Bitboards Explained

Tom Sandmann

17/12/2016

Abstract

When writing a chess engine, we need to have a representation of the chess board within the chess program. One of the most used representations is the so called bitboard representation. Bitboard are also used by most of the best engines in the world, including Stockfish, Komodo and Houdini. In this document, we will try to explain the basics behind the bitboard representation. The urge for writing this document started when we tried to understand bitboards ourselves. During our search, the lack of any satisfactory explanation eventually made us write such an explanation ourselves.

Contents

1 Representation

1.1 Coordinate System

The pieces on a chess board are as follows: Rook, Knight, Bishop, Queen and King. When naming the pieces and their types, we usually identify them by the first letter in the name of the piece. We often use the letter **K** for king, **Q** for queen, **B** for bishop and **N** for knight (because K is already used). We also differentiate between black and white pieces. White pieces are often represented with capital letters (KQBNRP), while black pieces are represented with small letters (kqbnrp). A row of a chess board is called a “rank” and a column is called a “file”. Ranks are denoted with the numbers *1* to *8*. Files are denoted with the letters *a* to *h*. The chessboard is placed with a light square at the right-hand end of

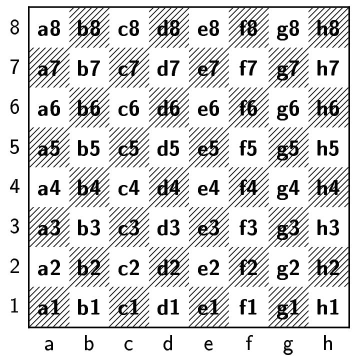


Figure 1: Algebraic chess notation for a chess board.

the rank nearest to each player. When talking about a specific square on the board, we use algebraic chess notation (see Figure 1). In Figure 2, we see the board setup when a new chess game is started. The common notation to label

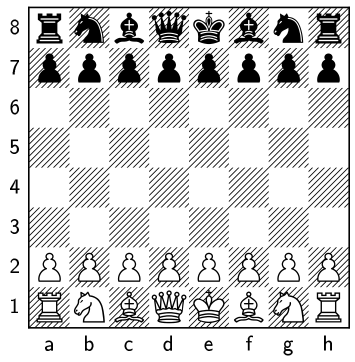


Figure 2: The initial position of a chess board.

squares on the chess board is to write the file-coordinate at the big-end and the rank-coordinate at the little-end, e.g. `a3` or `f7`. For example, if we want to describe the position of the white king, as seen in Figure 2, with the use of algebraic chess notation, we would say that the king is on square `e1`.

1.2 64 Bit Number

As you have probably figured out by yourself, a chess board contains 64 squares. Lets dive a bit into the deeper parts of the representation of numbers in computers. Computers represent numbers by strings of bits. Because a chess board contains 64 squares, we can actually represent it with a 64 bit number. An example of such a 64-bit number is the following:

0000000000000000000000000000000000000000000000000000000000000000

As you can see, both ends of the 64-bit number are colored differently. The green colored digit is the 0th bit. We also call this the **Least Significant Bit** (abbreviated to LSB). The red colored is the 63rd bit. We also call this the Most Significant Bit (abbreviated to MSB). I should put a little note on this.

1.2.1 Endianness

Endianness refers to the order of the bytes (a byte consists of 8 bits) in computer memory. These words may be represented in **big-endian** (big end first) or **little-endian** (little end first):

- When storing a word in **big-endian** format, the most significant byte, which is the byte containing the most significant bit, is stored first and the following bytes are stored in decreasing significance order with the least significant byte, which is the byte containing the least significant bit, thus being stored at last place.
- Little-endian format reverses the order of the sequence and stores the least significant byte at the first location with the most significant byte being stored last.

In this document, we **assume** that our 64-bit number is stored in **little-endian** format, where the least significant bit is stored first. Although it does not matter which representation you choose, we can read later on that this representation is more common.

Binary numbers can easily store positive numbers. But how are negative numbers stored? To be able to do this, we distinguish signed and unsigned integers. A signed integer can represent both positive and negative values, and unsigned integers can only represent non-negative numbers (zero or positive numbers). A common and simple way of storing negative numbers is to use most significant bit as a flag. This flag indicates the sign of the stored number. This way of representing positive and negative numbers is called two's complement. Because it is common to write a chess engine in C++, we have to make sure that we use a 64 bit **unsigned** int. By using an unsigned int, we guarantee that the MSB in our representation of the chessboard (which still is a number) will not be used as a sign for the stored number.

Now we got that cleared up, lets move on. The 64-bit number representing our chess board is what we call a **bitboard**. Bitboards (we can read later on that a chess engines uses at least 12 of them) will form the basis of our chess engine.

1.3 Logic Bitwise Operations

When using multiple or individual bits, we can perform several operations on them. These operations are called logic bitwise operations. Below we will discuss the most used ones:

- **NOT**. The NOT operator will flip the bit. If the bit was first 0, it will become 1 and vice versa.
- **AND**. The AND takes two bits, and outputs 1 if both bits are 1, and 0 otherwise.
- **OR**. The OR also takes two bits, and outputs 1 if at least one bit is 1, and 0 otherwise.
- **XOR**. The XOR operation also takes two bits, and outputs 1 if both bits are different, and 0 otherwise.

We can also write this in a table as can be seen in Table 1.

A	**B**	**NOT A**	**NOT B**	**A AND B**	**A OR B**	**A XOR B**
0	0	1	1	0	0	0
0	1	1	0	0	1	1
1	0	0	1	0	1	1
1	1	0	0	1	1	0

Table 1: Logical Bitwise Operations

Note that when we perform such logical bitwise operations on more than one bit, the operator will be applied to each consecutive pair in both the bit strings. Besides these basic operations, we also have two other operations that are not boolean logic. These actions can be performed with any bit number. The **LSHFT** and **RSHFT** operators respectively perform left and right shifts. In C++, the corresponding operation would be << for LSHFT and >> for **RSHFT**. Shifting means that you take the (in this case) 64-bit number and shift it left or right by the specified number of bits. If a bit 'falls of the edge', it is gone forever. If there are not enough bits to produce a 64-bit number, 'zero' bits (`0`) are added to always keep the total number of bits at 64. A **LSHFT** shifts the bits from the LSB to the MSB A **RSHFT** shifts the bits from the MSB to the LSB. These properties for both shifting directions **always** hold, regardless of the endianness being used (also discussed later on). An example:

- Suppose we have this 64-bit number:

```
1111111100000000000000000000000000000000000000000000000011111111
```

- We perform a LSHFT 4. This results in the following 64-bit number:

```
0000111111110000000000000000000000000000000000000000000000001111
```

- Performing a RSHFT 4, we get the following 64-bit number (note that it is not that same as our initial 64-bit number):

```
1111111100000000000000000000000000000000000000000000000011110000
```

As you can see, if we perform a LSHFT, padding zero's are added on the LSB side. The opposite occurs when performing a RSHFT, where padding zero's are added on the MSB side. We will always show the padding zero's, because we want our number to contain exactly 64 bits. Note that these padding zero's do not influence the number belonging to the binary representation.

1.4 FirstBit and LastBit

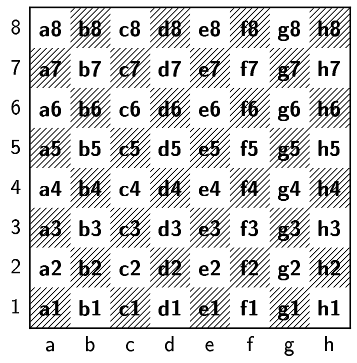
The function **FirstBit** give the index of the first bit that is turned on from the LSB side. For example, if we have the following bit string:

```
0000000000000000001000100000100001010110000000000100100000010000
```

FirstBit with the bit string given above will give us **18**, as the index of the bit that is turned on from the LSB is equal to 18. Alternatively, **LastBit** with this bit string will give us **59**, as this is the first bit turned on from the MSB.

1.5 Chess Board Mapping

Now we know the basic operations on one or more bits, we need to decide how to map each chess board square to a bit in the 64-bit integer. Although we could choose this mapping at random, it is better to pick a mathematically advantageous mapping. Remember the algebraic chess notation. We choose the mapping where the LSB corresponds to `a1` and the MSB corresponds to `h8`. The mapping together with the original chess board can be seen in Figure 3.



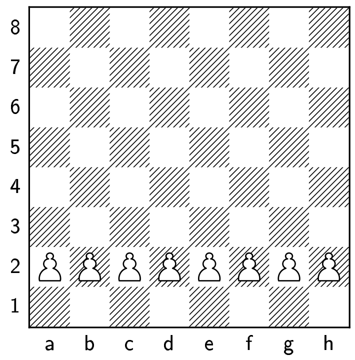
(a) Our chess board.

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

(b) Our mapping.

Figure 3: Our original chess board with our chosen mapping.

Lets take a look at our first real example, where we want to represent all of the white pawns from the initial position.



(a) Our chess board.

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	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0

(b) Bitboard representing intial white pawns.

Figure 4: The bitboard representing all of the initial white pawns.

The bitboard for all the initial white pawns is as follows:

```
0000000011111111000000000000000000000000000000000000000000000000
```

Lets recall some information regarding the chess board square identification and the mapping. By using a specific mapping, we explicitly assign a direct bit index to each position of the board. So:

$$\begin{array}{rr} A1 = 0 & G7 = 54 \\ B1 = 1 & H7 = 55 \\ C1 = 2 & A8 = 56 \\ D1 = 3 & B8 = 57 \\ E1 = 4 & C8 = 58 \\ F1 = 5 & D8 = 59 \\ G1 = 6 & E8 = 60 \\ H1 = 7 & F8 = 61 \\ A2 = 8 & G8 = 62 \\ B2 = 9 & H8 = 63 \\ \ldots & \end{array}$$

But we also explicitly assign a bit index to each position on the board:

$$\begin{array}{r} 0 = A1 \\ 1 = B1 \\ \ldots \\ 62 = G8 \\ 63 = H8 \end{array}$$

In general we call such a mapping a **square mapping**. We can create a square mapping based on couple of considerations:

- **Little-Endian**
  - **Files**: File index 0 maps the A-File, index 7 the H-File.
  - **Ranks**: Rank index 0 maps the first Rank, index 7 the eight Rank.
- **Big-Endian**

– **Files**: File index 0 maps the H-File, index 7 the A-File.
– **Ranks:** Rank index 0 maps the eight Rank, index 7 the first Rank.

Although you probably noticed that our mapping is a little-endian square mapping, we can easily deduct this by ourselves by enumerating files and ranks from 0 to 7 each. There are two common approaches to calculate the square-index from file or rank: Least Significant File Mapping (LSFM) or Least Significant Rank Mapping (LSRM). For each of these approaches, we can calculate the square index (as indicated by our mapping) as follows:

```
LSF squareIndex = 8*rankIndex + fileIndex
LSR squareIndex = 8*fileIndex + rankIndex
```

More common is the LSF-mapping where ranks are aligned to the eight consecutive bytes of a bitboard. Lets try to verify which mapping we just considered. For square *e3*, we know the square index should be equal to 20:

- LSF squareIndex = $8 \cdot \text{rankIndex} + \text{fileIndex} = 8 \cdot 2 + 4 = 20$.

As we can see in the calculation, we used a rank index of 2 and a file index of 4. Because the equality holds, we can conclude that we use a little-endian rank-mapping. In general, we most often use little-endian file-mapping with the A-File addressed by index zero and the H-file addressed by index seven. In C++, an enumeration of this rank-mapping might look like this:

```
enum enumRank {
er1stRank = 0,
er2ndRank = 1,
er3rdRank = 2,
er4thRank = 3,
er5thRank = 4,
er6thRank = 5,
er7thRank = 6,
er8thRank = 7,
};
```

The Little-Endian Rank-File (LERF) mapping implies the following C++ enumeration:

```
enum enumSquare {
a1, b1, c1, d1, e1, f1, g1, h1,
a2, b2, c2, d2, e2, f2, g2, h2,
a3, b3, c3, d3, e3, f3, g3, h3,
a4, b4, c4, d4, e4, f4, g4, h4,
a5, b5, c5, d5, e5, f5, g5, h5,
a6, b6, c6, d6, e6, f6, g6, h6,
a7, b7, c7, d7, e7, f7, g7, h7,
a8, b8, c8, d8, e8, f8, g8, h8
};
```

Below are some bitboards constants that apply to the LERF mapping. You can convert each of the hexadecimals to binary, probably converting the hexadecimal to a decimal first. When you convert the hexadecimal to binary, remember that the bit on the left will be the LSB bit, as we decided to use the little-endian format. When you have eventually converted the hexadecimal to binary, format the binary number as our mapping (Figure 3b) states.

```
a-file             0x0101010101010101
h-file             0x8080808080808080
1st rank           0x00000000000000FF
8th rank           0xFF00000000000000
a1-h8 diagonal     0x8040201008040201
h1-a8 antidiagonal 0x0102040810204080
light squares      0x55AA55AA55AA55AA
dark squares       0xAA55AA55AA55AA55
```

For example, the hexadecimal for the a-file, `0x0101010101010101` converted to decimal gives the number 72340172838076673. If we convert this number to binary (little-endian representation), we get the following bitboard:

```
100000001000000010000000100000001000000010000000100000001
```

As you can see, the total length of this bit string is not 64: we are missing the padding zeros after the MSB bit. If we pad the binary number with zeros till it has a length of 64 bits, it will look as follows:

```
1000000010000000100000001000000010000000100000001000000010000000
```

Formatting the bitboard according to our representation gives us:

1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0

In our representation, this is exactly the `a`-file being highlighted! Check the other hexadecimals for yourself! Now, one final example. Assume we have our chessboard as show in Section 1.5. Lets say we want to highlight the `b`-file, except for `b8`. The bit at `a8` has index square 56. We want to move it to 49. As we can see, this would shift the bits from MSB to the LSB. Recall that this is **RSHFT** with $56-49 = 7$ bits. By essentially performing `0x0101010101010101` $>>$ 7, we get the desired result (verify this yourself!). Some notes:

- Endianness only matters when printing the bitboard. When we want to print a bitboard, we need to access the individual bits in the 64-bit unsigned integer. By accessing the individual bits in memory, we also have to deal with the way values are stored in memory (internal representation). To do this correctly, we need to figure out which representation is currently being used (little-endian or big-endian). Note that endianness only matters for the layout of data in memory. As soon as data is loaded by the processor to be operated on, endianness is completely irrelevant. Shifts, bitwise operations, and so on perform as you would expect (data logically laid out as low-order bit to high) regardless of endianness. This means, for example, that the $>>$ operator always shifts the bits towards the LSB, and that the $<<$ operator always shifts the bits towards the MSB.

- When converting an hex to binary, C++ tends to print the binary in reversed order. You have to watch out for this!

```cpp
##include <stdio.h>
#include <iostream>
#include <cstdlib>
#include <stdint.h>
#include <algorithm>
#include <inttypes.h>
#include <string>

using namespace std;

bool isLittleEndian() {
    short int number = 0x1;
    char *numPtr = (char*) &number;
    return (numPtr[0] == 1);
}

string convertToBitString(uint64_t value) {
    string str(64, '0');
    for (int i = 0; i < 64; i++) {
        if ((1ll << i) & value)
```

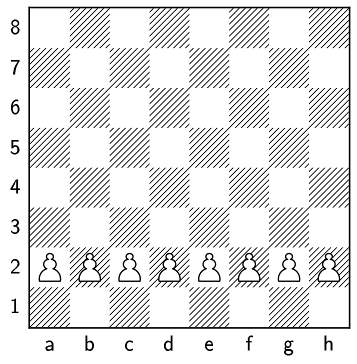
```cpp
            str[63 - i] = '1';
    }
    // The bits are returned in the reversed order, return reversed string.
    reverse(str.begin(), str.end());
    return str;
}

string formatBitBoard(string bitboard) {
    if (bitboard.length() == 8) {
        return bitboard;
    }
    int begin = 8;
    int end = bitboard.length() - 1;

    string subBitString = bitboard.substr(begin, end);
    string rank = bitboard.substr(0, 8);

    return formatBitBoard(subBitString) + "\n" + rank;
}

/*
 *
 */
int main(int argc, char** argv) {
    // Output if system is using little-endian representation
    cout << "Is little-endian representation: " << boolalpha << isLittleEndian() << "\n";

    // Hexadecimal constants for testing the bitboard representation.
    uint64_t LERF_CONSTANTS [8] = {
        0x0101010101010101,
        0x8080808080808080,
        0x00000000000000FF,
        0xFF00000000000000,
        0x8040201008040201,
        0x0102040810204080,
        0x55AA55AA55AA55AA,
        0xAA55AA55AA55AA55
    };
    /*
    sizeof() will get you the total number of bytes allocated for an array.
    To find the number of elements in the array, you divide the total size by
    the size of on element in the array
     */
    for (int i = 0; i < 8; i++) {
        string binary = convertToBitString(LERF_CONSTANTS[i]);
        printf("Hexadecimal: 0x%016llx" PRIx64 "\n", LERF_CONSTANTS[i]);
        cout << "Binary: " << binary << endl;
    }

    string bitString = convertToBitString(0x0101010101010101);
    string formattedBitString = formatBitBoard(bitString);
    cout << "Formatted bitString: \n" << formattedBitString << endl;
    return 0;
}
```

Listing 1: Code Snippit for showing the binary number of the given HEX constants that apply to a Little-Endian Rank-File (LERF) mapping.

2 Common Bitboards

To actually represent all the positions of each piece type and of each color on the chessboard, we need at least 12 bitboards. These bitboards are as follows: **WhitePawns**, **WhiteRooks**, **WhiteKnights**, **WhiteBishops**, **WhiteQueens**, **WhiteKing**, **BlackPawns**, **BlackRooks**, **BlackKnights**, **BlackBishops**, **BlackQueens** and **BlackKing**. I will only describe the bitboards for the white pieces. Once you get an understanding of it, you will be able to make bitboards for the initial black pieces yourself.

2.1 White Pawns


(a) Our chess board.

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	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	**1**	**1**	**1**	**1**	**1**	**1**	**1**
0	0	0	0	0	0	0	0

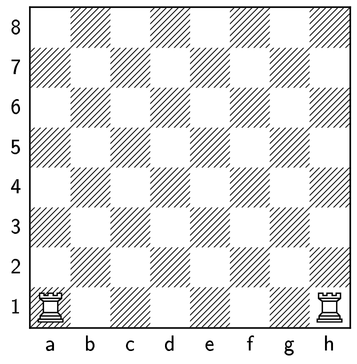
(b) Bitboard representing intial white pawns.

Figure 5: The bitboard of the initial white pawns.

This gives the following bitboard for all the initial white pawns:

```
0000000011111111000000000000000000000000000000000000000000000000
```

2.2 White Rooks


(a) Our chess board.

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	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	0	0	**1**

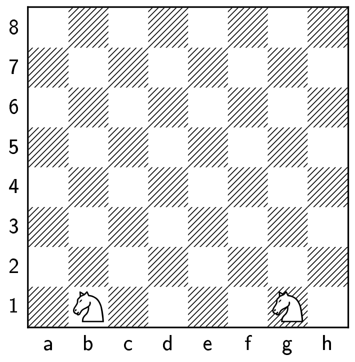
(b) Bitboard representing intial white rooks.

Figure 6: The bitboard of the initial white rooks.

This gives the following bitboard for all the initial white bishops:

```
1000000100000000000000000000000000000000000000000000000000000000
```

2.3 White Knights



(a) Our chess board.

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	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

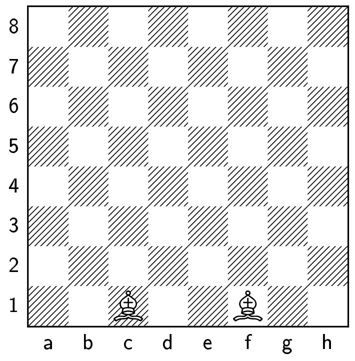
(b) Bitboard representing intial white knights.

Figure 7: The bitboard of the initial white knights.

This gives the following bitboard for all the initial white knights:

```
0100001000000000000000000000000000000000000000000000000000000000
```

2.4 White Bishops



(a) Our chess board.

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	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	**1**	0	0

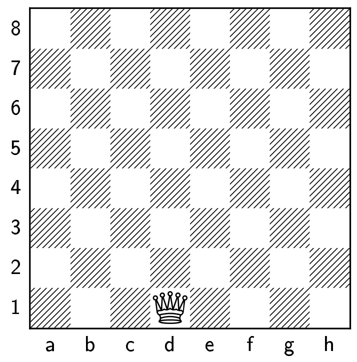
(b) Bitboard representing intial white pawns.

Figure 8: The bitboard of the initial white rooks.

This gives the following bitboard for all the initial white bishops:

```
0010010000000000000000000000000000000000000000000000000000000000
```

2.5 White Queens



(a) Our chess board.

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	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

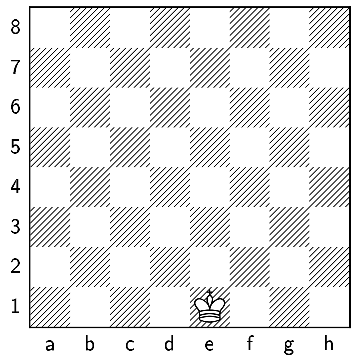
(b) Bitboard representing intial white queen.

Figure 9: The bitboard of the initial white queen.

This gives the following bitboard for the initial white queen:

```
0001000000000000000000000000000000000000000000000000000000000000
```

2.6 White King



(a) Our chess board.

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	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

(b) Bitboard representing intial white king.

Figure 10: The bitboard for the initial white king.

This gives the following bitboard for the initial white king:

```
0000100000000000000000000000000000000000000000000000000000000000
```

3 Move Generation

The non sliding pieces on a chess board are as follows: King, Knight and Pawn. The algorithm for generating moves for non sliding pieces are easier to understand and compute than for sliding pieces. Therefore, we decided to take a look at these pieces first. At this point, we only look at places where a piece can move (non-capture moves and capture moves). As you know, a move is not valid if it puts its own king in check. In this section, we do not take this rule into account. As we will also be talking in the direction a piece can move to, we rely on the compass rose shown in Figure 11. For

```
. NW. . . N . . .NE . .
. . . . . . . . . . . .
. . +7. . +8. . .+9 . .
. . . . \ .|. / . . . .
W . -1. <-.0. ->.+1 . E
. . . . / .|. \ . . . .
. . -9. . -8 . .-7. . .
. . . . . . . . . . . .
. SW. . . S . . .SE . .
```

Figure 11: Our compass rose.

better readability, we use abstractions for shifting into a direction one time. This can be seen in Listing 2. Do not worry if you do not understand the operations that are shown that code block. It will be explained when we are going to find the move set for the king, which follows directly after this section.

```
/*
These are post-shift masks.
This means that we first apply the bitshift,
and then remove unwanted wraps that could occur
in certain circumstances.
*/

uint64_t northOne(uint64_t bitboard){
        return bitboard << 8;
}

uint64_t southOne(uint64_t bitboard){
        return bitboard >> 8;
}

uint64_t eastOne(uint64_t bitboard){
        return (bitboard << 1) & Utils.ClearFile(FILE_A);
}

uint64_t southEastOne(uint64_t bitboard){
        return (bitboard >> 7) & Utils.ClearFile(FILE_A);
}

uint64_t northEastOne(uint64_t bitboard){
        return (bitboard << 9) & Utils.ClearFile(FILE_A);
}

uint64_t westOne(uint64_t bitboard){
        return (bitboard >> 1) & Utils.ClearFile(FILE_H);
}

uint64_t northWestOne(uint64_t bitboard){
        return (bitboard << 7) & Utils.ClearFile(FILE_H);
}

uint64_t southWestOne(uint64_t bitboard){
        return (bitboard >> 9) & Utils.ClearFile(FILE_H);
}
```

Listing 2: Post-shift masks for removing unwanted wraps when shifting a bitboard to obtain a move.

3.1 Non Sliding Pieces

3.1.1 King

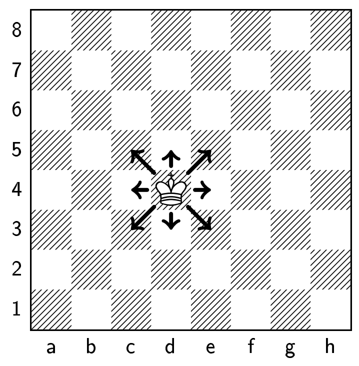


Figure 12: All the king moves.

As you can see in Figure 12, the king can move one square in each direction of our compass rose. Because the position of the king in the corresponding bitboard is one, we can use shifts on the bitboard to generate all spots the king can move to. For each direction the king can move to, we specify the corresponding bit shift.

```
uint64_t getKingMoves(uint64_t king_loc, uint64_t friendly_pieces){
        uint64_t N  = northOne(king_loc);
        uint64_t NE = northEastOne(king_loc);
        uint64_t E  = eastOne(king_loc);
        uint64_t SE = southEastOne(king_loc);
        uint64_t S  = southOne(king_loc);
        uint64_t SW = southWestOne(king_loc);
        uint64_t W  = westOne(king_loc);
        uint64_t NW = northWestOne(king_loc);

        uint64_t king_moves = N | NE | E | SE | S | SW | W | NW;
        /*
        Final AND makes sure we only move to squares that are empty or occupied by an enemy piece.
        */
        return king_moves & ~friendly_pieces;
}
```

The implementation of the used methods can be seen in Listing 2. An important note is that we use the bitboard of the position of the king itself as a seed for the calculation as a “mask” for determining the movements. **Masks** is data that is used in bitwise operations. As you can see, we use the OR operator where all of the generated possible moves represented as bitboards are involved. By using the OR (|), we enable (setting the value of a bit to 1) all the bits to where the king could move. In some cases, a bit shift could wrap a bit to another file that would clearly not be correct as a valid move for the king. We demonstrate this ‘bit wrapping’ by an example. Assume the white king is on square a3. The bitboard of the white king is shown in Figure 13a.

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	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(a) Original king position.

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	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	0	0	0	0

(b) King position when calculating the move to the west.

Figure 13: The position of the kig after shifting to the West when the king's original position is in `a` file.

Lets try to generate the move to the west direction, which essentially is a bit shift of 1 to the LSB, or a right bit shift. We end up with the bitboard shown in Figure 13b. But this position is not reachable at all for the king from its original position! To fix this kind of mistakes, bit masks come into play. When we want to calculate the NW, W and SW positions for a king that is standing in the `a` file, we know for sure that they will produce invalid bitboards representing king moves. To prevent this, we use bit masks. If the king is in the `a` file, we perform the bit shift and clear the `h` file afterwards. The mask for clipping out the `h` file is as follows:

1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0

Figure 14: Bit mask for clipping out the `h` file.

Note that it is not possible for a king to stand in rank 1, to end up in rank 8 after move generation of N, NE or NW. Because the bit will fall of the edge (bit overflow), the resulting bitboard will always contain zero's only. The same applies for a king standing on rank 8 and generating S, SE or SW moves, which are corrected by an bit underflow. In the same line as described for a kin in `a` file, we also have to use a bit mask in `h` file for directions. In summary:

- For the move generation in direction N and W, we do not have to perform a correcting clipping mask;
- For the move generations in direction E, NE and SE, we have to perform a correcting clipping mask: clipping out the `a` file.
- For the move generations in direction W, NW, SW, we have to perform a correcting clipping mask: clipping out the `h` file.

The implementation of the bit shifts in Listing 2 removes the possible wrongly wrapped bits. Take a look at them! Note that we can also improve the move generation for the king. If we OR the bitboards representing the east and west moves with the original position of the king, we get a bitboard we can use to get the moves in the other directions by two shifts:

- Shifting up one rank to determine the NW, N and NE moves.
- Shifting down one rank to determine the SE, S and SW moves.

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	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(a) The original bitboard of the King

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	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	0	0	0	0

(b) The bitboard of the king after shifting to the west.

1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0

(c) The bitboard for clearing the `h` file

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	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

(d) The bitboard of the king after using the mask that clears the `h` file

Figure 15: Preventing wrong bitboards in king move generation, when the king is in `a` file and we generate a move in the west direction. This is done by clipping out the `h` file using a bit mask. This mask is applied after the bit shift (these masks are called post-shift masks).

This is faster then our previous method, which involves calculating all of the 8 directions independently. Make sure to use the bit mask of all the friendly pieces in the end, otherwise, the current position of the king will also be seen as a valid new position.

3.1.2 Knight

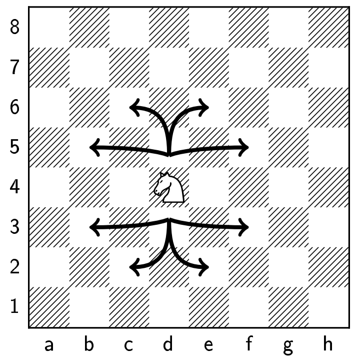


Figure 16: All the knight moves.

In Figure 16, you can see the possible moves of the knight. Looking at the file and row distance, it could to files and rows that are a distance 2 away from their current row or file. In contrary to the masks applied to our king, we need to apply either a mask that clips both the a and b file, or the g and h. Again, we do not have to worry about mask clipping, because the integer overflows and underflows will take care of that for us. The custom wind rose of the knight positions is shown in Figure 17.

```
. . . NNW . . . NNE . . . .
. . . . +15 . +17 . . . . .
. . . . | . . . | . . . . .
NWW +6__| . . . |__+10 NEE.
. . . . \ . . ./. . . . . .
. . . . . >0< . . . . . . .
. .  __ / . . .\ __ . . . .
SWW -10 | . . . | . -6 SEE.
. . . . | . . . | . . . . .
. . . -17 . . -15 . . . . .
. . . SWW . . SEE . . . . .
```

Figure 17: Custom wind rose for the possible knight moves.

Now the following rules apply:

- We have to clip the h file if we calculate spot NNW.
- We have to clip the h file if we calculate spot SSW.
- We have to clip the a file if we calculate spot NNE.
- We have to clip the a file if we calculate spot SSE.
- We have to clip both the h file and the g file if we calculate spot NWW.

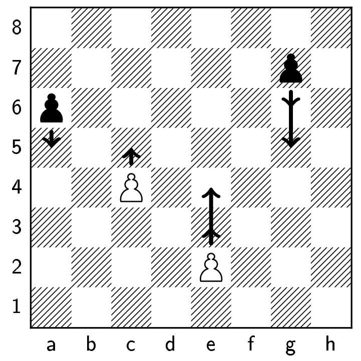
- We have to clip both the h file and the g file if we calculate spot SWW.
- We have to clip both the a file and the b file if we calculate spot NEE.
- We have to clip both the a file and the b file if we calculate spot SEE.

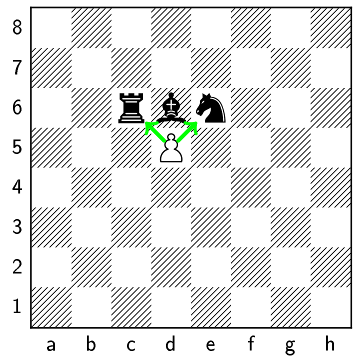
This gives code shown in Listing 3.

```
uint64_t getKnightMoves(uint64_t knight_loc, uint64_t friendly_pieces) {
        uint64_t NNW_clip = clear_file[FILE_H];
        uint64_t SSW_clip = clear_file[FILE_H];
        uint64_t NNE_clip = clear_file[FILE_A];
        uint64_t SSE_clip = clear_file[FILE_A];

        uint64_t NWW_clip = clear_file[FILE_H] & clear_file[FILE_G];
        uint64_t SWW_clip = clear_file[FILE_H] & clear_file[FILE_G];
        uint64_t NEE_clip = clear_file[FILE_A] & clear_file[FILE_B];
        uint64_t SEE_clip = clear_file[FILE_A] & clear_file[FILE_B];

        uint64_t NWW = (knight_loc & NWW_clip) << 6;
        uint64_t NEE = (knight_loc & NEE_clip) << 10;
        uint64_t NNW = (knight_loc & NNW_clip) << 15;
        uint64_t NNE = (knight_loc & NNE_clip) << 17;

        uint64_t SEE = (knight_loc & SEE_clip) >> 6;
        uint64_t SWW = (knight_loc & SWW_clip) >> 10;
        uint64_t SSE = (knight_loc & SSW_clip) >> 15;
        uint64_t SSW = (knight_loc & SSW_clip) >> 17;

        uint64_t knight_moves = NWW | NEE | NNW | NNE | SEE | SWW | SSW | SSE;
        return knight_moves & ~friendly_pieces;
}
```

Listing 3: Calculation of the the knight moveset.

3.1.3 Pawn


(a) Pawn movement: A pawn can move to the square directly in front of itself, if that square is clear. A pawn on its starting rank has the option of moving two squares.


(b) The white pawn at d5 may capture either the black rook at c6 or the black knight at e6, but not the bishop at d6, which blocks the pawn's ability to move directly forward.

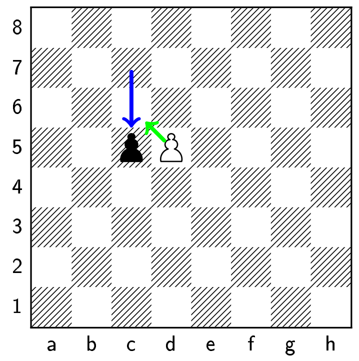

(c) *En passant* capture, assuming that the black pawn has just moved from c7 to c5. The white pawn moves to the c6-square and the black pawn is removed.

Figure 18: Pawn movement.

Move generating for pawns is more difficult compared to other pieces. Pawns are the only piece that can move into one direction. Pawns can only capture diagonally, one square forward and to the left or right. Another unusual move is the *en passant* capture. When an enemy pawns moves forward two squares instead of one, the square behind the pawn that just made this move, becomes a possible move for the enemy pawn. Note that the pawn that moved two squares passed over a square that was attack by an enemy pawn. That enemy pawn, which would have been able to capture the moving pawn had it advanced only one square, is entitled to capture the moving pawn "in passing" as if it had advanced only one square. The capturing pawn moves into the empty square over which the moving pawn passed, and the moving pawn is removed from the board. Note that an en passant capture must take place directly when the enemy pawn moves two square forward. If a player chose not to play the en passant move, the move cannot be applied to the same pawn when the player has turn again. Pawns can also promote into other pieces when reaching the opposite side of the board (the first rank of the other player). When the a pawn reaches this position, it is promoted into another piece of that players's choice: a queen, rook, bishop or knight of the same color. Most of the times, players choose to promote their pawn into a queen. When an other piece is chosen, it is also called "underpromotion".

Now we know the possible moves of the pawn, lets dive into the move generation using bitboards. Let first talk about the pawn pushes, which do not involve captures. To check whether pawn can move forward one square is easily determined by shifting to the North when the pawn is white, and to the South when the pawn is black. Finally, we intersect the resulting bitboard with the bitboard containing all of empty squares. The code is shown in Listing 4. For

```
uint64_t whitePawnsSinglePushMoves(uint64_t whitePawns, uint64_t emptySquares){
        return northOne(whitePawns) & emptySquares;
}

uint64_t blackPawnsSinglePushMoves(uint64_t blackPawns, uint64_t emptySquares){
        return southOne(blackPawns) & emptySquares;
}
```

Listing 4: Calculation of the single push squares for the pawns.

calculating the double push targets, we shift the single push square in the right direction and check whether the new rank of the pawn is empty (rank 4 for white pawns, rank 5 for black pawns). The code is shown in Listing 5. To get the

```
uint64_t whitePawnsDoublePushMoves(uint64_t whitePawns, uint64_t emptySquares){
        // mask in which the bits on rank 4 are turned on.
        const uint64_t rank4 = 0x00000000FF000000;
        uint64_t singlePushs = whitePawnsSinglePushMoves(whitePawns, emptySquares);
        return northOne(singlePushs) & rank4 & emptySquares;
}

uint64_t blackPawnsDoublePushMoves(uint64_t blackPawns, uint64_t emptySquares){
        // mask in which the bits on rank 5 are turned on.
        const uint64_t rank5 = 0x000000FF00000000;
        uint64_t singlePushs = blackPawnsSinglePushMoves(blackPawns, emptySquares);
        return southOne(singlePushs) & rank5 & emptySquares;
}
```

Listing 5: Calculation of the double push squares for the pawns.

pawns that are able to push one square, we shift the free squares towards the rank closest to the player and intersect the result with all the pawns. We describe the calculations to get the set of source squares of pawns being able to double push for white pawns: We start from the rank 4 for white pawns. We shift the rank, intersected with the empty squares set, towards rank 3. Now we intersect it again with the empty squares to verify if rank 3 is empty. Finally, we call `whitePawnsAbleToPush()` with the white pawns and this calculated empty rank 3. For black pawns, the calculation can be done similar for black. The code snippet is shown in Listing 6. Since double pushing triggers determination of

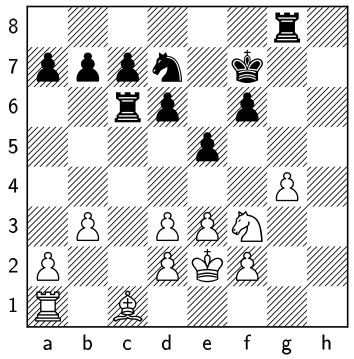
```
uint64_t whitePawnsAbleToPush(uint64_t whitePawns, uint64_t emptySquares){
        return southOne(emptySquares) & whitePawns;
}

uint64_t whitePawnsAbleToDoublePush(uint64_t whitePawns, uint64_t emptySquares) {
        const uint64_t rank4 = 0x00000000FF000000;
        uint64_t emptyRank3 = southOne(emptySquares & rank4) & emptySquares;
        return whitePawnsAbleToPush(whitePawns, emptyRank3);
}
```

Listing 6: Calculation of the source squares of the white pawns that are able to double push.

en passant target square, it makes sense to serialize both sets separately for different move encoding. The same applies to promotion moves.

3.2 Sliding Pieces & Magic Bitboards

The sliding pieces in chess are the rook, bishop and the queen. As these pieces can have multiple rays of squares they can move to, we also have to deal with obstructions blocking certain rays. Friendly and enemy pieces can block an attack ray in one particular direction. Although we can use the shifting operations, as we have seen in the calculation of the move set for pawns, knights and the king, for calculating the moves for these pieces, other methods exists as well. For the sliding pieces, we will use so called magic bitboards. Before diving into these magic bitboards, lets explain some terminology by showing an example. Lets assume we are dealing with the chess board shown in Figure 19. The corresponding bitboard is shown in Figure 19b.



(a) Example position.

0	0	0	0	0	0	**1**	0
1	**1**	**1**	**1**	0	**1**	0	0
0	0	**1**	**1**	0	**1**	0	0
0	0	0	0	**1**	0	0	0
0	0	0	0	0	0	**1**	0
0	**1**	0	**1**	**1**	**1**	0	0
1	0	0	**1**	**1**	**1**	0	0
1	0	**1**	0	0	0	0	0

(b) The bitboard of the example position. We also call this the **occupancy bitboard**.

0	0	**1**	0	0	0	0	0
0	0	**1**	0	0	0	0	0
1	**1**	0	**1**	**1**	**1**	**1**	**1**
0	0	**1**	0	0	0	0	0
0	0	**1**	0	0	0	0	0
0	0	**1**	0	0	0	0	0
0	0	**1**	0	0	0	0	0
0	0	**1**	0	0	0	0	0

(c) The mask that removes all the pieces that cannot influence the possible moves of the rook on `c6`. We call this the **attack bitboard**.

0	0	0	0	0	0	0	0
0	0	**1**	0	0	0	0	0
0	0	0	**1**	0	**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	0	0
0	0	0	0	0	0	0	0
0	0	**1**	0	0	0	0	0

(d) Result of the bitboard representing the example position AND-ed with the corresponding mask of a rook on `c6`. We also call this the **blocker bitboard**.

Figure 19: From chess board to bitboard.

Lets say we want to calculate the moves for the rook on `c6`. As we know, rooks move in the following directions: N, E, S and W. In this particular case, the rook on `c6` does not care about pieces that are not on the same rank or on the same file as where he currently is standing. Therefore, we use the mask shown in Figure 19c. This particular mask is essentially a bitboard where all the possible moves of the rook on `c6` are present, as if there were no pieces blocking its attack rays. We call such a mask for a piece on a particular square an **attack bitboard**. If we `AND` this **attack bitboard** with the bitboard representing our example position (Figure 19b), we get the so called **blocker bitboard** as shown in Figure 19d. Now we want the legal moves for our rook. A simple algorithm would be to iterate over the squares in all the positions (N, E, W and S) and mark all of the empty squares. This is done until an occupied square is found. We then return the marked bits as legal moves.

However, a better method exists. We have to make two observations:

1. We do not care about the edges on our board: if there is an enemy piece on an edge, we can always capture it. Therefore, an edge piece does not contribute to the information we need for determining the set of legal moves.

0	0	0	0	0	0	0	0
0	0	**1**	0	0	0	0	0
1	**1**	0	**1**	0	0	0	0
0	0	**1**	0	0	0	0	0
0	0	**1**	0	0	0	0	0
0	0	**1**	0	0	0	0	0
0	0	**1**	0	0	0	0	0
0	0	**1**	0	0	0	0	0

Figure 20: Move set corresonding to the blocker bitboard shown in Figure 19d.

2. So what is the maximum number of blocker bitboards when we consider all of the squares on the chess board? Well, lets place a rook on `a1`. As we said previously, we do not count the edge squares as blockers. In the case of a rook standing on `a1`, the edges are: `a8` and `h1`. This leaves us with 6 positions to the north which could be occupied (`a2` through `a7`) and 6 positions to the east that could be occupied (`b1` through `g1`). Note that for other positions, the squares that could be occupied would be less or equal to 12. Thus, there are at maximum 2^{12} blocker bitboards for a random square occupied by a rook. This number is not big at all: if we would have a lookup table, we can simply look up the move set for each of the 2^{12} states.

The next question that arises is: how do we go from a blocker bitboard to an index in this lookup table? This is where magic bitboards come in. The idea is as follows. We find a **magic number** m_i for every square such that multiplying the blocker bitboard by m_i gives a perfect hash into the indices of the lookup table. A magic number for a rook on `c6` is:

$$m_i = 432627108460691524$$

Lets review our blocker bitboard as shown Figure 19d. As a 64-bit number, this is simply $b = 1169880371953668b = 1169880371953668$. Now we calculate

$$mb = 4692364060652732688 \pmod{64}$$

Because there are only ten relevant occupancy bits for a rook on `c6`, we only retrieve the top **10** bits of this hash. This gives the following index:

$$\text{idx} = mb >> 54 = 2083$$

Now we can use this index to retrieve the valid moves for a rook on `c6` with blocker bitboard shown in Figure 19d. Simple right? But how do we find these magic numbers and what do we need to take into account when finding such a hash function? Well, lets again take a look at the blocker bitboard shown in Figure 19d. We can easily see that the move set for this blocker board is equal to the moves shown in Figure 20. But is the blocker bitboard as shown in Figure 19d the only blocker bitboard that maps to the move set in Figure 20? No, it is not! As we can see in Figure 19, multiple blocker bits are set to the east (E) of the rook on `c6`. But does it matter if there is another blocker bit if there was already a blocker bit before that one (closer to the piece)? Well, only blocker bits that are the first to block the ray matter. Therefore, all of these blocker bitboard that share the same ‘first’ blocker bits map to the same move set. The reduced blocker bitboard for Figure 19d can be seen in Figure 21.

0	0	0	0	0	0	0	0
0	0	**1**	0	0	0	0	0
0	0	0	**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	0	0	0	0
0	0	0	0	0	0	0	0
0	0	**1**	0	0	0	0	0

Figure 21: Reduced blocker bitboard for the blocker bitboards shown in Figure 19d. The blockers are as follows: `c7`, `d6` and `c1`.

In Figure 22 we can see some variations of the blocker bitboards shown Figure 19d that all can be reduced to the bitboard shown in Figure 21.

0	0	0	0	0	0	0	0
0	0	**1**	0	0	0	0	0
0	0	0	**1**	**1**	**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	0	0
0	0	0	0	0	0	0	0
0	0	**1**	0	0	0	0	0

(a) Blocker bitboard variation 1.

0	0	**1**	0	0	0	0	0
0	0	**1**	0	0	0	0	0
0	0	0	**1**	0	0	0	**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
0	0	0	0	0	0	0	0
0	0	**1**	0	0	0	0	0

(b) Blocker bitboard variation 2.

Figure 22: Variations of the blocker bitboard as shown Figure 19d. All of these variations have the same reduced blocker bitboard.

Now that we know that not all of the bits matter in finding the move set for a certain blocker bitboard, we continue our journey to find these so-called magic numbers. Well, lets first calculate all the possible blocker bitboards for the piece on a specific square. For each blocker bitboard variation, we calculate the corresponding reduced bitboard. We also maintain a database which we can access to see which index maps to which reduced bitboard. We then multiply the blocker bitboard variation by a random bitboard (which could be a possible magic number as we can see later). Now we obtain the index by shifting the result of the multiplication by n bits to the right, where n is the number of bits set in the corresponding attack set of the piece on that square (make sure that the corresponding edge squares are not set!). Now we access the database using the calculated index. If the index is not 0, we need to verify if the stored move set is equal to the move set that we would expect for the given blocker bitboard variation. If the move sets are the same, our mapping is still valid because it will map variations of the same blocker bitboard to the same index. We continue this validation process for all the different blocker bitboards to see whether our potential magic number maps the right blocker bitboard variations for a given square to the right index. If no clash occurs during our search we found a magic number for a square!

So in general, generating moves using magic bitboards consists of four steps:

1. Pick some random 64-bit number, call it **magic**. Get all the possible the blocker bitboard variations of the piece.
2. Create a database of size 2^{bits} (where bits in the number of bits set in the original attack bitboard).
3. For each blocker bitboard variation:
   (a) Calculate the corresponding **move set** for the variation.

   (b) Create an index by multiplying variation with magic and shift it right with $64 - \text{bits}$.
   (c) **If** database[index] is not 0 **and** the move set stored at the index in our mapping bitboard is **not** equal to the move set we expected it to have, then we have a clash. If this happens, we need to clear our mapping and generate a new number for which we want to check whether it is a magic number.
   (d) If the move sets are equal or there was no move set stored at this index, we store the move set for the current variation at `database[index]` and continue.

4. If the loop ends with no clashes, then magic is a valid magic number for the piece on the square we were currently looking at.