

inline U64 move(const unsigned int square, const U64 occupancy)
{
  return *(
           DBindex[square]+
           (((occupancy&mask[square])*magic[square])>>shift[square])
           );
}
```

Optimizations for 32-bit platforms can be made by splitting up the 64-bit multiplication into two 32-bit multiplications and a combination as shown in listing 7.

Listing 7: 32-Bit Optimized 64-bit Magic Hashing Function

```
/*All 64-bit values are considered unions of type U64 and a structure
 *of two 32-bit unsigned integers--hi and lo. Note that the magic
 *used for the 32-bit optimized 64-bit magic hashing function will be
 *different from the magic used for the original 64-bit hashing function.
 *a  - a sparsely populated key
 *b  - the magic used to hash the key a
 *s  - the number of bits in the index
 */
index = (a.u32.hi*b.u32.hi + a.u32.lo*b.u32.lo)>>(32-s);
info = database[index];
```

If available memory becomes a problem[14] one can split up the move-bitboard generation by only generating a part of the move bitboard at a time. This can be done by generating the move-bitboard for each direction of movement separately.

4 Generation of Optimal Magics

This section will introduce a generalized approach for generating optimal magics[15]. It is highly recommended that the previously developed method of generating magics for the magic bitscan is well understood.[16].

[14]like for cheap hand-held platforms

[15]An optimal magic is one that requires the smallest possible number of bits in the index

[16]One can apply this generalized magic generation technique to the bitscan to help visualize how it works

4.1 Step One - Assume variable bits in the magic b

In this paper b_i will represent a bit i bits away from b's LSB. Here are some graphical representations of the variable magic b:

$$\begin{bmatrix} b_{56} & b_{57} & b_{58} & b_{59} & b_{60} & b_{61} & b_{62} & b_{63} \\ b_{48} & b_{49} & b_{50} & b_{51} & b_{52} & b_{53} & b_{54} & b_{55} \\ b_{40} & b_{41} & b_{42} & b_{43} & b_{44} & b_{45} & b_{46} & b_{47} \\ b_{32} & b_{33} & b_{34} & b_{35} & b_{36} & b_{37} & b_{38} & b_{39} \\ b_{24} & b_{25} & b_{26} & b_{27} & b_{28} & b_{29} & b_{30} & b_{31} \\ b_{16} & b_{17} & b_{18} & b_{19} & b_{20} & b_{21} & b_{22} & b_{23} \\ b_8 & b_9 & b_{10} & b_{11} & b_{12} & b_{13} & b_{14} & b_{15} \\ b_0 & b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7 \end{bmatrix}$$

$$b_{63}b_{62}b_{61}b_{60}b_{59}b_{58}b_{57}b_{56}b_{55}b_{54}b_{53}b_{52}b_{51}b_{50}b_{49}b_{48}\ldots b_{15}b_{14}b_{13}b_{12}b_{11}b_{10}b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0$$

4.2 Step Two - Compute index mappings in terms of the variable magic

Multiply all possible inputs by the variable magic to compute all the index mappings[17] for each input. Let us go through an example of computing an index mapping for a particular input of a rook on D4.

$$Key = a = \begin{bmatrix} 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \end{bmatrix}$$

$$IndexMapping = Key * Magic = a * b$$

$$= \begin{bmatrix} 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \end{bmatrix} * \begin{bmatrix} b_{56} & b_{57} & b_{58} & b_{59} & b_{60} & b_{61} & b_{62} & b_{63} \\ b_{48} & b_{49} & b_{50} & b_{51} & b_{52} & b_{53} & b_{54} & b_{55} \\ b_{40} & b_{41} & b_{42} & b_{43} & b_{44} & b_{45} & b_{46} & b_{47} \\ b_{32} & b_{33} & b_{34} & b_{35} & b_{36} & b_{37} & b_{38} & b_{39} \\ b_{24} & b_{25} & b_{26} & b_{27} & b_{28} & b_{29} & b_{30} & b_{31} \\ b_{16} & b_{17} & b_{18} & b_{19} & b_{20} & b_{21} & b_{22} & b_{23} \\ b_8 & b_9 & b_{10} & b_{11} & b_{12} & b_{13} & b_{14} & b_{15} \\ b_0 & b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7 \end{bmatrix}$$

We will now simplify the multiply as we did for the magic bitscan. From ring theory[18] we can identify that the distributive, associative, and commutative properties apply to multiplication with overflow and addition with overflow. We can then split up the key into powers of two and apply the identity that $b * 2^n = b << n$.

$$a = \begin{bmatrix} 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \end{bmatrix} = 2^{11} + 2^{25} + 2^{30}$$

[17]Recall that an index mapping was defined as the product of the key multiplied by the magic
[18]http://mathworld.wolfram.com/Ring.html

$$\begin{aligned} IndexMapping &= a * b \\ &= b * a \\ &= b * (2^{11} + 2^{25} + 2^{30}) \\ &= b * 2^{11} + b * 2^{25} + b * 2^{30} \\ &= b << 11 + b << 25 + b << 30 \end{aligned}$$

To generalize this, if we have a key

$$a = \sum_{i=0}^{n} 2^{p_i},$$

where p_i is an integer and $0 \le p_i < p_{i+1} < 64$, then

$$IndexMapping = b * a = \sum_{i=0}^{n} b << p_i.$$

Likewise, for the 32-bit optimized 64-bit hashing function

$$IndexMapping = b^1 * a^1 + b^2 * a^2 = \sum_{i=0}^{n^1} b^1 << p_i^1 + \sum_{i=0}^{n^2} b^2 << p_i^2.$$

Back to our example,

$$IndexMapping = b << 11 + b << 25 + b << 30$$

$$= \begin{bmatrix} b_{45} & b_{46} & b_{47} & b_{48} & b_{49} & b_{50} & b_{51} & b_{52} \\ b_{37} & b_{38} & b_{39} & b_{40} & b_{41} & b_{42} & b_{43} & b_{44} \\ b_{29} & b_{30} & b_{31} & b_{32} & b_{33} & b_{34} & b_{35} & b_{36} \\ b_{21} & b_{22} & b_{23} & b_{24} & b_{25} & b_{26} & b_{27} & b_{28} \\ b_{13} & b_{14} & b_{15} & b_{16} & b_{17} & b_{18} & b_{19} & b_{20} \\ b_5 & b_6 & b_7 & b_8 & b_9 & b_{10} & b_{11} & b_{12} \\ 0 & 0 & 0 & b_0 & b_1 & b_2 & b_3 & b_4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} b_{31} & b_{32} & b_{33} & b_{34} & b_{35} & b_{36} & b_{37} & b_{38} \\ b_{23} & b_{24} & b_{25} & b_{26} & b_{27} & b_{28} & b_{29} & b_{30} \\ b_{15} & b_{16} & b_{17} & b_{18} & b_{19} & b_{20} & b_{21} & b_{22} \\ b_7 & b_8 & b_9 & b_{10} & b_{11} & b_{12} & b_{13} & b_{14} \\ 0 & b_0 & b_1 & b_2 & b_3 & b_4 & b_5 & b_6 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} +$$

$$\begin{bmatrix} b_{26} & b_{27} & b_{28} & b_{29} & b_{30} & b_{31} & b_{32} & b_{33} \\ b_{18} & b_{19} & b_{20} & b_{21} & b_{22} & b_{23} & b_{24} & b_{25} \\ b_{10} & b_{11} & b_{12} & b_{13} & b_{14} & b_{15} & b_{16} & b_{17} \\ b_2 & b_3 & b_4 & b_5 & b_6 & b_7 & b_8 & b_9 \\ 0 & 0 & 0 & 0 & 0 & 0 & b_0 & b_1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Note that the index cannot be computed immediately as we did for the magic-bitscan because there are carry effects on the index from addition. Instead, we will have to wait for the trial-error step to be able to compute the indices for some of the inputs. If $IndexMapping = b << p_0$ then the index can be computed immediately as was done for the magic bitscan. Note that IndexMappings don't have to be pre-computed and can be computed on the fly.

4.3 Step Three - Trial and error

Although there are some additional complications, we will use the same trial-error technique used for the magic bitscan. In other words we will guess bits in the magic while checking for unwanted collisions.

4.3.1 Grouping indices

Unlike the magic bitscan, multiple inputs can hash to the same index for move-bitboard generation. Identify which inputs and indices can map to the same index and organize them into collision groups.

4.3.2 Determining the order of guessing

For the magic bitscan we guessed the LSB first then guessed up to the MSB. If we went the other way[19], then the search tree would be much larger. Therefore picking the correct guessing order of b_i is *essential*. We can use a simple non-optimal ordering scheme for move-bitboard generation. b_i that occur in the index of hashed keys that are powers of two should be ordered first. Then, b_i related with the higheset sharing of b_i across index mappings should be ordered next[20]. Order b_i from LSB to MSB for b_i having direct influence on the index bits and order b_i from MSB to LSB for those having only carry influence on the index bits.

4.3.3 Collision detection

If we can compute the index for an input key, we can check if the index collides with any other computed indices that are not in the same collision group. If the index of a particular index mapping cannot be computed, we can assume it doesn't collide with any other indices. Listing 8 presents a function that will determine whether an index can be computed for the 64-bit magic hashing function.

Listing 8: Identification of Computable 64-bit Indices

```
typedef unsigned char bool;
#define true  1
#define false 0

/* This function makes sure that there are no carry effects on the index
 *from the unknown magic bits. This is done by subtracting the lowest
 *possible effect on the index (1<<(64-s)) by mappings that can possibly
 *help cause a collision.
 *a       - input key
 *known   - known active bits in the variable magic
 *unknown - here, unknown bits in the variable magic are active
 *s       - number of bits in the index
 *returns true if the index can be computed by the known magic bits
 */
bool computable(const U64 a, const U64 known, U64 unknown, const int s)
{
  U64 max = (C64(1)<<(64-s)) - (a*known & (U64FULL>>s));
  while(unknown)
  {
    U64 b_i = (unknown&-unknown);
    U64 map = a*b_i;
    if(map >= max) return false;
    max-=map;
    unknown^=b_i;
  }
  return true;
}
```

The magic is computed when all b_i are guessed without bad collisions.

[19]Start with a instead of h

[20]For rooks, the b_i related with the horizontal occupancy information should be ordered first

A Optimal Magics for 64-Bit Magic Move-Bitboard Generation

The following tables will present a selection of optimal magics for the 64-bit magic hashing function. The first column gives the square in question, the second column gives the magic, and the third column gives the smallest number of bits in the index that the magic is valid for.

Rooks		
Square	Magic (Hexadecimal)	Bits
A1	0x1234567890123456	10
A2	0x1234123546243543	9

Bishops		
Square	Magic (Hexadecimal)	Bits
A1	0x1234567890123456	10
A2	0x1234123546243543	9