The Final Report for CSEE 4840 Embedded System Design

Chess Game

Spring 2025

Team members:

Pengfei Yan (py2324)

Hongchi Liu (hl3813)

Hooman Khaloo (hhk2123)

Contents:

1. Introduction

1.1. Problem Statement and Project Goal

Chess is one of the most enduring and popular board games worldwide. Implementing a digital version of chess was among the earliest uses of board games in the realm of computer science. Here, we plan to develop chess for FPGA users.

In this project, we aim to build an interactive chess system on the DE1-SoC FPGA board. The system allows a user to make moves via a 1024x768 monitor, while all processing related to chess logic, board rendering, and graphics is integrated between software and hardware.

1.2. Motivation and Importance

- **Embedded Systems Architecture Learning**: This project merges hardware design (on FPGA), bus interfaces (Avalon or memory-mapped IO), and high-level code (in C) for game logic.
- **FPGA-Based Graphics**: Rendering a real-time 1024×768 display via a VGA module in an FPGA environment demonstrates how limited FPGA resources can be leveraged for 2D graphics.

1.3. Scope and Key Modules

This project separates responsibilities between hardware and software. The software (HPS) handles all chess logic, input parsing, and move validation, while the hardware (FPGA) is responsible for rendering the board and pieces in real time. This clean separation simplifies debugging and ensures efficient use of FPGA resources.

Hardware:

- In FPGA kernel, the hardware will display 8×8 chessboard with 64×64 pixel squares, rendered in real-time at 1024×768 resolution.
- Display and movement of chess pieces using **Sprite** or shape data stored in on-chip ROM.
- Capability to update the board graphics after each move, with potential future enhancements for **smooth piece movement** (animations).

Software:

- **Chess logic** implemented in C software running on the ARM processor (HPS) that maintains and validates the game state.
- The universal chess interface (**UCI**) component handles communication between the chess logic and move history, enabling support for AI moves and structured recording of game progress.

2. System Block Diagram

2.1. System Block Diagram

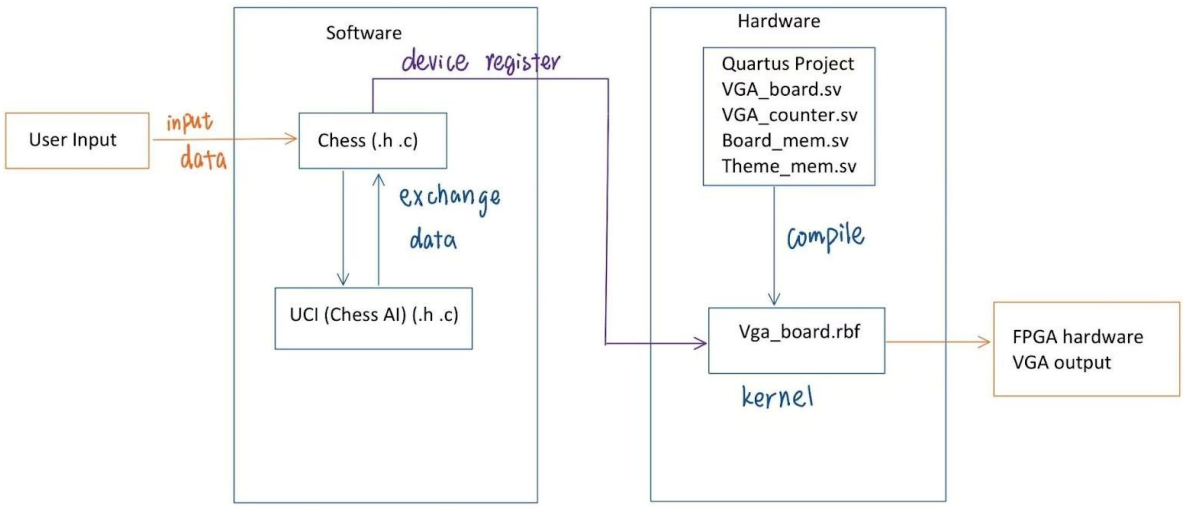


Figure 1. The overall system block diagram

2.2. Hardware Components Explanation

1. **Quartus Project and Verilog source file:** this part is modified from the sample code from lab3. We design the new structure for vga_board component, making it communicate with chess program in software. In addition, the VGA counter driver in Verilog is modified to hand the correct program resolutions 1024×768 @ 60Hz.
2. **Vga_board.sv**: receive 8x8 board from HPS and draw the box of the board and pieces.
3. **vga_counters.sv:** create VGA's h_count and v_count signal, configured for 1024x768@60Hz, 65MHz pixel clock. The h_count is [10:0] and the v_count is [9:0].
4. **board_mem.sv:** two write_address and one read address. We use two write addresses to draw two pieces at the same time. For example, if we move one piece, we need to write the initial position as empty and the final position as the piece.
5. **theme_mem.sv:** Stores 16 color themes, each with 5 RGB color sets for board, pieces, and background. Uses 4-bit theme ID to index into 120-bit theme data (5 colors × 3 channels × 8 bits). Enables customizable visuals by separating style data from rendering logic.
6. **Linux Kernel Configuration in rbf format**: after compiling the Quartus project, a rbf file will be uploaded to SD card on the FPGA board. This file will communicate the actual hardware and generate VGA output.
7. **Shape ROM (Hex Array in Verilog)**: Each piece's pattern is typically 16*16 pixels as a compressed variant.

2.3. Software Components Explanation

1. **Chess Logic in C:** the complete chess program including main menus, different game mode (PvP and PvE), and replay of a game. Runs on the ARM processor in the HPS (Hard Processor System). It also includes data structures for chess pieces, rules validation, and final checks (checkmate, draw, etc.), then notifies the hardware side accordingly.
2. **Universal chess interface:** The UCI module (uci.c, uci.h) manages communication between the chess logic and game state history. It records each move (e.g., "e2e4\0") in a global buffer for replay and analysis. In PvE mode, it facilitates AI integration by formatting and processing moves. While it doesn't directly interact with hardware, it structures data for the software to send to the FPGA via memory-mapped registers. UCI ensures modularity, separating move handling from game logic, and enables future extensions like AI engines or network play.

3. Algorithms

3.1. Chessboard Rendering Algorithm (in hardware)

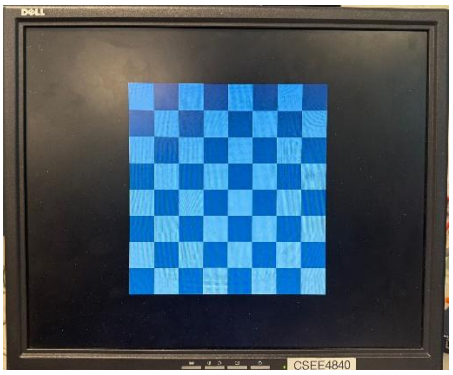


Figure 2. chessboard algorithm

1. **Obtain hcount, vcount**: The VGA Controller uses horizontal (hcount) and vertical (vcount) counters that indicate which pixel is currently being generated.
2. **Board or background**: Since each box is 64*64, the whole board on the center is 512*512 pixel. The range for the chessboard is (128, 640) for vcount, (256, 768) for hcount. Pixels outside the board are the background.
3. **Compute square index**:
    - Each square is 64×64 pixels, so x_box_no = (hcount-256) / 64, y_box_no = (vcount-128) / 64.
    - x_box_no and y_box_no range from 0 to 7, for an 8×8 board.
4. **Determine black or white square**:
    - x_parity = x_box_no % 2.
    - y_parity = y_box_no % 2.

- color = x_ parity XOR y_ parity→ If color = 0, the square is white; if 1, the square is black.

This lightweight algorithm relies on basic division and XOR operations, meaning no large texture is required to store the entire board.

3.2. Theme Memory (in hardware)

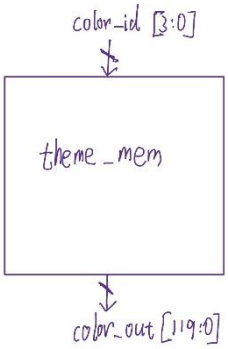


Figure 3. theme_mem

1. Theme and Color Storage: Each theme is 120 bits and contains 5 RGB color entries (8 bits × 3 channels) for background, black/white pieces, and black/white board colors.

2. Theme Selection: A 4-bit theme_id is used to select one of the 16 available themes stored in theme_mem.

3. Color Role Mapping: Each theme stores colors in a fixed order: black piece, white piece, black square, white square, background.

4. Color Extraction: The rendering module extracts appropriate RGB values from the selected theme based on square color and piece ownership.

Theme 1: Blue Tone. (Black, White, Navy Blue, Sky Blue, different Background)

Theme 2: Red Tone. (Black, White, Red, Bright Green, different Background)

Theme 3: Green Tone. (Black, White, Dark Green, White, different Background)

Theme 4: Grey Tone. (Black, White, Grey, White, different Background)

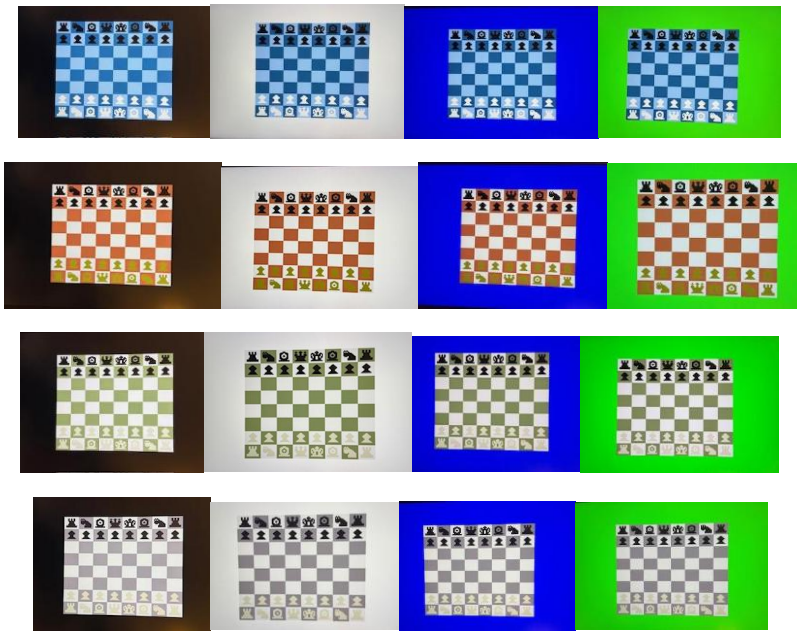


Figure 4. All color themes.

This modular color system enables flexible visual customization with minimal logic and memory overhead.

3.3. Theme Memory (in hardware)

1. Read Port: read_addr + data_out continue reading for VGA counter, streaming 4-bit values out in real time for display
2. Two independent Write Ports:
   a) Channel 1: addr1, data1
   b) Channel 2: addr2, data2
3. Write Modes:
   a) Single-cell write: when addr1 == addr2, only one write occurs (updates one square).
   b) Dual-cell write: when addr1 != addr2, both channels write simultaneously to two squares (e.g., for piece capture or swapping).

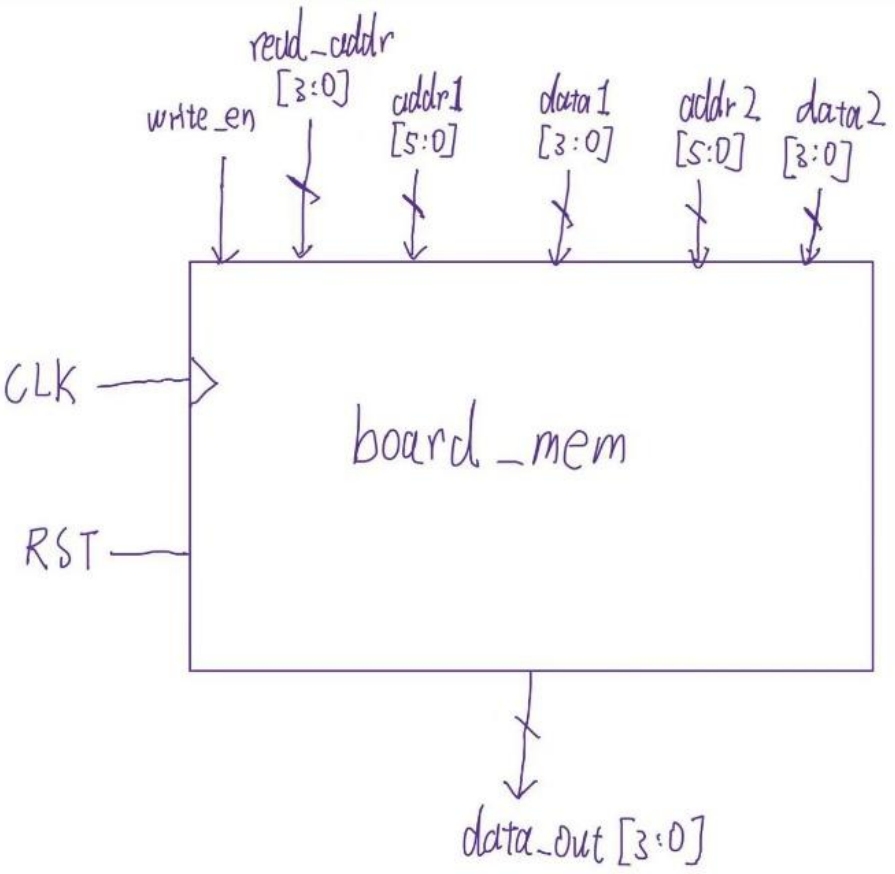


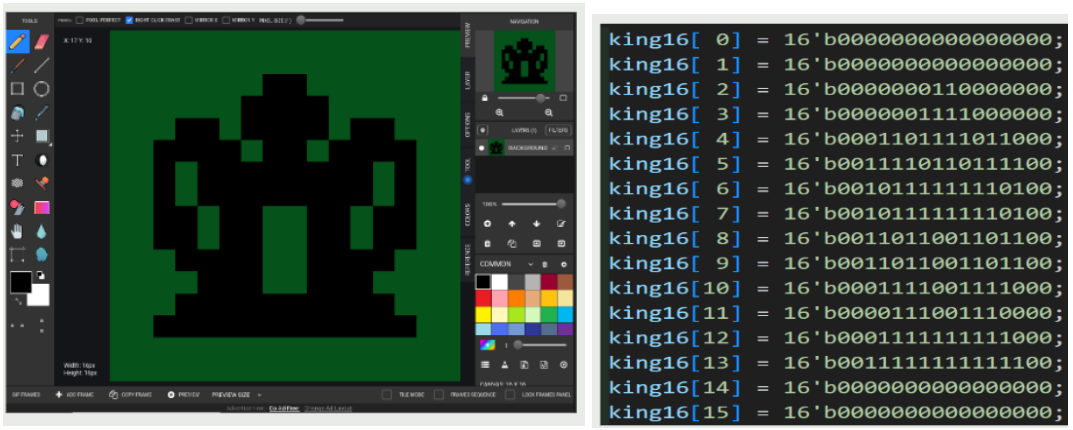
Figure 5.Board memory.

This dual-port memory design allows efficient and parallel updates to the board state, supporting smooth piece transitions and real-time rendering without disrupting VGA read operations.

3.4. Piece Rendering Algorithm (in hardware)

To display pieces, a layer on top of the board background must be used. The piece's pixels override or blend with the background squares:

1. **Retrieve piece location**: Each frame (or whenever updated), the software writes the piece's location (square x,y) into a shared hardware memory or register.
2. **Piece type lookup**: The type and color information will be stored in the board memory with 4 bits per box. The first bit represents the color, while the other three bits represent the type of pieces. For example, 4'b1001 presents a white pawn, since the first 1 means white and 001 means pawn. Here, the different write and read addresses make sure the VGA counter can always work while writing.
3. **Local pixel address in ROM**: If the pixel (hcount, vcount) lies within a piece's bounding box, compute local coordinates inside the piece graphic:
    - local_x = (hcount - 256 - (box_x * 64))
    - local_y = (vcount - 128 - (box_y * 64))
    - (box_x * 64, box_y * 64) represents the top-left corner of the board square.
4. **ROM data fetch**: The system checks (piece_type, color, local_y, local_x) in Shape ROM. If the stored bit is 1, that pixel belongs to the piece; if 0, the background is used.
5. **Drawing pieces:** we make the 64 pixels box divided by 4, there are 16 pixel in the box as we can draw it easily. Then we found a website called pixels drawing to draw the shape and transform it into 0 and 1 codes



```
king16[ 0] = 16'b0000000000000000;
king16[ 1] = 16'b0000000000000000;
king16[ 2] = 16'b0000000110000000;
king16[ 3] = 16'b0000001111000000;
king16[ 4] = 16'b0001101111011000;
king16[ 5] = 16'b0011110110111100;
king16[ 6] = 16'b0010111111110100;
king16[ 7] = 16'b0010111111110100;
king16[ 8] = 16'b0011011001101100;
king16[ 9] = 16'b0011011001101100;
king16[10] = 16'b0001111001111000;
king16[11] = 16'b0000111001110000;
king16[12] = 16'b0001111111111000;
king16[13] = 16'b0011111111111100;
king16[14] = 16'b0000000000000000;
king16[15] = 16'b0000000000000000;
```

Figure 6. Pixel drawing and shape array for king.

This layered rendering algorithm enables efficient and scalable piece drawing by leveraging compact ROM patterns and coordinate-based indexing, minimizing memory use while preserving visual clarity.

3.5. Chess Logic in C (in software)

1. **Data Structure**:
    - Piece types and colors are defined using enum.
    - Each piece contains a type and a color.
    - The board is represented as an 8×8 2D array of piece object.

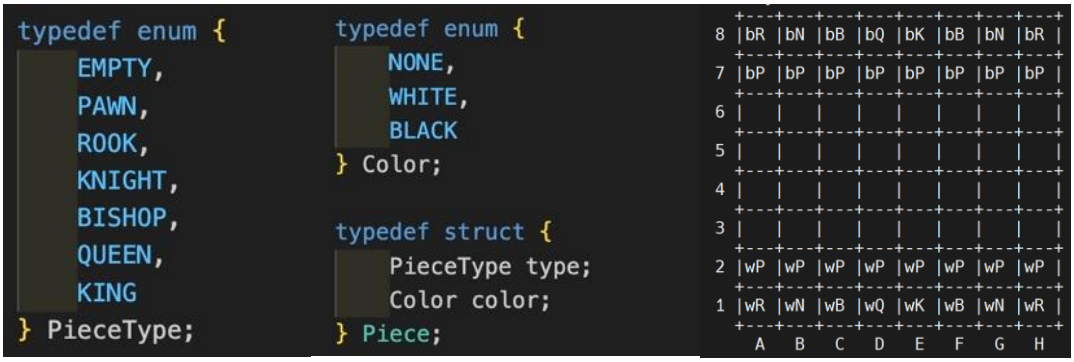


Figure 7: data structure and chess board

2. **Initial Setup**: White pieces occupy rows 1–2; black pieces occupy rows 7–8, following standard chess layout.
3. **Move Validation**:
    - Check bounds and ensure the selected piece exists.
    - Ensure the piece belongs to the current player.
    - Prevent self-capture and apply rules for each piece type (see appendix).
4. **Special Move**: Special movement: en passant, Pawn Promotion, and castling (explained in appendix) are supported through some global flag variables.
5. **Apply Changes & Notify FPGA**:
    - If a move is valid, data is encoded via the register map and sent to the FPGA for rendering.

The structured C logic ensures accurate rule enforcement and seamless coordination with hardware for real-time game updates.

3.6. Universal Chess Interface (UCI) in C

1. Stores all moves in a global history buffer (moves[2048]), each as a 5-character string (e.g., "e2e4").
2. Enables PvE mode by interfacing with AI logic for move generation and parsing.
3. Separates move handling from core logic, making replay, debugging, and future engine integration easier.

UCI provides a clean interface for move history and AI support, enhancing modularity and future extensibility.

4. Resource Budgets

4.1. Memory and Storage

1. **On-Chip Memory**

   The Cyclone V FPGA on the DE1-SoC provides roughly 10–100 KB of internal memory. A full 1024×768 framebuffer would require over 700 KB, which exceeds on-chip capacity. Instead, we employ an on-the-fly rendering approach using pixel coordinates and shape data, eliminating the need for a framebuffer.

   Piece graphics are stored as 16×16 binary patterns and scaled up in rendering. At 4 bits per pixel and 12 piece variants (6 types × 2 colors), the total ROM usage is under 48 KB, well within the FPGA's limits.

2. **External RAM**

   For advanced features like high-resolution themes or animations, external SDRAM or HPS DDR3 memory (hundreds of MB) could be utilized. Our current implementation relies entirely on internal memory, ensuring low latency and simplicity.

3. **Software Memory**

   The chess logic executes on the HPS ARM core, with access to large DDR3 memory. This is more than sufficient for handling game state, logic, and move history using structures like Piece board[8][8] and move[2048].

4.2. Bandwidth and Computational Constraints

- **VGA Pixel Rate**: At 1024×768 resolution and 60 Hz refresh, the system processes roughly 47 million pixel operations per second. Our 65–75 MHz pixel clock sustains this throughput, including blanking intervals, without issue.
- **Avalon Bus Bandwidth**: Each move involves updating just two 8-bit registers (start and end square info). This results in minimal traffic and no bottleneck across the memory-mapped interface.
- **Software Load:** Each move involves checking 20–30 possible positions. The HPS easily handles this logic with negligible CPU load, keeping real-time interactivity responsive.

Thanks to careful partitioning and efficient rendering techniques, our design remains well within the DE1-SoC's computer and memory budget, leaving headroom for future enhancements.

5. Hardware/Software Interface

This section describes how the software and hardware components interact through shared memory-mapped registers, allowing efficient communication without requiring the FPGA to interpret game logic directly.

5.1. Board Memory Registers

We implemented a 64-entry register block to represent the chessboard state, where each address corresponds to one of the 8×8 squares:

- Each address stores a 4-bit value:
  - Bits [3]: Piece color (0 = Black, 1 = White)
  - Bit [2:0]: Piece type (e.g., Pawn = 001, Rook = 010, etc.)
- The 6-bit address maps directly to board coordinates:
  - Bits [5:3]: x position (0–7)
  - Bits [2:0]: y position (0–7)

5.2 Write Sequence (from Software)

After each valid move:

- The software updates its internal 8×8 Piece array.
- It transmits data for each movement to the register map.
- Verilog uses write_enable to update the board memory.

This approach allows the FPGA to refresh the board state without needing to interpret game logic.

addr	writedata[7]	writedata[6]	writedata[5]	writedata[4]	writedata[3]	writedata[2]	writedata[1]	writedata[0]
0	-	-	-	-	color id			
1	Start X			End X			Start Y[2:1]	
2	Start Y[0]	End Y			Color and Type			
3	Done Movement							

Figure 8: Register Map

To connect to this register map, these two lines pack move information into two 8-bit registers:

```
uint8_t reg1 = ((start_x & 0x7) << 5) | ((end_x & 0x7) << 2) | ((start_y >> 1) & 0x3);

uint8_t reg2 = ((start_y & 0x1) << 7) | ((end_y & 0x7) << 4) | (e_p & 0xF);
```

6. Project Files and Compilation Instructions

6.1. Hardware Files and Compilation

Quartus source files (from lab3): soc_system.qsys, soc_system_top.sv, soc_system.srf,

Makefile, soc_system_board_info.xml, soc_system.tcl, vga_ball_hw.tcl

vga_board.sv: from lab 3, draw the chess board

vga_counters.sv: from lab 3, change output to 1024x768 @ 60Hz @ 65MHz

board_mem.sv: save the 8*8 board, each box is 4bits. 1b color + 3b type.

theme_mem.sv: store color themes. for each theme, 5 color * 3rgb value * 8bits each = 120 bits.

To compile, use:

cd hw

qsys-edit soc_system

make quartus

make rbf

6.2. Software Files and Compilation

Makefile: used to compile

chess.h/.c: store the main function of chess including menu, rule, and interaction with HW.

uci.c: help to communicate with AI server for PvE mode.

To run, use:

make

./chess

7. Appendix (Chess Glossary)

7.1. Piece Types

Source of Pictures: https://commons.wikimedia.org/wiki/Category:SVG_chess_pieces

1. Pawn

Movement: Moves forward 1 square. From its starting position, it can move forward 2 squares.

Capturing: Diagonally forward 1 square.

2. Rook

Movement and Capturing: Any number of squares horizontally or vertically.

3. Knight

Movement and Capturing: In an L-shape: 2 squares in one direction, then 1 square perpendicular.

4. Bishop

Movement and Capturing: Any number of squares diagonally.

5. Queen

Movement and Capturing: Any number of squares vertically, horizontally, or diagonally. (Rook or Bishop)

6. King

Movement and Capturing: 1 square in any direction.

7.2. Special Rules

1. En Passant

This rule allows a pawn to capture an opponent’s pawn that has just moved two squares forward from its starting position. The capture is made as if the pawn had only moved one square. It must be done immediately on the next move or the opportunity is lost.

2. Pawn Promotion

When a pawn reaches the farthest row on the opponent's side (rank 8 for white, rank 1 for black), it is promoted. In our system, it is automatically promoted to a queen. This occurs instantly after the pawn moves to the last rank.

3. Castling

Castling is a special move involving the king and either rook. The king moves two squares toward a rook, and the rook jumps over the king to the square next to it. Castling is only allowed if neither piece has moved, no pieces are between them, and the king is not in, through, or moving into check.