

Chess Engine for HPC

Alexis Evaristo

February 25, 2026

Contents

1	Introduction	2
2	Counting	2
2.1	Binary	2
2.2	Hexadecimal	2
3	Bits Operators	2
3.1	Left Shift Operator	3
3.2	Right Shift Operator	3
3.3	AND & Operator	3
3.4	OR Operator	3
3.5	XOR ^ Operator	3
3.6	NOT ~ Operator	3
4	The Board	4
4.1	Bit Board	4
4.2	Accessing a Piece	5
4.3	Displaying the Board	5
5	Moving Pieces	5
5.1	How to describe a move	6
5.2	How to play a move	6
6	Advanced Game Mechanics	6
6.1	Sliding Pieces (Raycasting)	6
6.2	Special Moves: Castling & Promotion	7
7	Evaluation Function	7
7.1	Material Evaluation	7
7.2	Positional Evaluation (Piece-Square Tables)	7
8	Search Algorithm	8
8.1	Alpha-Beta Pruning	8
8.2	Move Ordering	8
9	Performance Analysis & Benchmarking	8
9.1	Methodology	8
9.2	Results	8
9.3	Discussion: The Power of Bitboards and Pruning	9
9.4	Future Work: The Transposition Problem	9

1 Introduction

I wrote this report for anyone interested in coding an advanced chess engine. This report summarize briefly how I build mine without going too deep into details. There are plenty of ways to program a chess engine, this one can be seen as a state of the art.

2 Counting

This section may seem a bit simple but yet is very important. Through this report we will essentially manipulate binary and hexadecimal numbers. These aren't exhaustive explanations.

2.1 Binary

Everyday we use decimal numbers which means that we can write an integer with this writing

$$n = \sum_{i \geq 0} a_i \times 10^i$$

with

$$a_i \in \{1, 2, \dots, 9\}$$

For example $345 = 3 \times 10^2 + 4 \times 10^1 + 5 \times 10^0$. More generally, it is possible to write an integer with base $b \in \mathbf{N}^*$

$$n = \sum_{i \geq 0} a_i \times b^i$$

with

$$a_i \in \{1, 2, \dots, b-1\}$$

When we consider, $b = 2$ we obtain binary numbers. For example $101_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5_{10}$.

An important propriety we will often use is that, multiplying a number by 2 shift its binary representation to the left by adding one zero on the right. For example $1010_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 10_{10}$. That stands no matter the base b . The demonstration is immediate regarding the writing we just gave.

2.2 Hexadecimal

The hexadecimal representation is when $b = 16$. We note $A = 10$, $B = 11$, $C = 12$, $D = 13$, $E = 14$, $F = 15$. We will use hexadecimal numbers because they are easy to handle in C language, and because they fit surprisingly well our subject. One important property of hexadecimal numbers is that they can have an equivalent writing with 4 bits. As an example, we have $F_{16} = 1111_2$ and $A_{16} = 1010_2$, such that $FA_{16} = 11111010_2$. That is possible because $16 = 2^4$, so there is a shift of 4 to the left.

3 Bits Operators

In this chapter, we define operators and give examples of how to use them. They are the angular stone of our future engines. The following examples are purely educational, they are not real C language code.

3.1 Left Shift Operator

Our first operator is the Left Shift Operator.

```
1 x = 00001010
2 k = 2
3
4 1 << k = 00000100
```

In this example, we have the variable x which is equal to 10 in base 10 ($x = 2^0 + 2^1 + 2^0 + 2^3 + \dots = 2^1 + 2^3 = 2 + 8 = 10$). The left shift operator simply multiplies by 2^k . If we take line 4 of the example, we can find that $1 \times 2^2 = 4 = 0000\ 0100_2$.

```
1 x << 1 = 00010100 (20)
2 x << 2 = 00101000 (40)
```

3.2 Right Shift Operator

The Right Shift Operator is similar to the previous one but correspond to a division by 2^k .

```
1 x = 00101000 (40)
2 x >> 2 = 00001010 (10)
```

3.3 AND & Operator

The AND & Operator checks if two bits of two expressions correspond or not.

```
1 x = 00001010
2 y = 00000100
3 x & y = 00000000
4
5 x = 00001010
6 y = 00001000
7 x & y = 00001000
```

3.4 OR | Operator

The OR | Operator checks whether at least one bit of two expressions is active.

```
1 x = 00001010
2 y = 00000001
3 x | y = 00001011
```

3.5 XOR ^ Operator

The XOR ^ Operator deactivates a bit if it is being active in the two expressions. This property makes it perfectly reversible, which is highly valuable for making and unmaking moves on a chess board.

```
1 x = 00001010
2 y = 00000010
3 x ^ y = 00001000
```

3.6 NOT ~ Operator

The NOT ~ Operator corresponds to a toggle of all active bits into inactive and all the inactive bits into active.

```
1 x = 00001010
2 ~x = 11110101
```

4 The Board

Now that we can manipulate bits, we can define our board. One way to do so is by using an array of 64 bits.

4.1 Bit Board

```
1 #include <stdint.h>
2 uint64_t board = 0ULL;
```

Each bit indicate if there is a piece present or not. Since there are 6 different types of pieces and 2 colors, we will define a structure with 12 **uint64_t board**.

*Let's note that **ULL** stands for **Unsigned Long Long** (≥ 64 bits).*

```
1 #include <stdint.h>
2
3 typedef struct {
4     uint64_t white_pawns;
5     uint64_t white_rooks;
6     uint64_t white_knights;
7     uint64_t white_bishops;
8     uint64_t white_queens;
9     uint64_t white_king;
10
11     uint64_t black_pawns;
12     uint64_t black_rooks;
13     uint64_t black_knights;
14     uint64_t black_bishops;
15     uint64_t black_queens;
16     uint64_t black_king;
17
18     int castling_rights;
19     int en_passant_square;
20     int halfmove_clock;
21 } Board;
```

We can now initialize our boards.

```
1 Board board = {
2     .white_pawns    = 0x00000000000000FF00ULL,
3     .white_rooks    = 0x000000000000000081ULL,
4     .white_knights  = 0x000000000000000042ULL,
5     .white_bishops  = 0x000000000000000024ULL,
6     .white_queens   = 0x000000000000000008ULL,
7     .white_king     = 0x000000000000000010ULL,
8
9     .black_pawns    = 0x00FF00000000000000ULL,
10    .black_rooks     = 0x810000000000000000ULL,
11    .black_knights   = 0x420000000000000000ULL,
12    .black_bishops   = 0x240000000000000000ULL,
13    .black_queens    = 0x080000000000000000ULL,
14    .black_king      = 0x100000000000000000ULL,
15
16    .castling_rights = 15, .en_passant_square = -1, .halfmove_clock = 0
17};
```

These long lines of hexadecimal code can be intimidating at first, let's explain them. The first **0x** indicate that we use hexadecimal. One hexadecimal digit is coded on 4 bits, and there are 16 digits which represents $16 \times 4 = 64$ bits, exactly the size of our board. If we take the example of the white paws board, we end up with the following representation :

```
1 00000000 00000000 00000000 00000000
2 00000000 00000000 11111111 00000000
```

We have, indeed, $F = 1111_2$ and $0_{16} = 0000_2$.

4.2 Accessing a Piece

We detail here how to properly get access to a piece on the board. This code is simple, we code the integer 1 on 64 bits (63 zeros and 1 one). We move the activated bit on the desired square with the Left Shift Operator. Then we check with the AND & Operator if a piece match in one of the boards at this place.

```
1 char get_piece_on_square(Board *b, int square) {
2     uint64_t mask = 1ULL << square;
3
4     if (b->white_pawns & mask) return 'P';
5     if (b->white_knights & mask) return 'N';
6     if (b->white_bishops & mask) return 'B';
7     if (b->white_rooks & mask) return 'R';
8     if (b->white_queens & mask) return 'Q';
9     if (b->white_king & mask) return 'K';
10
11     if (b->black_pawns & mask) return 'p';
12     if (b->black_knights & mask) return 'n';
13     if (b->black_bishops & mask) return 'b';
14     if (b->black_rooks & mask) return 'r';
15     if (b->black_queens & mask) return 'q';
16     if (b->black_king & mask) return 'k';
17
18     return '.';
19 }
```

4.3 Displaying the Board

We can now print pretty easily the board.

```
1 void print_board(Board *b) {
2     printf("\n");
3     for (int rank = 7; rank >= 0; rank--) {
4         printf("%d ", rank + 1);
5         for (int file = 0; file < 8; file++) {
6             int square = rank * 8 + file;
7             printf("%c ", get_piece_on_square(b, square));
8         }
9         printf("\n");
10    }
11    printf("\n  a b c d e f g h\n\n");
12 }
```

The display of the board should look just like this.

```
1 8 r n b q k b n r
2 7 p p p p p p p p
3 6 . . . . . . . .
4 5 . . . . . . . .
5 4 . . . . . . . .
6 3 . . . . . . . .
7 2 P P P P P P P P
8 1 R N B Q K B N R
9
10 a b c d e f g h
```

5 Moving Pieces

This section is dedicated to moving pieces on the boards using the different operators we introduced.

5.1 How to describe a move

We want to extract all the possible moves and for that we need to be able to represent these moves. To be highly optimized for CPU operations, we encode a move using a single 32-bit integer.

```
1 bits 0-5   : starting point (0-63)
2 bits 6-11  : ending point (0-63)
3 bits 12-15 : piece type
4 bits 16-19 : captured piece (if any)
5 bits 20-23 : promotion piece (if any)
6 bits 24-31 : special flags (quiet, capture, castling...)
```

We can use a macro to simplify the way we encode and decode our moves.

```
1 #define ENCODE_MOVE(from, to, piece, captured, promotion, flags) \
2   ( (from) | ((to) << 6) | ((piece) << 12) | ((captured) << 16) | ((promotion) \
3     << 20) | ((flags) << 24) )
4 #define GET_FROM(move)      ( (move) & 0x3F )
5 #define GET_TO(move)       ( ((move) >> 6) & 0x3F )
```

We keep all the generated moves safe in a dedicated structure called a ‘MoveList’, assuming a theoretical maximum of 256 legal moves per turn.

```
1 typedef struct {
2     uint32_t moves[256];
3     int count;
4 } MoveList;
```

5.2 How to play a move

To physically apply a move on our bitboards, we heavily rely on the XOR ^= operator. It elegantly toggles the bit off on the starting square, and toggles it on at the destination.

```
1 void make_move(Board *b, uint32_t move, int color) {
2     int from = GET_FROM(move);
3     int to = GET_TO(move);
4     int piece = GET_PIECE(move);
5     int flags = GET_FLAGS(move);
6
7     uint64_t move_mask = (1ULL << from) | (1ULL << to);
8
9     if (color == WHITE) {
10         switch (piece) {
11             case PAWN:    b->white_pawns ^= move_mask; break;
12             case KNIGHT: b->white_knights ^= move_mask; break;
13             case BISHOP:  b->white_bishops ^= move_mask; break;
14             case ROOK:    b->white_rooks   ^= move_mask; break;
15             case QUEEN:   b->white_queens  ^= move_mask; break;
16             case KING:    b->white_king    ^= move_mask; break;
17         }
18     }
19     // Handle captures and special moves below...
20 }
```

6 Advanced Game Mechanics

6.1 Sliding Pieces (Raycasting)

For pieces that slide across the board like Rooks, Bishops, and Queens, we cannot simply use a static pre-calculated mask. Their movement is blocked by other pieces (both friendly and

enemy). To solve this, we generate their attacks dynamically by "raycasting" in every allowed direction until we hit an occupied square.

```

1 uint64_t get_rook_attacks_fallback(int square, uint64_t occupancy) {
2     uint64_t attacks = 0ULL;
3     int tr = square / 8, tf = square % 8;
4
5     // Raycast Up
6     for (int r = tr + 1; r <= 7; r++) {
7         attacks |= (1ULL << (r * 8 + tf));
8         if (occupancy & (1ULL << (r * 8 + tf))) break; // Stop at first piece
9     }
10    // Similar loops for Down, Left, and Right...
11    return attacks;
12 }

```

6.2 Special Moves: Castling & Promotion

Castling involves moving two pieces (King and Rook) at the same time. We check if the path is clear using a simple bitwise AND with the occupancy board, and ensure the squares are not under attack. If valid, the move is flagged with FLAG_CASTLING.

When pawns reach the final rank, they trigger a promotion. Because of our 32-bit encoding, we can easily generate four different moves for a single pawn push, utilizing the bits 20-23 to store the promoted piece type.

7 Evaluation Function

Once the legal moves are generated, the engine needs a way to evaluate if a resulting position is favorable or not.

7.1 Material Evaluation

The most basic evaluation counts the pieces. We assign arbitrary values to each piece type (e.g., Pawn = 100, Queen = 900). To count the active bits efficiently, we use a GCC intrinsic function `__builtin_popcountll` which compiles down to a single hardware instruction.

```

1 int evaluate_material(Board *b) {
2     int score = 0;
3     score += __builtin_popcountll(b->white_pawns) * 100;
4     score += __builtin_popcountll(b->white_queens) * 900;
5     // ... subtract black pieces ...
6     return score;
7 }

```

7.2 Positional Evaluation (Piece-Square Tables)

Material alone is not enough to play good chess. A Knight in the center of the board is vastly superior to a Knight stuck in a corner. We introduce Piece-Square Tables (PST) to grant bonus or malus points depending on the exact coordinate of the piece.

```

1 // Bonus to centralize Knights (malus on edges)
2 const int knight_pst[64] = {
3     -50, -40, -30, -30, -30, -30, -40, -50,
4     -40, -20,  0,  5,  5,  0, -20, -40,
5     -30,  5, 10, 15, 15, 10,  5, -30,
6     // ...
7 };

```

8 Search Algorithm

8.1 Alpha-Beta Pruning

To look ahead into the future, we use the Minimax algorithm enhanced with Alpha-Beta pruning. By keeping track of the best score we can force ('alpha') and the worst score the opponent can force on us ('beta'), we can completely skip ("prune") evaluating branches that are demonstrably worse than a previously explored option.

```
1 int alpha_beta(Board *b, int depth, int alpha, int beta, int color, int
  eval_type) {
2     if (depth == 0) return evaluate_positional(b);
3
4     // ... Move generation ...
5
6     for (int i = 0; i < list.count; i++) {
7         make_move(b, list.moves[i], color);
8         int score = -alpha_beta(b, depth - 1, -beta, -alpha, 1 - color,
  eval_type);
9         // ... Unmake move ...
10
11         if (score > alpha) alpha = score;
12         if (alpha >= beta) break; // Pruning!
13     }
14     return alpha;
15 }
```

8.2 Move Ordering

Alpha-Beta pruning works best when the optimal move is evaluated first. To achieve this, we sort the 'MoveList' before diving into the recursive tree. Captures are given an arbitrary bonus of +10000 and Queen promotions +9000, ensuring the engine investigates aggressive and highly impactful moves before quiet ones.

9 Performance Analysis & Benchmarking

To validate the efficiency of our bitboard architecture and our Alpha-Beta search algorithm, we conducted a series of benchmarks on the CPU. In the field of chess engine development, the standard metric for performance is not FLOPS (Floating-Point Operations Per Second), as bitboards rely entirely on integer and bitwise logic, but rather **NPS (Nodes Per Second)**.

9.1 Methodology

The benchmark was performed from the official starting position of a chess game. The engine was tasked with finding the best move at increasing depths. We measured the total computation time using the standard `<time.h>` library and counted the exact number of nodes evaluated by incrementing a global counter at each recursive call of the `alpha_beta` function. The C code was compiled using GCC with aggressive optimization flags (`-O3 -march=native`) to fully leverage the CPU's architecture.

9.2 Results

The results of the single-core CPU benchmarks are summarized in Table 1.

Depth	Evaluated Nodes	Time (seconds)	Speed (NPS)
Depth 6	227,362	0.0913	$\approx 2,500,000$
Depth 7	1,327,027	0.4718	$\approx 2,800,000$
Depth 8	12,895,763	3.9007	$\approx 3,300,000$
Depth 9	61,428,964	20.8654	$\approx 2,900,000$

Table 1: Benchmark results of the engine on a single CPU core from the starting position.

9.3 Discussion: The Power of Bitboards and Pruning

These results highlight two massive engineering achievements in our engine:

1. **The Efficiency of Alpha-Beta Pruning:** At Depth 6, the theoretical number of possible chess games without pruning would be in the hundreds of millions. Our engine evaluated exactly 227,362 nodes. This proves that our Move Ordering (sorting captures and promotions first) allows the Alpha-Beta algorithm to prune over 99% of the mathematical tree while still guaranteeing the optimal move.
2. **The Raw Speed of Bitwise Operations:** Reaching a peak of 3.3 million Nodes Per Second on a single core is highly competitive. By encoding the board state in 64-bit integers and the moves in 32-bit integers, we avoid costly loops and memory allocations. The CPU evaluates a board state and generates moves using native hardware instructions (like `__builtin_ctzll` and `__builtin_popcountll`), acting as a pseudo-SIMD (Single Instruction, Multiple Data) architecture.

9.4 Future Work: The Transposition Problem

While the engine is exceptionally fast, it currently suffers from "amnesia". In chess, different move orders can lead to the exact same board position (a concept known as a Transposition). For example, playing 1. `e4 e5` 2. `Nf3` results in the same position as 1. `Nf3 e5` 2. `e4`.

Currently, our engine evaluates this identical subtree multiple times, wasting valuable computational time. The next logical evolution for this High-Performance Computing project would be to implement **Zobrist Hashing** and a **Transposition Table (TT)**. By assigning a unique 64-bit hash to every unique board state, the engine could store previous evaluations in a large hash map in RAM. If it encounters the same position again, it could instantly retrieve the score in an $O(1)$ memory lookup, drastically increasing the search depth achievable within tournament time limits.