Production-Grade Bitboard Chess Engine

Engine Core Specification (Extended v2)

1. Purpose & Scope

This document defines the engine core of a modern chess engine.

The engine core is responsible for:

- Representing chess positions efficiently
- Generating pseudo-legal and legal moves
- Applying and undoing moves deterministically
- Maintaining incremental state (hashes, clocks, rights)
- Guaranteeing correctness, speed, and thread safety

Non-Goals (Explicitly Excluded):

- Search algorithms (PVS, AlphaBeta, etc.)
- Evaluation logic (HCE, NNUE, etc.)
- Time management
- UCI command handling

The engine core must be stable, deterministic, and reusable.

2. Design Principles

1. **Correctness First**
    - Any performance gain that risks state corruption is unacceptable.
2. **Incremental State Only**
    - No recomputation inside `makeMove()` / `unmakeMove()`.
3. **Zero Dynamic Allocation**
    - No heap allocation in move generation or state updates.
4. **Determinism**
    - Same input → same move list → same hash, every time.
5. **Search-Agnostic**
    - The engine core must not assume how it will be searched.

3. Board Representation

3.1 Bitboard Model

- Use `uint64_t` for all bitboards
- **LERF mapping:**
    - A1 = bit 0
    - H8 = bit 63
- **One bitboard per:**
    - Piece type per color

    - Aggregate white / black occupancy
- **Example:**
    - `whitePawns , blackKnights`
    - `whitePieces , blackPieces`
    - `allPieces`
- No redundancy unless it improves speed.

3.2 Square & Piece Encoding

- **Squares:** 0–63
- **Pieces:** compact enum (pawn = 0 … king = 5)
- **Colors:** WHITE = 0, BLACK = 1
- Avoid polymorphism. Use plain data.

4. Position Object

4.1 Stored State (Undoable)

The `Position` object must store only state that changes. **Mandatory fields:**

- Piece bitboards (per color, per type)
- Side to move
- Castling rights (4-bit mask)
- En-passant square (or NONE)

- Halfmove clock
- Fullmove number
- Zobrist hash (incremental)
- Pawn hash
- Material hash

4.2 Derived State (Never Stored)

These must be computed on demand:

- Occupancy bitboards
- Attacked squares
- Check / double check status
- Pinned pieces

Derived state must **never** be pushed onto the undo stack.

5. Move Representation

5.1 16-Bit Move Encoding

- Bits 0–5 : from square (0–63)
- Bits 6–11 : to square (0–63)
- Bits 12–13 : move type
- Bits 14–15 : promotion piece

Move Types:

- Quiet
- Capture
- En-passant
- Castling

Promotion encoding:

- Knight, Bishop, Rook, Queen

All moves must be self-describing. No inference allowed during `makeMove()`.

5.2 Move Lists

- Fixed-size array: `Move moves[256]`
- No `std::vector`
- Explicit count
- **Guarantees:**
  - Cache friendliness
  - No allocator involvement
  - SMP safety

6. Move Generation Pipeline

6.1 Generation Stages

1. **Pseudo-Legal Move Generation**
    - Ignores king safety
    - Includes illegal moves
2. **Legality Check**
    - Ensures king is not left in check
    - Only king safety is verified
3. **Legal Move List**
    - Wrapper around (1) + (2)

These stages must remain logically separate.

6.2 Special Move Handling

- **Castling:**
    - Rights validated explicitly
    - Path must be empty
    - No square crossed may be attacked
- **En-passant:**
    - Only legal if it does not expose own king
    - Must validate captured pawn existence
- **Promotions:**
    - Always generate all four promotions

- Ordered: Q, R, B, N

7. Sliding Piece Attacks

7.1 Primary Mechanism

- BMI2 PEXT/PDEP if available
- Runtime CPU feature detection

7.2 Fallback

- Magic bitboards (precomputed)
- Same API as BMI2 path

7.3 Abstraction Layer

Do not expose BMI2 or Magics to callers. **Example API:**

- `rookAttacks(square, occupancy)`
- `bishopAttacks(square, occupancy)`

8. Make / Unmake Move

8.1 MakeMove Requirements

- Fully incremental
- **Updates:**
  - Piece bitboards

    - Zobrist hash
    - Pawn / material hash
    - Castling rights
    - EP square
    - Halfmove clock
- No recomputation. No branching based on move history.

8.2 Undo Stack (UndoInfo)

`UndoInfo` must store only what is needed to revert:

- Captured piece (if any)
- Previous castling rights
- Previous EP square
- Previous halfmove clock
- Zobrist delta

`UndoInfo` must be POD and cache-friendly.

9. Hashing & Game History

9.1 Zobrist Hashing

- Incremental XOR updates
- Separate random keys for:

- Piece on square
- Side to move
- Castling rights
- EP file

9.2 Repetition Detection

The engine must detect 3-fold repetition to claim draws.

- **The Problem:** The current Zobrist hash describes the *current* position, but does not track the *path* taken to get there.
- **The Adjustment:**
  - The `Position` object (or a wrapping `Game` class) must maintain a history array of Zobrist hashes for the current game.
  - This history is appended to during `makeMove` and popped during `unmakeMove`.
- **Core API Requirement:**
  - `bool isRepetition(int plyLimit)`
  - This method iterates backwards through the history array.
  - If the current hash appears 2 times previously (total 3 occurrences), return true.
  - Must handle the "50-move rule" reset (repetition logic breaks if a pawn moves or capture occurs).

9.3 Auxiliary Hashes

- Pawn hash (pawn structure only)
- Material hash (piece counts)

9.4 Debug Hash

- 128-bit hash in debug builds
- Used only for assertion checks

10. FEN Handling

10.1 Parsing

- Must validate:
  - Exactly one king per side
  - Pawns not on rank 1 or 8
  - Valid castling configuration
  - Valid EP square
- Reject invalid FENs aggressively.

11. Invariants & Assertions

Engine must enforce invariants internally. **Examples:**

- `whitePieces & blackPieces == 0`
- Exactly one king per color
- Occupancy consistency
- Hash consistency after make/unmake

Assertions are not tests. They are guard rails.

12. Thread Safety

- Engine core must be re-entrant
- No global mutable state
- All tables are read-only after initialization
- **Enables:**
  - Lazy SMP
  - Per-thread Position copies
  - Shared TT

13. Performance Constraints

- No heap allocation in:
  - Move generation
  - `makeMove / unmakeMove`
- Use:
  - `_mm_popcnt_u64`
  - `_BitScanForward64`
- Precompute everything possible at startup
- Favor branchless logic where practical

14. Testing Strategy & Tooling

14.1 Visualization Tools (Mandatory)

Debugging 64-bit integers is impossible without visualization.

- **Requirement:** Implement `printBitboard(uint64_t bb)` immediately.
- It must print the board in an 8x8 grid (ranks 8->1, files A->H).
- Use `.` for empty squares and `X` (or 1) for set bits.
- This tool must be available in the global scope for easy debugging access.

14.2 Perft

- Must pass:
  - Initial position
  - KiwiPete
  - Edge-case positions
- Support:
  - Divide mode
  - Bulk counting

14.3 Internal Consistency Tests

- Hash reversibility
- Make/unmake symmetry
- FEN round-trip correctness

15. API Stability Contract

The following APIs are frozen once implemented:

- `setFromFEN()`
- `generatePseudoLegalMoves()`
- `generateLegalMoves()`