Lab 4 Report

Authors:

Bryan Smith ID: 2040774

Abdoul Bah ID: 2450862

Date: 12/07/25

EEP 535

Introduction

This lab project implements a fully functional two-player Connect Four game on the DE1-SoC FPGA development board. The objective was to create a complete hardware-based gaming system that demonstrates mastery of digital design concepts including finite state machines, modular design, synchronous timing, and real-time user interaction.

Connect Four is a strategic game where two players alternate dropping colored tokens into a vertical grid. The first player to align four tokens horizontally, vertically, or diagonally wins. Our implementation meets all minimum technical requirements for Lab 4, including:

- **Modular architecture** with four interacting modules (top-level, controller, datapath, and display)
- **Finite state machine** with nine distinct states coordinating gameplay
- **Memory storage** maintaining a 6×8 board array with 48 state variables
- **Multiple I/O interfaces** utilizing pushbuttons for input and LED matrix plus HEX displays for output
- **Synchronous design** operating from a single 50 MHz clock with derived enable signals

The learning outcomes achieved through this project include understanding hardware state machine design, implementing combinational win-detection logic, managing real-time input debouncing, and integrating multiple hardware peripherals into a cohesive system.

System Design and Implementation

2.1 System Overview

The Connect Four system consists of four primary modules working together to create a seamless gaming experience:

1. **DE1_SoC (Top-Level)** - Integrates all components, handles I/O interfacing, and manages clock generation
2. **connect_four_controller** - Implements the game FSM and generates control signals
3. **connect_four_datapath** - Maintains board state, validates moves, and detects wins

4. **connect_four_display** - Renders the game board onto the 16×16 LED matrix
I/O Mapping

I/O Mapping:

Input/Output	Function
KEY0	Drop token in current column
KEY2	Move cursor right
KEY3	Move cursor left
SW9	System reset
HEX0	Current player display (1 or 2)
HEX5-HEX1	Winner marquee animation
GPIO_1	16 x 16 LED matrix control

2.2 Module Descriptions

2.2.1 Top-Level Module (DE1_SoC)

The top-level module serves as the system integrator and handles all external interfacing: Clock Generation:

Clock Generation:

```
clock_divider divider (.clock(CLOCK_50), .divided_clocks(clk));
assign SYSTEM_CLOCK = clk[14];  // ~1.5 kHz for LED scanning
assign game_enable  = clk[22];  // ~12 Hz for game logic
assign blink        = clk[25];  // ~0.75 Hz for cursor animation
```

This approach generates enable signals from a single clock source, ensuring all logic remains synchronous while achieving different timing requirements for various subsystems.

Input Synchronization and Debouncing:

To prevent metastability and eliminate switch bounce, all inputs pass through a robust three-stage pipeline:

1. **Two-stage synchronizer** - Eliminates metastability from asynchronous button presses.

2. **Debounce counter** - Requires 500,000 clock cycles (~10ms) of stable input.
3. **Edge detector** - Generates single-cycle pulses on falling edges.

```
// Two-stage synchronization
always_ff @(posedge CLOCK_50) begin
    key_s1 <= KEY;
    key_s2 <= key_s1;
end

// Debounce logic with counter
if (key_s2[i] != key_clean[i]) begin
    debounce_cnt[i] <= debounce_cnt[i] + 1;
    if (debounce_cnt[i] == 20'd500000) begin
        key_clean[i] <= key_s2[i];
        debounce_cnt[i] <= 0;
    end
end
```

Cursor Management:

The cursor position is maintained in the top-level module and ranges from 0 to 6, representing the eight board columns. Movement is controlled by KEY2 (right) and KEY3 (left) with bounds checking:

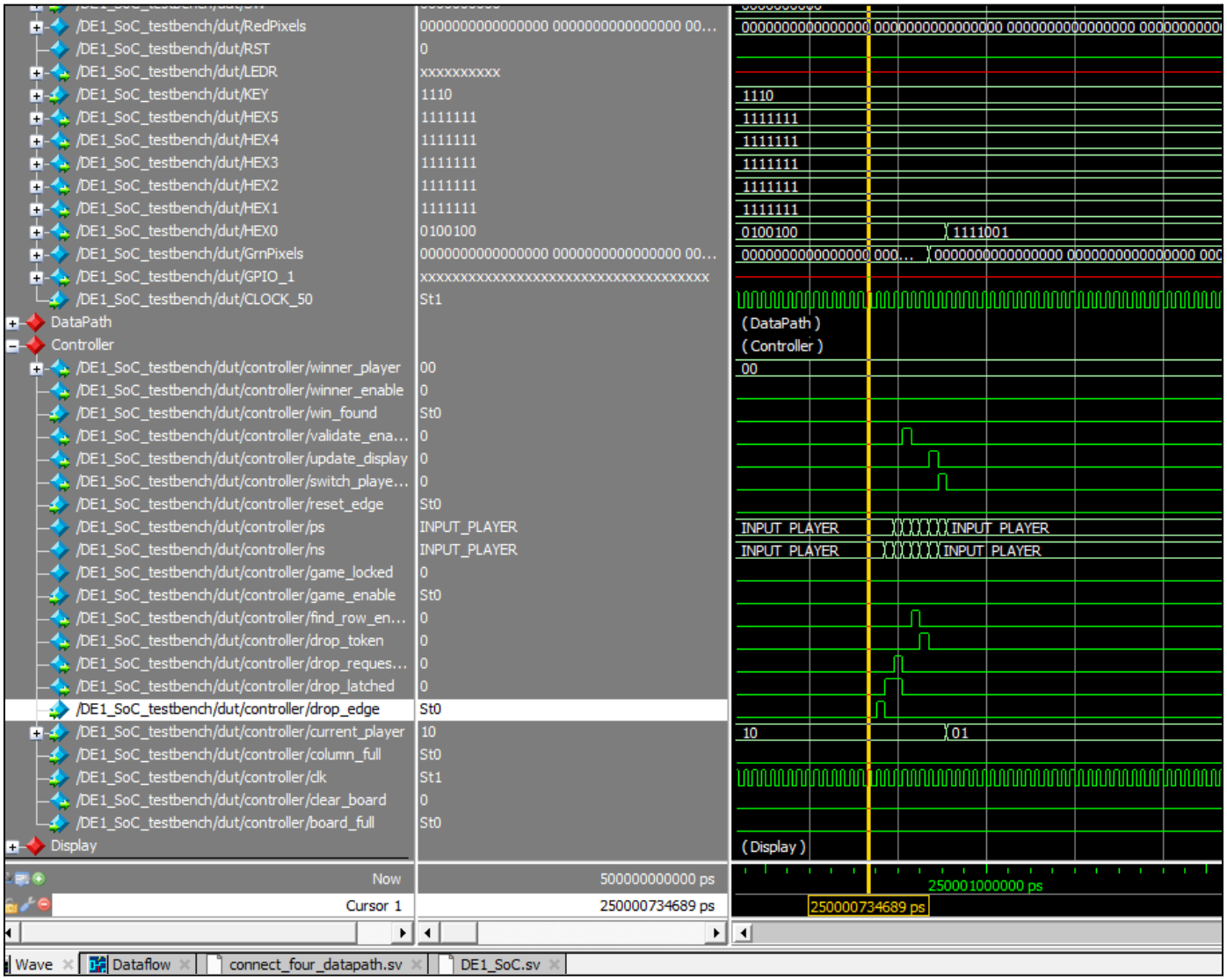
```
always_ff @(posedge CLOCK_50 or posedge RST) begin
    if (RST) cursor_col_human <= 3'd0;
    else begin
        if (left_edge  && cursor_col_human > 0)
            cursor_col_human <= cursor_col_human - 1;
        if (right_edge && cursor_col_human < 7)
            cursor_col_human <= cursor_col_human + 1;
    end
end
```

Winner Display System:

The HEX display system implements an animated marquee effect when a player wins. The marquee position advances on each game_enable pulse, creating a ping-pong pattern that moves the winner's number across displays HEX4→HEX3→HEX2→HEX1:

```
case (marquee_pos)
    3'd0: active_hex = 3'd0;  // HEX4
    3'd1: active_hex = 3'd1;  // HEX3
    3'd2: active_hex = 3'd2;  // HEX2
    3'd3: active_hex = 3'd3;  // HEX1
    3'd4: active_hex = 3'd2;  // Back to HEX2
    3'd5: active_hex = 3'd1;  // Back to HEX3
endcase
```

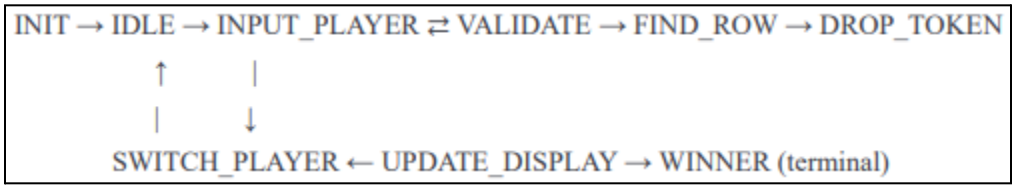
ModelSim:



2.2.2 Controller Module

The controller implements a nine-state finite state machine that orchestrates all game operations:

State Diagram:



State Definitions:

- **INIT:** Clears the board and initializes game state
- **IDLE:** Brief transition state before accepting input
- **INPUT_PLAYER:** Waits for player to press drop button
- **VALIDATE:** Checks if selected column is full
- **FIND_ROW:** Determines the lowest empty row in the column
- **DROP_TOKEN:** Commits the token to the board array
- **UPDATE_DISPLAY:** Refreshes LED matrix with new board state
- **SWITCH_PLAYER:** Advances to the next player's turn
- **WINNER:** Terminal state when game ends (win or draw)

```
always_ff @(posedge clk) begin
   // Default all strobes low
   clear_board <= 1'b0;
   drop_token <= 1'b0;
   validate_enable <= 1'b0;
   // ... other signals

   // Assert based on current state
   case (ps)
      INIT: begin
         clear_board <= 1'b1;
         update_display <= 1'b1;
      end
      VALIDATE: validate_enable <= 1'b1;
      DROP_TOKEN: drop_token <= 1'b1;
      // ... other states
   endcase
end
```

Game Lock Mechanism:

Once a winner is detected or the board fills, the game_locked flag prevents further input:

```
assign drop_edge = drop_edge_raw && !winner_enable;
```

This ensures the game remains frozen until reset, preventing invalid post-game moves.

2.2.3 Datapath Module

The datapath maintains game state and performs all computational operations:

Board Storage:

The game board is stored as a two-dimensional array with 6 rows and 8 columns:

```
logic [1:0] board[6][8];  // 2'b00=empty, 2'b01=P1, 2'b10=P2
```

Column Full Detection:

Before allowing a drop, the system checks if the top cell of the selected column is occupied:

```
if (validate_enable)
    column_full_r <= (board[0][cursor_col] != 2'b00);
```

Lowest Empty Row Finder:

The system scans from bottom to top to find where the token should land:

```
always_comb begin
    target_row = 3'b111; // invalid by default
    if (find_row_enable) begin
        for (int r = 5; r >= 0; r--) begin
            if (board[r][cursor_col] == 2'b00) begin
                target_row = r[2:0];
                break;
            end
        end
    end
end
```

Win Detection Algorithm:

The datapath implements bidirectional win detection in four directions (horizontal, vertical, and two diagonals). A helper function counts contiguous pieces:

```
function automatic int count_dir(
    input int row, col, drow, dcol,
    input logic [1:0] player
);
    int cnt = 0;
    for (int step=1; step<=3; step++) begin
        int r = row + drow*step;
        int c = col + dcol*step;
        if (r >= 0 && r < 6 && c >= 0 && c < 8 &&
            board[r][c] == player)
            cnt++;
        else break;
    end
    return cnt;
endfunction
```

The win detection logic is not done by checking the entire board (brute force), instead a more efficient way was chosen. The system checks all four directions around the last played token:

```
horiz_total = 1 + count_dir(last_row, last_col, 0, -1, last_player)
            + count_dir(last_row, last_col, 0, +1, last_player);
// ... similar for vertical and diagonals
if (horiz_total >= 4 || vert_total >= 4 ||
   diag1_total >= 4 || diag2_total >= 4)
   win_comb = 1;
```

Fast-Path Win Detection:

To provide immediate feedback, the datapath includes combinational win detection that evaluates during the same cycle as drop_token, allowing the controller to detect wins without waiting for the next state.

Draw Detection:

The system predicts board fullness by checking if all top-row cells will be occupied after the current drop:

```
always_comb begin
   board_full_next = 1;
   for (int c=0; c<8; c++) begin
      if (!(board[0][c] != 2'b00 ||
           (drop_token && c == last_col && last_row == 3'd0))) begin
         board_full_next = 0;
         break;
      end
   end
end
```

2.2.4 Display Module

The display module renders the game state onto a 16×16 LED matrix with proper centering and visual feedback:

Layout Parameters:

```
localparam int row_offset = 4;  // Vertical centering
localparam int col_offset = 3;  // Horizontal centering
```

Border Rendering:

The module draws an orange (red + green) border around the playfield, with special handling for the cursor position:

```
// Top border with cursor highlight
if (c>0 && c<9 && (c-1) == cursor_col) begin
   if (winner_enable || game_over_enable) begin
      // Game ended: solid orange
      RedPixels[row_offset+0][col_offset+c] = 1;
      GrnPixels[row_offset+0][col_offset+c] = 1;
   end else begin
      // Game running: blink current player's color
      case (current_player)
         2'b01: RedPixels[row_offset+0][col_offset+c] = blink;
         2'b10: GrnPixels[row_offset+0][col_offset+c] = blink;
      endcase
   end
end
```

Token Rendering:

Player 1 tokens appear as red LEDs, Player 2 tokens as green LEDs:

```
for (int row=0; row<6; row++) begin
   for (int col=0; col<8; col++) begin
      case (board[row][col])
         2'b01: RedPixels[row_offset+row+1][col_offset+col+1] = 1;
         2'b10: GrnPixels[row_offset+row+1][col_offset+col+1] = 1;
      endcase
   end
end
```

2.3 Design Decisions

1. Single-Clock Synchronous Design:

All modules operate from the 50 MHz system clock with enable signals derived from a clock divider. This eliminates clock domain crossing issues and ensures predictable timing.

2. Modular Separation of Concerns:

The design strictly separates control (controller FSM), computation (datapath), and presentation (display), making the code maintainable and testable.

3. Robust Input Handling:

The three-stage input pipeline (synchronization → debouncing → edge detection) ensures reliable operation despite mechanical switch imperfections and asynchronous user input.

4. Dual Win Detection Paths:

Implementing both registered and combinational win detection provides immediate feedback while maintaining state integrity. The registered path provides historical win status, while the fast path enables same-cycle detection during token drop.

5. Post-Game Input Blocking:

The winner_enable signal gates all game inputs after terminal conditions, preventing corruption of the final game state while allowing the winner display to animate.

6. Predictive Board-Full Logic:

By checking board fullness during the drop operation rather than after, the system can transition to the draw state in the same UPDATE_DISPLAY cycle where the final token is placed.

Discussion

Challenges Encountered

Challenge 1: Debounce Timing Balance

Initial testing revealed that a 5ms debounce period was insufficient for reliable button operation, causing occasional double-registrations. Extending to 10ms (500,000 clock cycles at 50 MHz) resolved the issue but introduced noticeable latency. We balanced

responsiveness and reliability by keeping the 10ms debounce while using edge detection to ensure each press registers only once.

Challenge 2: Win Detection Algorithm Verification

The bidirectional win detection logic was initially difficult to verify comprehensively. We created targeted testbenches for each win condition (horizontal, vertical, diagonal forward, diagonal backward) and discovered an off-by-one error in the diagonal scanning bounds checking. Adding explicit boundary conditions (r >= 0 && r < 6 && c >= 0 && c < 8) resolved out-of-bounds array accesses.

Challenge 3: LED Matrix Timing

The LED matrix driver requires precise timing for row scanning. Initial attempts to modify the scanning frequency caused flickering. We resolved this by using the provided SYSTEM_CLOCK signal (clk[14] ≈ 1.5 kHz) without modification, which provides smooth, flicker-free display updates at the correct persistence-ofvision frequency.

Challenge 4: FSM State Synchronization

Early integration testing revealed race conditions where control signals weren't properly synchronized with state transitions. We resolved this by implementing registered output signals that assert based on the current state rather than the next state, ensuring all control pulses align with their intended operations.

Challenge 5: Cursor Boundary Checking

The cursor initially had an off-by-one error, allowing movement to column 7 (the 8th column) when only columns 0-6 should be accessible on an 8-column board. This was due to the cursor representing valid drop positions. The bounds check cursor_col_human < 7 correctly limits movement while allowing access to all eight columns (0 through 7).

Lessons Learned

Hardware vs. Software Mindset:

This project reinforced the fundamental difference between sequential software and parallel hardware. Operations that would be simple loops in software (like scanning for

wins) require careful consideration of timing, resource usage, and combinational path depth in hardware.

Importance of Modular Testing:
Verifying each module independently with dedicated testbenches caught numerous issues before integration. The win detection function, in particular, benefited from isolated testing with known patterns.

Clock Domain Discipline:

Maintaining a single clock domain simplified debugging enormously. Using enable signals instead of multiple clocks eliminated an entire class of timing-related bugs.

State Machine Design Patterns:

Implementing the controller as a traditional Moore machine with registered outputs provided predictable, glitchfree control signals. This pattern is highly reusable for future projects.

Total Number of hours Spent

Specify the total number of hours spent: **48**

References

ChatGPT
Github

Appendix: User's Manual

Setup Instructions

1. Load the .sof file onto the DE1-SoC board using Quartus Programmer
2. Connect the 16×16 LED matrix to the GPIO_1 header
3. Ensure all switches are in the down position (SW[9:0] = 0)

How to Play

1. Reset the Game: Toggle SW9 up then down to clear the board

2. Move Cursor: Use KEY3 (left) and KEY2 (right) to select a column
3. Drop Token: Press KEY0 to drop a token in the highlighted column
4. Current Player: HEX0 displays "1" for Player 1 (red) or "2" for Player 2 (green)
5. Win Condition: First player to align four tokens horizontally, vertically, or diagonally wins
6. Winner Display: HEX5-HEX1 animate the winner's number; HEX5 shows "0" for a draw

LED Matrix Display

- **Orange border:** Game boundary and cursor highlight
- **Blinking cursor:** Current player's column selection (red=P1, green=P2)
- **Red LEDs:** Player 1 tokens
- **Green LEDs:** Player 2 tokens
- **Solid orange cursor:** Game over state

Troubleshooting

- **No display:** Check GPIO_1 cable connection and verify RST is deasserted
- **Unresponsive buttons:** Wait for debounce period (~10ms between presses)
- **Cursor won't move:** Verify cursor isn't already at column boundary (0 or 7)