The Design Document 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 the 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 **Wacom DTF-510** graphics tablet, while all processing related to chess logic, board rendering, and graphics are 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.
- **User Interaction via a Graphics Tablet**: The project uses a Wacom tablet with touch screen. It allows exploration of **I/O driver** challenges in embedded systems.

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.
- In C software, integrate the driver of a **Wacom DTF-510** tablet for user input (touch/pen coordinates). And keyboard for secondary input.

2. System Block Diagram

2.1. System Block Diagram

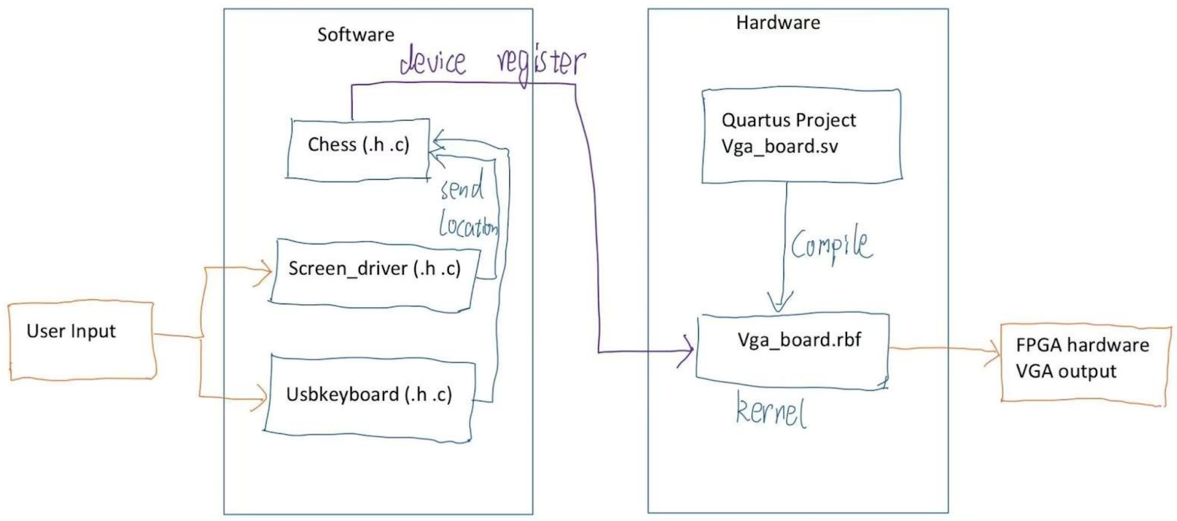


Figure 1. The overall system block diagram

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. **Linux Kernel 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.
3. **Shape ROM (Hex Array in Verilog)**
    1. Stores the graphical data for each chess piece (Pawn, Rook, Knight, Bishop, Queen, King) with 4-bit memory with 6-bit address. The 4-bits data represent the color and shape of one piece. The 6-bit address represent 64 chess boxes.
    2. Each piece's pattern is typically 64×64 pixels or 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. **Touch Screen Driver:** the tablet driver collects user inputs and convert them into the xy coordinates on the chess board. Then, send these coordinates to chess program. The specific input format will be explained in the next part.

3. **USB Keyboard Driver:** this part is used as the secondary input method in case of touch screen is not working or for developing purposes. The source code is retrieved from lab 2 sample code.

3. Algorithms

3.1. Chessboard Rendering Algorithm (in hardware)

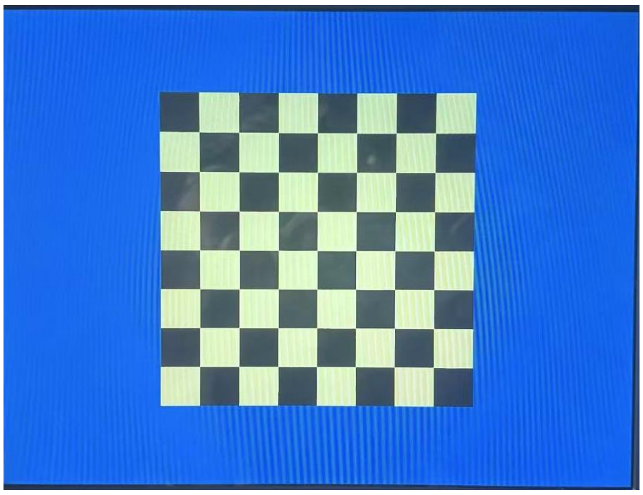


Figure 2. chessboard algorithm

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. 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 could always work while writing.

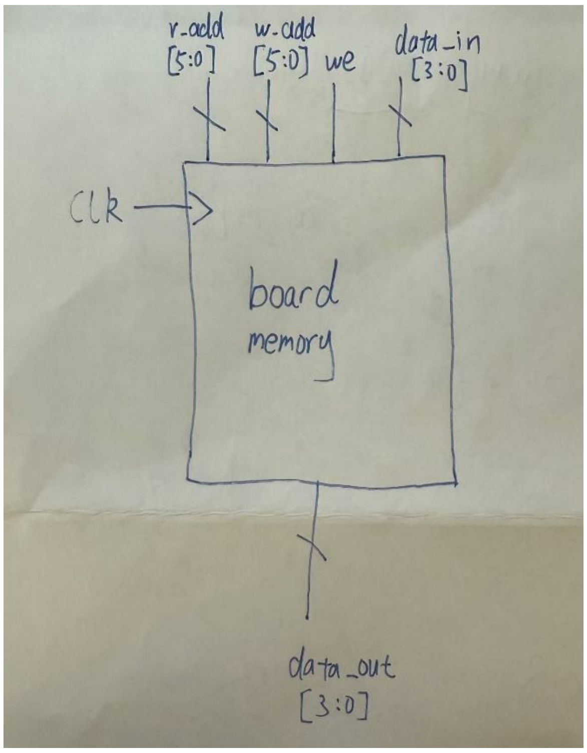
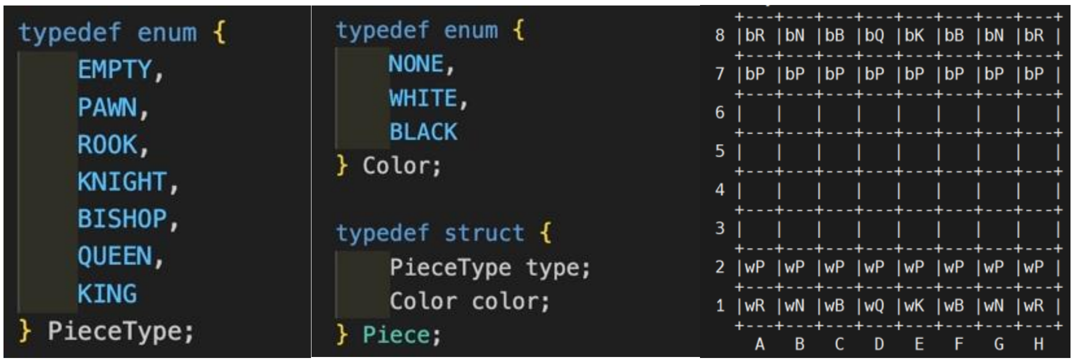


Figure 3. board memory

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.

3.3. Chess Logic in C (in software)

1. **Data Structure**:
   - We use enum to store different types of pieces.
   - We use enum for colors, which is also important in chess game.
   - We define the pieces with type and color.
   - A chess board is 8*8 2-dimentional arrays of Piece.



```c
typedef enum {
    EMPTY,
    PAWN,
    ROOK,
    KNIGHT,
    BISHOP,
    QUEEN,
    KING
} PieceType;
```

```c
typedef enum {
    NONE,
    WHITE,
    BLACK
} Color;

typedef struct {
    PieceType type;
    Color color;
} Piece;
```

```
  +---+---+---+---+---+---+---+---+
8 |bR |bN |bB |bQ |bK |bB |bN |bR |
  +---+---+---+---+---+---+---+---+
7 |bP |bP |bP |bP |bP |bP |bP |bP |
  +---+---+---+---+---+---+---+---+
6 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
5 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
4 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
3 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
2 |wP |wP |wP |wP |wP |wP |wP |wP |
  +---+---+---+---+---+---+---+---+
1 |wR |wN |wB |wQ |wK |wB |wN |wR |
  +---+---+---+---+---+---+---+---+
    A   B   C   D   E   F   G   H
```

Figure 3: data structure and chess board

2. **Initial Setup**:
   - Rows 1 and 2 for white pieces, rows 7 and 8 for black pieces, following standard chess notation.
3. **Coordinates Input**:
   - Moves are entered as 4-character strings (e.g., "A2A4"), representing start and end squares refer to figure 3. The program parses these using:
     - decode_x() to convert file letters ('A'-'H') to column indices (0–7).
     - 8 - (rank) to convert rank numbers ('1'-'8') to row indices.
     - Invalid inputs (wrong length or out-of-bounds) prompt an error message. This format supports both keyboard and touchscreen input, with touchscreen coordinates converted into board indices.
4. **Validate Move**:
   - Checking bounds and ensuring a piece exists.
   - Verifying the piece matches the player's color.
   - Rejecting self-capture.
   - Applying piece-specific rules (see appendix for specific rules).
5. **Special Move**: Special movement: en passant, Pawn Promotion, and castling (explained in appendix) are supported through some global flag variables.

6. **Apply Changes & Notify FPGA**:
    - If the move is valid, the software updates the 8×8 array accordingly (clears the origin square, sets the destination square to the piece code).
    - This new arrangement is written to shared memory/FPGA registers.
    - Special rules like castling, en passant, or promotion can be added for advanced logic. Checkmate or check states can also be flagged.
    - The software will send the movement to rbf kernel. Then, the hardware will draw the new board.

4. Resource Budgets

4.1. Memory and Storage

1. **On-Chip Memory**
    - DE1-SoC's Cyclone V FPGA typically provides 10-100 kB of on-chip memory.
    - Storing a full 1024×768 frame buffer in on-chip memory is not feasible, so we use an **on-the-fly rendering** approach, removing the need for a large buffer.
    - For **piece graphics** (shapes), the memory is generally sufficient: each piece is 64×64 pixels. Even at 4 bits per pixel, that's 4 KB per piece. With 6 types in 2 colors (12 total), that's about 48 KB. Still feasible for internal ROM.
2. **External RAM**
    - If advanced features like high-quality backgrounds or advanced animation are desired, data may be loaded from external SDRAM or the HPS's main DDR3 memory.
    - This project's design mainly relies on internal memory for storing shape patterns, requiring no large, dedicated external frame buffer.
3. **Software Memory**
    - The chess logic runs on the HPS (ARM), which has access to larger external DDR3 (hundreds of MB). This is more than enough for a C program that handles the game logic.

4.2. Bandwidth and Computational Constraints

- **VGA Pixel Rate**: For 1024×768 at 60 Hz, ~60 × 1024 × 768 ≈ 47 million pixel operations per second (strictly counting visible pixels; blanking intervals add more). With a pixel clock around 65–75 MHz, the FPGA can generate these signals.
- **Avalon Bus Bandwidth**: Only 2*8 bits are updated per move, so the bus traffic is minimal and not a bottleneck.
- **Chess Logic Computation**: The CPU overhead from checking 20–30 possible moves each turn is small and does not strain the HPS.

Overall, the design fits comfortably within the DE1-SoC's resource limits.

5. Hardware/Software Interface

We are still developing this part. These are some initial attempts.

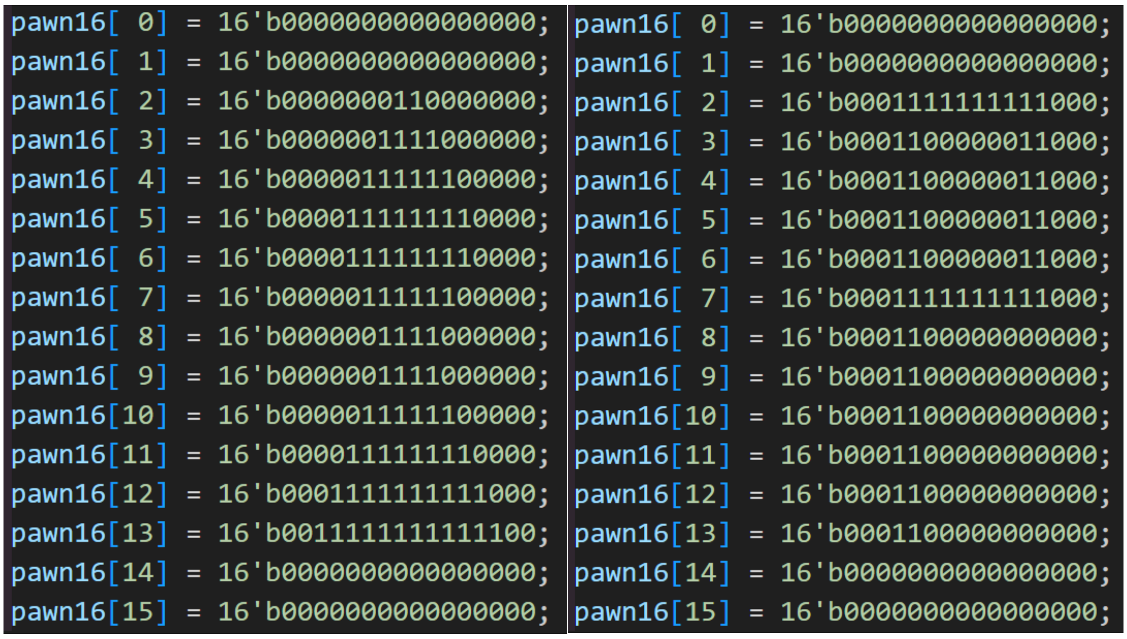
5.1. Registers (Board Memory)

We designed board memory with registers (in Figure 3).

- Each address stores a 4-bit value:
  - Bits [3]: Piece color (0 = Black, 1 = White)
  - Bit [2:0]: Piece type (e.g., Pawn = 1, Rook = 2, etc.)
- The address is 6 bits representing 64 chess boxes. The first 3 bits are x value (0 to 7). The last 3 bits are y value (0 to 7). A coordinate on board could be easily converted to binary to work with the address.
- Different write and read addresses make sure VGA counter could always work while writing.

5.2 Theme: Shape and Color

For each box we have 64 pixels. We use 16 lines to draw the shape and zoom it 4 times larger to fill the square. We are trying to create two different style pieces. One is a classical piece in shape, the other one is a new style as letters. The two figures below show these two styles. theme_id has 8 bits. The high nibble ([7:4]) is defined as shape_id and the low nibble ([3:0]) is defined as color_id. Here, we can have maximum 16 of different shapes and 16 of different colors (256 different themes in total).



```
pawn16[ 0] = 16'b0000000000000000;
pawn16[ 1] = 16'b0000000000000000;
pawn16[ 2] = 16'b0000000110000000;
pawn16[ 3] = 16'b0000001111000000;
pawn16[ 4] = 16'b0000011111100000;
pawn16[ 5] = 16'b0000111111110000;
pawn16[ 6] = 16'b0000111111110000;
pawn16[ 7] = 16'b0000011111100000;
pawn16[ 8] = 16'b0000001111000000;
pawn16[ 9] = 16'b0000001111000000;
pawn16[10] = 16'b0000011111100000;
pawn16[11] = 16'b0000111111110000;
pawn16[12] = 16'b0001111111111000;
pawn16[13] = 16'b0011111111111100;
pawn16[14] = 16'b0000000000000000;
pawn16[15] = 16'b0000000000000000;
```

```
pawn16[ 0] = 16'b0000000000000000;
pawn16[ 1] = 16'b0000000000000000;
pawn16[ 2] = 16'b0001111111111000;
pawn16[ 3] = 16'b0001100000011000;
pawn16[ 4] = 16'b0001100000011000;
pawn16[ 5] = 16'b0001100000011000;
pawn16[ 6] = 16'b0001100000011000;
pawn16[ 7] = 16'b0001111111111000;
pawn16[ 8] = 16'b0001100000000000;
pawn16[ 9] = 16'b0001100000000000;
pawn16[10] = 16'b0001100000000000;
pawn16[11] = 16'b0001100000000000;
pawn16[12] = 16'b0001100000000000;
pawn16[13] = 16'b0001100000000000;
pawn16[14] = 16'b0000000000000000;
pawn16[15] = 16'b0000000000000000;
```

Figure 4: different piece shapes

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 or theme 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.

Address 0 is used to setup the color/shape of the game.

Address 1 and 2 are used to transmit a movement, including start/end coordinates (box #0 to 7, which are 12 bits total), type and color (4 bits).

Register Map:

addr	Writedata[7:0]
0	theme_id[7:0]
1	start_x: [7:5], end_x: [4:2], start_y [2:1]: [1:0]
2	start_y[0]: [7] end_y: [6:4] type: [3:1] color: [0]

5.3 FPGA Decoder Logic (in Hardware)

The hardware (VGA module or sprite generator) continuously reads the 64 registers:

- For each screen pixel, it checks which square it falls into the board
- Then, it reads that square's 4-bit register value to decide:
  - Which piece sprite to display (based on piece type)
  - What color (white/black) to render

6. Appendix (Chess Glossary)

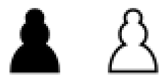
6.1. Piece Types

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

1. Pawn

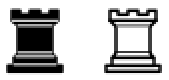
Movement: Moves forward 1 square. From its starting position, it can move forward 2 squares.

Capturing: Diagonally forward 1 square.



2. Rook

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

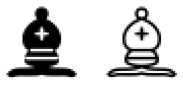


3. Knight

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

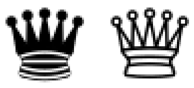
4. Bishop

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.

♚ ♔

6.2. 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.