VGA Sudoku Project Report (Draft 3)

FPGA Implementation for Basys3 Board

Project Documentation and Technical Report

CSCEA 342 - Digital Circuits

University of Alaska Anchorage

Engineers:

Gwendolyn Beecher

Constantine Hinds

December 2025

This report was created with the help of Claude AI (Anthropic), based on project files and drafts provided by the authors and the project presentation.

Table of Contents

1. Project Overview

This project implements a fully playable Sudoku game on the Basys3 FPGA development board with VGA display output. The system features a 640×480 VGA display showing a 9x9 Sudoku grid, cursor-based navigation, number entry with preview, puzzle selection, real-time correctness checking, and win/lose state detection.

1.1 Key Features

- **VGA Display:** 640×480 @ 60Hz, with a centered 360×360-pixel Sudoku grid.
- **Puzzle Selection:** Three selectable puzzles with on-screen game selector UI (left panel).
- **Input System:** Two-mode input system: **MOVE** mode for cursor navigation, **NUMBER** mode for digit entry.
- **Visual Feedback:** Flashing cursor with number preview in NUMBER mode.
- **Seven-Segment Display:** Shows cursor position (X, Y), edit indicator, and current/selected value.
- **Game Logic:** Fixed cell protection (pre-filled puzzle values cannot be modified).
- **Validation:** Real-time correctness checking via SW2 (cells turn green/red).
- **Status UI:** Right-side UI panel with WIN, LOSE, and ENTER 2 status boxes.
- **Flow Control:** Automatic puzzle reload when switching between puzzles.

1.2 Hardware Platform

- **Target Board:** Digilent Basys3 (Xilinx Artix-7 XC7A35T)
- **Clock:** 100 MHz system clock, divided to 25 MHz for VGA pixel clock
- **Display:** VGA output at 640×480 resolution
- **Input:** 5 push buttons (Center, Up, Down, Left, Right), 16 switches
- **Output:** 4-digit seven-segment display, VGA connector

2. System Architecture

The system follows a simplified modular design with the game engine handling all puzzle storage, cursor control, and correctness checking internally using arrays rather than separate RAM modules.

2.1 Module Hierarchy

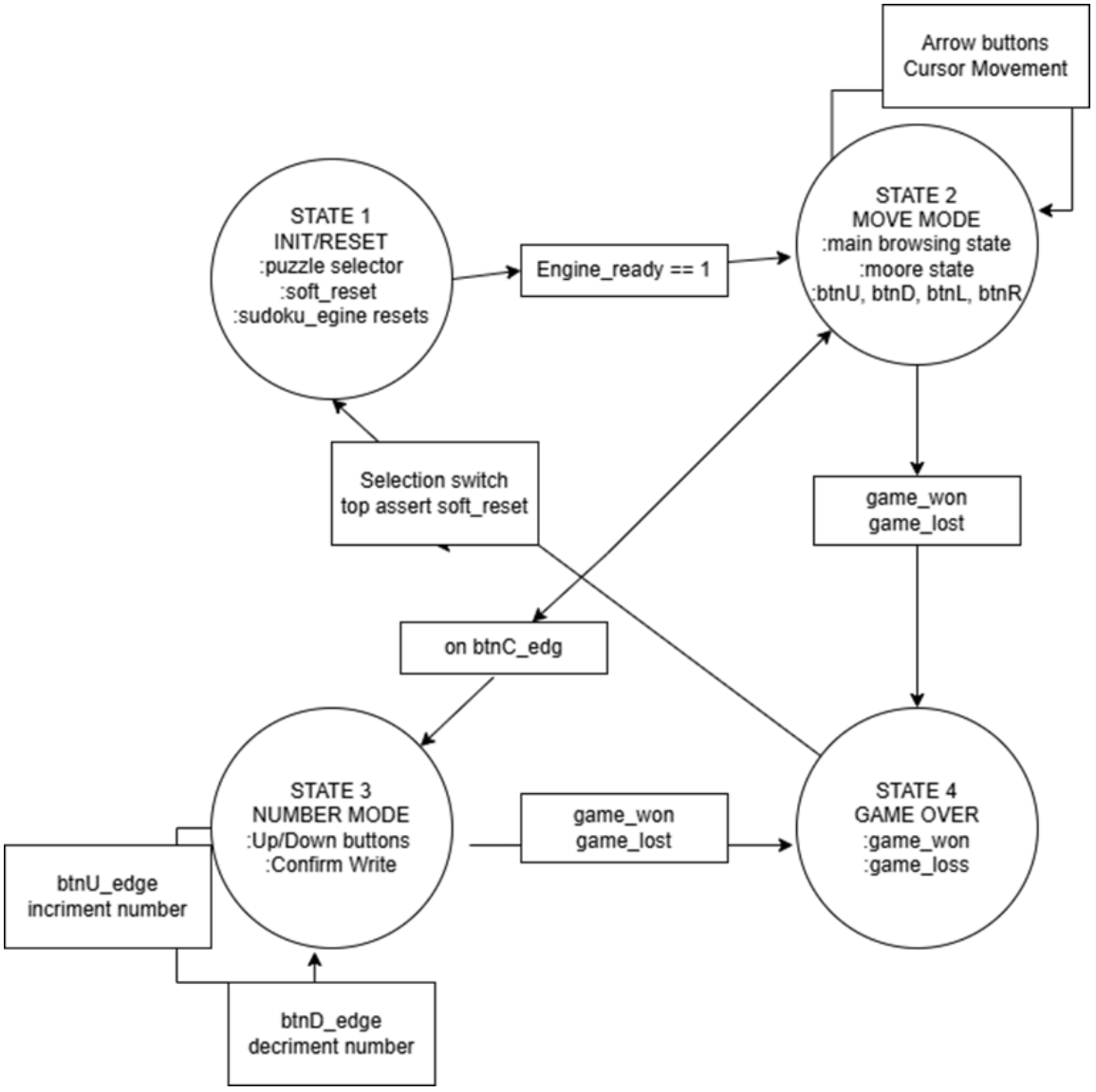
- top.sv (Top-level module)
    - clock_divider.sv (100MHz → 25MHz)
    - vga_controller.sv (VGA timing generation)
    - conditioner.sv (x5) (Button debounce/one-pulse)
    - sudoku_engine.sv (Game logic - integrated design)
    - sudoku_draw.sv (VGA pixel rendering)
        - digit_font.sv (8×12 digit bitmaps)
        - letter_font.sv (8×12 letter bitmaps)
    - sevenseg_sudoku.sv (7-segment controller)
        - sevenseg.sv (Low-level 7-seg driver)

2.2 Data Flow

1. **Puzzle Loading:** On reset or puzzle change, sudoku_engine reads from internal ROM (puzzles.mem via $readmemh) and populates internal grid, solution, and fixed_mask arrays.
2. **Gameplay:** sudoku_engine handles cursor movement and number entry directly, updating internal arrays.
3. **Display:** sudoku_draw reads grid_out and fixed_mask_out (combinationally exposed from engine) to render pixels in real-time with zero latency.
4. **Correctness:** When SW2 is high, the cell_match array indicates per-cell correctness; check_win/check_lose signals drive UI.

2.3 State Machines

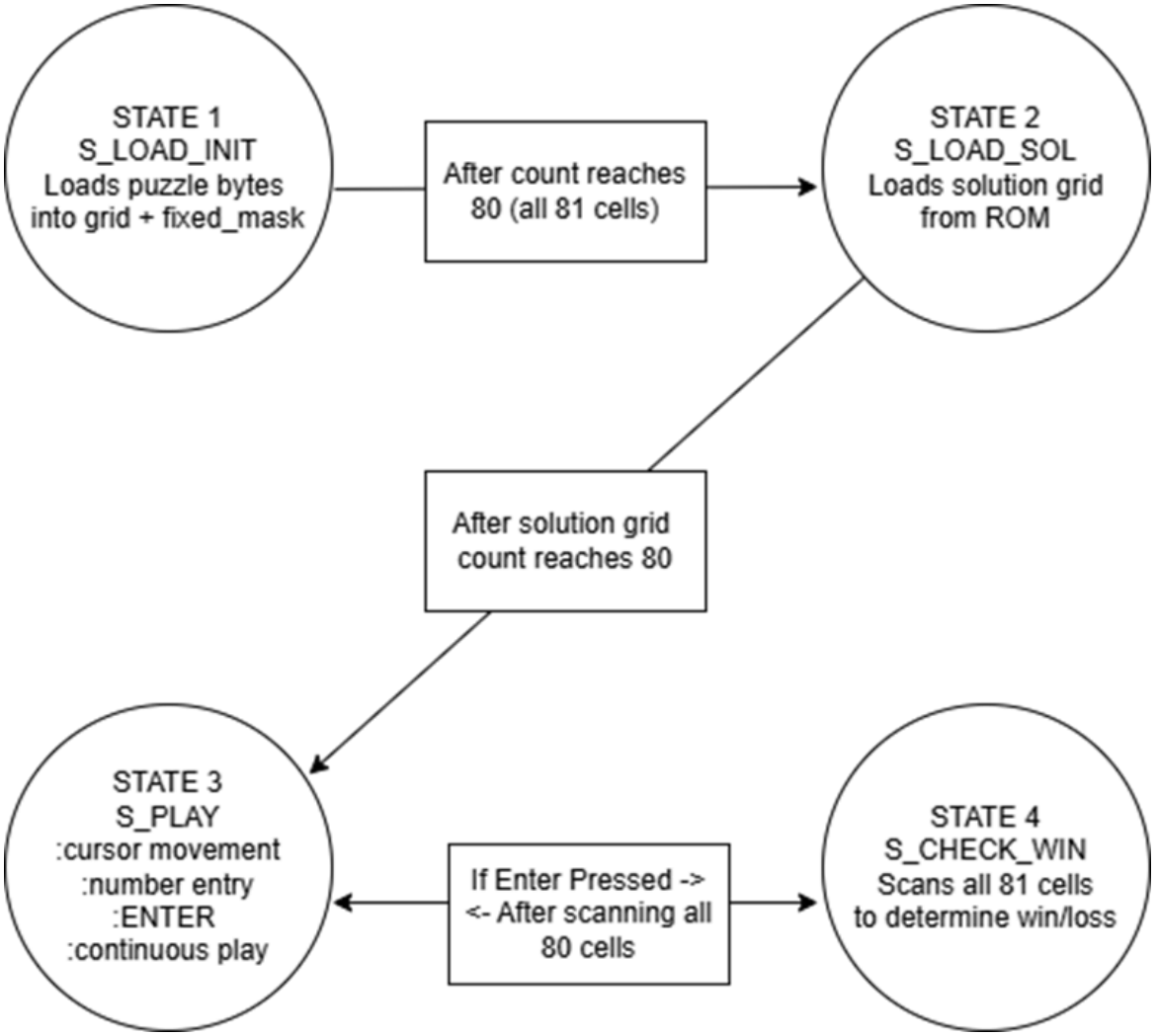
2.3.1 Mode State Machine (top.sv)



The top-level module implements a two-state mode machine for user input:

Mode	Button Functions	Cursor Display
MODE_MOVE	U/D/L/R: cursor movement C: switch to NUMBER	Solid (always visible)
MODE_NUMBER	U/D: increment/decrement number C: enter value	Flashing (~0.67s period)

2.3.2 Game Engine State Machine (`sudoku_engine.sv`)



The game engine operates with a four-state FSM:

State	Description	Transition
S_LOAD_INIT	Load puzzle grid from ROM	S_LOAD_SOL after 81 cycles
S_LOAD_SOL	Load the solution grid from ROM	→ S_PLAY after 81 cycles
S_PLAY	Active gameplay (cursor, entry)	→ S_CHECK_WIN on ENTER
S_CHECK_WIN	Verify solution correctness	→ S_PLAY after 81 cycles

3. Module Documentation

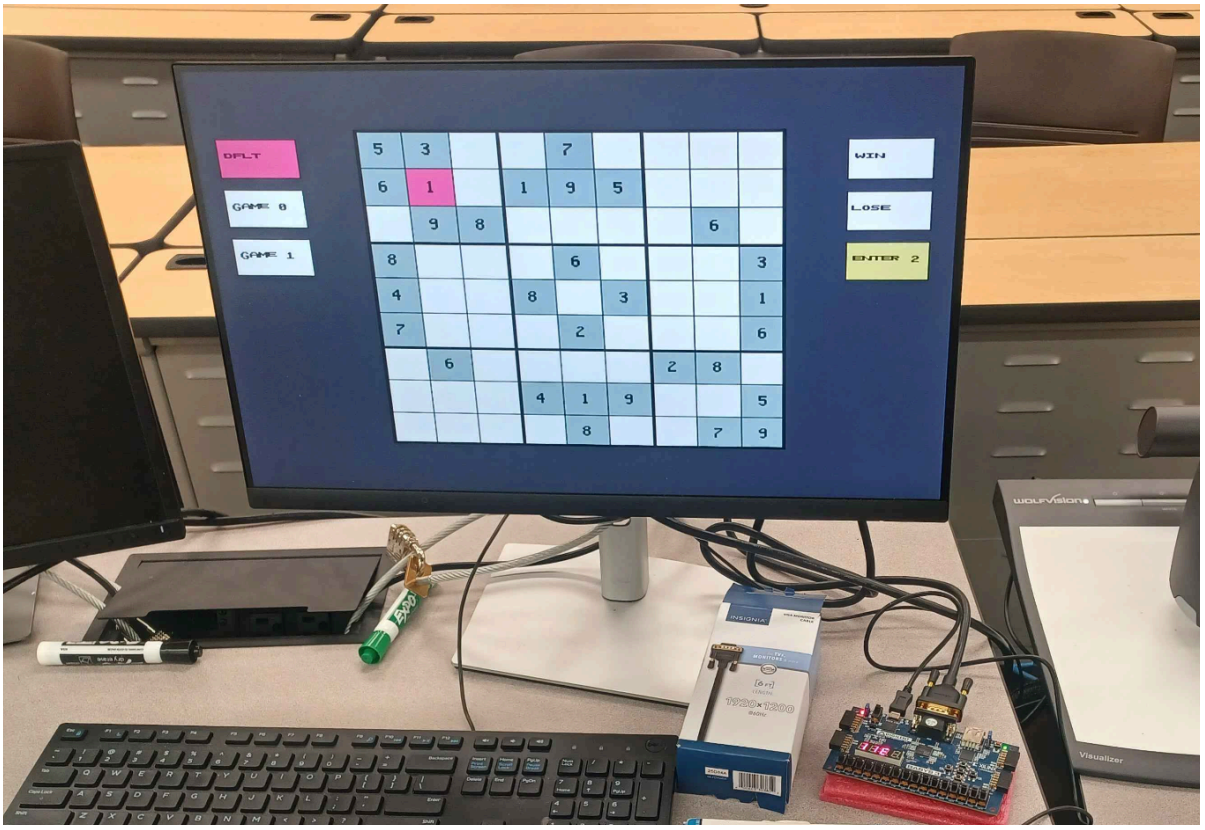
3.1 top.sv - Top-Level Module

- **Authors:** Gwendolyn Beecher, Constantine Hinds
- **Lines:** 278
- **Purpose:** Top-level integration module that instantiates and connects all subsystems.
- **Key Features:**
    - Switch synchronizers for stable puzzle selection (SW0, SW1).
    - Correctness mode is enabled via SW2 (`check_enable` signal).
    - Soft reset generation when puzzle selection changes.
    - Mode logic (MOVE vs. NUMBER) with a center-button toggle.
    - Cursor position tracking for automatic number selection in NUMBER mode.
    - Cursor flashing logic using bit 26 of a counter (~0.67s period).

3.2 sudoku_engine.sv - Game Engine

- **Authors:** Gwendolyn Beecher, Constantine Hinds
- **Lines:** 221
- **Purpose:** Central game engine that handles puzzle loading, cursor movement, number entry, and correctness checking.
- **Architecture:**
    - Internal storage arrays: `grid[0:8][0:8]`, `solution[0:8][0:8]`, `fixed_mask[0:8][0:8]`.
    - Direct ROM loading via `$readmemh("puzzles.mem", puzzle_rom)`.
    - Nibble extraction for cell values from packed ROM data.
    - Combinational grid exposure (`always_comb`) for zero-latency VGA display.
    - Real-time `cell_match` computation when `check_enable` is active.
    - Integrated cursor movement with wrap-around (0↔8).

3.3 sudoku_draw.sv - VGA Renderer



- **Authors:** Gwendolyn Beecher, Constantine Hinds
- **Lines:** 453
- **Purpose:** Combinational pixel color generation for VGA display, including game board, UI panels, and correctness overlay.
- **Layout:**
    - **Sudoku board:** 360×360 pixels (9×9 cells, 40px each), centered at (140, 60).
    - **Left panel:** Game selector boxes (DFLT, GAME 0, GAME 1) at x=20.
    - **Right panel:** Status boxes (WIN, LOSE, ENTER 2) at x=550.

Color Palette:

Element	Hex Color	RGB
Background	Dark blue-gray (0x334)	(51, 51, 68)
Pre-filled cell	Light teal (0xABB)	(170, 187, 187)

Editable cell	Off-white (0xFFE)	(255, 255, 238)
Cursor	Pink (0xF59)	(255, 85, 153)
Correct (SW2)	Green (0x4F4)	(68, 255, 68)
Incorrect (SW2)	Red (0xF44)	(255, 68, 68)
ENTER 2 box	Yellow (0xDD7)	(221, 221, 119)

3.4 vga_controller.sv - VGA Timing

- **Author:** Gwendolyn Beecher
- **Lines:** 67
- **Purpose:** Standard VGA timing generator for 640×480 @ 60Hz.

Parameter	**Horizontal**	**Vertical**
Active	640 pixels	480 lines
Front Porch	16 pixels	10 lines
Sync Pulse	96 pixels	2 lines
Back Porch	48 pixels	33 lines
Total	800 pixels	525 lines

3.5 Supporting Modules

Module	**Lines**	**Description**
clock_divider.sv	26	Divides 100MHz to 25MHz for VGA pixel clock
conditioner.sv	38	Button debounce (1.5M cycles) + one-pulse generation

digit_font.sv	168	8×12 bitmap font for digits 1-9
letter_font.sv	266	8×12 bitmap font for letters (G, A, M, E, D, F, L, T, S, W, I, N, R, O) and digits 0-2
sevenseg.sv	75	Low-level 7-segment driver with multiplexed refresh
sevenseg_sudoku.sv	75	Sudoku-specific display logic (position, edit, value)
puzzles.mem	7	Hex puzzle data: 6 rows x 81 nibbles (3 puzzles x init+solution)
Basys3_Master.xdc	295	Pin constraints for VGA, buttons, switches, and 7-segment

3.6 VGA Display Pipeline

The VGA display operates as a purely combinational pipeline synchronized to the pixel clock.

3.6.1 Color Priority System

Because multiple layers are needed to occupy the same pixel position (for example, the background, grid lines, highlighted cells, digit glyphs, correctness indicators, and flashing mode overlay), the VGA renderer must decide which layer's color should "win" at each pixel point. This was implemented as a deterministic color-priority system inside the pixel generation module.

Each pixel begins with the default background color, and then higher-priority layers can overwrite it. The fixed order occurs incrementally in priority as follows:

1. Background layer
2. Grid Line
3. Cell Highlighting
4. Digit Rendering
5. Correctness (Game state: Win/Lose)
6. Flashing mode (highest priority)

In SystemVerilog, this is implemented by using sequential conditional assignments within an always_comb block, where later conditions can override the earlier ones.

The ordering ensures that digits remain visible within a highlighted region, and correctness feedback appears over the digits. The flashing overlay can temporarily dominate the rendering to signal a specific cell in flashing mode.

3.6.2 Character (Digit and Letter) Bitmap Generation

When entering a value in NUMBER mode:

1. User in MODE_MOVE, cursor on desired cell
2. Press CENTER: switch to MODE_NUMBER
   - Cursor starts flashing
   - Seven-segment shows 'E' and flashing '1.'
   - Preview number appears in cursor cell (if empty)
3. Press UP/DOWN to cycle selected_number (1-9)
   - Preview updates on VGA
   - Seven-segment digit 3 flashes the new number
4. Press CENTER to confirm entry
   - If cell is NOT fixed (cell_is_editable = 1):
     - grid_we = 1, grid_din = selected_number
     /- Value written to Grid RAM
   - If cell IS fixed:
     - Write signals remain 0
     - No change occurs
   - Mode returns to MODE_MOVE
   - Cursor stops flashing

Digits (1–9) and UI text ("WIN", "LOSE", "ENTER", "DFLT", "GAME 1", "GAME 2") are rendered using hand-crafted bitmap fonts. The FPGA stores each character as a small grid of bits (typically around 8×8). Each bit represents whether a pixel inside the character's bounding box should be "lit."

3.6.3 Bitmap Storage Format

An 8x8 digit bitmap is stored in SystemVerilog as a logic array.

Example (Digit 5):

Code snippet - An 8x8 digit bitmap is stored in System Verilog as:

```
logic [7:0] bitmap_5 [0:7] = '{
        8'b11111111, //row 0
```

```
        8'b10000000, //row 1
        8'b11111110, //row 2
        8'b00000001, //row 3
        8'b00000001, //row 4
        8'b11111110, //row 5
        8'b10000001, //row 6
        8'b01111110, //row 7
};
```

An 8x8 letter bitmap letter example: “N.”

```
Logic [7:0] bitmap_N [0:7] = '{
        8'b100000001,
        8'b110000001,
        8'b101000001,
        8'b100100001,
        8'b100010001,
        8'b100001001,
        8'b100000101,
        8'b100000011,
        8'b100000001,
};
```

3.6.4 Pixel-to-Bitmap Coordinate Mapping

When the current VGA pixel (x, y) falls within the bounding box of a character, the renderer computes the corresponding 8×8 bitmap coordinate:

1. Compute Local Coordinates:

```
local_x = x – charOriginX;
local_y = y – charOriginaY;
```

2. Convert Local Pixel to Bitmap Row/Column:

```
col = local_x / pixelScale
row = local_y / pixelScale
```

For a 1:1 mapping:

- pixelScale = 1

- col and row are simply (0-7) indexes

3.6.5 Flashing Logic and Visibility Control

To help the player easily locate the currently selected Sudoku cell, the VGA system implements a flashing highlight overlay. This overlay periodically toggles its visibility and is positioned at the highest priority level of the rendering pipeline, ensuring it cannot be obscured by lower layers such as the background, grid lines, or cell shading. At the same time, the flashing logic must not hide the 8x8 digit glyphs inside the cell, which requires careful ordering with the color-priority chain.

3.6.6 Flash Clock Generation

The Basys3's 25 MHz VGA pixel clock is too fast to drive visible blinking, so a slow flashing signal is generated by incrementing a counter and using one of its higher bits as a toggling enable.

```
always_ff @(posedg clk25) begin
flashCounter <= flashcounter + 1;
flashOn <= flashCounter[24];
end
```

Key properties:

- Bit 24 toggles at approximately 1-2 Hz, which is visually comfortable.
- If the flashing is too fast or slow, adjusting the bit index changes the frequency without altering game logic.
- This produces a stable square-wave signal used to control the visibility of the flashing overlay.

3.6.7 Interaction with Color Priority

The flashing overlay is intentionally placed near the top of the color-priority chain. However, digits displayed using the 8×8 bitmap renderer must remain visible even during flashing. To achieve this, the flashing overlay is applied **BEFORE** digit rendering in the final priority ordering:

```
//Flash overlay (highest priority)
if (flashOn && isSelectedCell)
    pixelColor = FLASH_COLOR;
// 8x8 digit rendering overrides flash
if (digitPixel)
    pixelColor = DIGIT_COLOR;
```

This ensures that the cell visibly flashes and the digit inside is always readable. This design choice resolved a bug during testing in which digits vanished when flashing was applied.

3.6.8 Integrated Rendering Pipeline

The VGA subsystem integrates timing, combinational rendering, game logic, and priority rules into a single real-time pipeline:

- **VGA Timing Generation:** Generates pixel coordinates + sync signals.
- **Coordinating Region Decoder:** Determines pixel's logical meaning.
- **8x8 Bitmap Rendering Engine:** Converts digits/letters into pixel-level geometry.
- **Flash Engine (Highlighting):** Toggles the visibility of the selected cell.
- **Priority Resolver (Correctness Overlays):** Chooses final pixel color.
- **VGA Output:** Sends RGB to the monitor.

Although each module serves a distinct purpose, they must work together synchronously to generate a stable, accurate 640 x 480 VGA output at 60 frames per second.

3.6.9 No Frame Buffer: True Hardware Rendering

Unlike GPU-based graphics, this FPGA Sudoku implementation uses direct procedural rendering:

- No memory is used to store a completed image.
- No frame buffer is double-buffered or swapped.
- The board is redrawn entirely from logic and registers every frame (25 million times per second).

The rendering process begins with the VGA timing generator:

1. A 25 MHZ pixel clock drives the horizontal and vertical counters.
2. The counters determine:
    - (x, y) pixel coordinates
    - Visible region (0 – 639 for x, 0-479 for y)
    - Hsync and Vsync pulse periods
3. Only pixels within the visibility region are assigned colors; all other pixels are output as black.

This ensures strict compliance with VGA standards, providing the monitor with a reliable and stable signal.

Here is the formatted section with the numbering corrected to follow the previous content (starting at 3.6.6) and the text preserved verbatim.

3.6.6 Color Composition and Priority Resolution

After the pixel's region is classified, the VGA renderer determines the final color for that pixel by evaluating each possible layer in a fixed, deterministic order. This stage does not compute shapes; it decides "which layer wins."

The renderer begins with a default background color and conditionally overwrites it based on the pixel's classification flags:

```
pixelColor = BG_COLOR;

if(isGridLine) pixelColor = GRID_COLOR;

if(isBoxBoundary) pixelColor = BOX_COLOR;

if(isCorrectnessMark) pixelColor = HIGHLIGHT_COLOR;

if(digitPixel) pixelColor = DIGIT_COLOR;

if(flashOn && isSelectedCell) pixelColor = FLASH_COLOR;

if(uiPixel) pixelColor = UI_COLOR;
```

Only one color is ultimately assigned for the current pixel. Because this logic is purely combinational and evaluated every pixel cycle, the VGA output responds instantly to game state changes without needing any form of frame buffer.

3.6.7 Integration with Game Logic and State Registers

The VGA pipeline does not store frame images. Instead, the current visual state of the board is reconstructed in real time from the system's registers:

- **grid_vals[r][c]**: the number currently stored in each cell
- **fixed_mask[r][c]**: identifies uneditable starter digits
- **cursor_x, cursor_y**: controlled by MODE_MOVE
- **preview_number**: used only in MODE_NUMBER
- **flash_state**: toggled by the counter in top.sv
- **game_over, win_state, lose_state**: determine UI overlays

Each of these signals is fed directly into sudoku_draw.sv, meaning that any change in gameplay state is visible on the very next frame.

3.6.8 Per-Pixel Deterministic Rendering Workflow

For every pixel, during every frame, the rendering pipeline executes the following sequence:

1. Timing generator produces (x, y) and sync pulses
2. Visibility check ensures that only the active region is drawn
3. Region classification determines which type of object occupies the pixel
4. Bitmap lookup is performed if the pixel lies inside a digit/letter bounding box
5. The flashing state is applied to selected-cell pixels
6. The color priority resolver selects the final color
7. RGB output is driven to the DAC pins
8. Pipeline repeats for the next pixel

This happens 25 million times per second, producing a stable 60 FPS display.

3.6.9 No Frame Buffer: True Hardware Rendering

Unlike GPU-based graphics, this FPGA Sudoku implementation uses direct procedural rendering:

- No memory is used to store a completed image
- No frame buffer is double-buffered or swapped
- The board is redrawn entirely from logic and registers every frame

Advantages:

- Minimal resource usage (fits comfortably in Basys3's LUTs)
- Deterministic outputs across runs
- Instant visual updates driven purely by combinational logic
- No tearing or flickering, since pixels are generated exactly as VGA expects.

3.6.10 Summary of Pipeline Integration

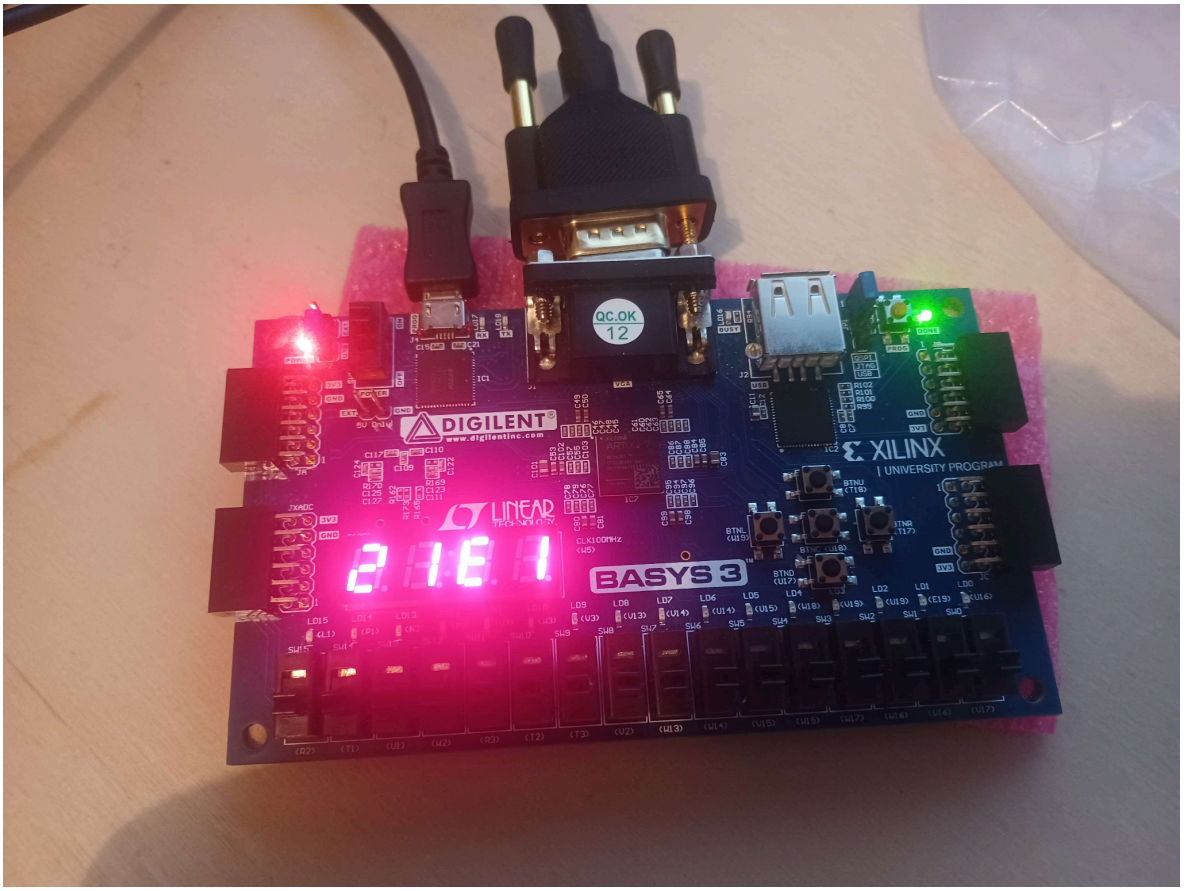
The complete system integrates timing, combinational rendering, game logic, and priority rules into a single real-time pipeline:

Stage	**Purpose**
VGA Timing	Generates pixel coordinates + sync signals
Region Decoder	Determines pixel's logical meaning

Bitmap Engine	Converts digits/letters into pixel-level geometry
Flash Engine	Toggles selected cell visibility
Priority Resolver	Chooses the final pixel color
VGA Output	Sends RGB to the monitor

Together, these modules form the complete VGA display subsystem that drives the Sudoku UI.

3.7 Seven-Segment Display Interface



The Basys3 seven-segment display provides immediate, real-time feedback on the game state. This allows the player to track their cursor position and current input mode without obstructing the VGA view. The sevenseg_sudoku.sv module drives this unit, initiating the lower-level sevenseg. The SV driver manages the multiplexing (refreshing digits approx. every 4ms, simulating static numbers).

The display logic is context-sensitive. So, its behavior changes based on the current state of the main engine (MODE_MOVE vs. MODE_NUMBER). The four digits are utilized as follows:

Display Logic & Pin Mapping

Digit (Location)	**MODE_MOVE (Navigation)**	**MODE_NUMBER (Entry/Edit)**
Digit 0 (Left)	**Cursor X Position** (0–8)	**Cursor X Position** (0–8)
Digit 1	**Cursor Y Position** (0–8)	**Cursor Y Position** (0–8)
Digit 2	Blank (No indicator)	**'E'** (Edit Mode Indicator)
Digit 3 (Right)	**Current Cell Value** (1–9 or 0 for empty)	**Selected Input Number** (Flashing)

Operational Details:

- **Navigation Feedback (Digits 0 & 1):** The left two digits continuously track the cursor's coordinates on the 9x9 grid, helping the user identify the exact cell they are modifying.
- **Number Mode Indication (Digit 2):** When the user switches to number entry mode (Edit), Digit 2 displays the character **'E'**. In navigation mode, this digit remains blank to reduce visual clutter.
- **Value Preview (Digit 3):**
    - In **MOVE** mode, this displays the value currently stored in the selected grid cell.
    - In **NUMBER** (Edit) mode, this displays the potential number the user is about to enter. This digit blinks at 2Hz to indicate that the value is tentative and has not yet been committed to the grid.

4. Removed Modules

The following modules were removed from the original design because the RAM-based architecture with shadow registers was too complex. The simplified design uses internal arrays instead, which provides immediate VGA display updates. The RAM and Shadow register approach introduced structurally inherent synchronization delays.

4.1 Modules Removed

Module	Reason for Removal
sudoku_loader.sv	Loading logic integrated into sudoku_engine.sv (S_LOAD_INIT/S_LOAD_SOL states)
sudoku_gameplay.sv	Cursor and number entry logic integrated into sudoku_engine.sv (S_PLAY state)
sudoku_checker.sv	Verification logic integrated into sudoku_engine.sv (S_CHECK_WIN state)
shadow_register.sv	Replaced by a combinational always_comb block exposing grid arrays directly
ram.sv	Replaced by internal logic arrays (grid, solution, fixed_mask)
puzzle_rom.sv	Replaced by direct $readmemh in sudoku_engine.sv
sudoku_games.sv	Hardcoded puzzles unused; puzzles.mem is the primary data source

4.2 Technical Justification

The original RAM-based architecture caused a VGA display delay bug:

- In **NUMBER** mode, the seven-segment display updated immediately when numbers were incremented.
- However, the VGA display lagged until the value was actually written to RAM.
- **Root cause:** shadow_register is only updated on RAM write, not on preview_number change.

The simplified architecture resolves this by:

- Using internal arrays instead of RAM modules.
- Exposing `grid_out` via combinational `always_comb` (zero latency).
- Eliminating RAM arbitration and shadow register synchronization.
- Reducing total module count from 19 to 12 files.

5. Test Bench

The test bench was essential for verifying the timing of the engine states and calibrating the flash interval. Below is a snippet of the test vector file used for simulation.

```
// sudoku_engine.tv

// timing adjustment for line number match. The lines are two off, so line 161 was line 163

1_0_0_0_0_0_0_0_0_00_00_0_0  // vec 0: reset active

0_0_0_0_0_0_0_0_0_00_00_0_0  // vec 1: release reset, S_LOAD_INIT starts

0_0_0_0_0_0_0_0_0_00_00_0_0  // vec 2

//… vec 3 through 161 are all the same …//

0_0_0_0_0_0_0_0_0_00_00_0_0  // vec 161

0_0_0_0_0_0_0_0_1_00_00_0_0  // vec 162: ready=1


0_0_1_0_0_0_1_0_1_00_00_0_0  // vec 163: valid=1, right=1 vec 163-165: Move RIGHT

0_0_0_0_0_0_0_0_1_01_00_0_0  // vec 164: release (x=1 immediately)

0_0_0_0_0_0_0_0_1_01_00_0_0  // vec 165: hold x=1


0_0_1_0_1_0_0_0_1_01_00_0_0  // vec 166: valid=1, down=1 // vec 166-168: Move DOWN

0_0_0_0_0_0_0_0_1_01_01_0_0  // vec 167: release (y=1 immediately)

0_0_0_0_0_0_0_0_1_01_01_0_0  // vec 168: hold y=1


0_0_1_0_0_1_0_0_1_01_01_0_0  // vec 169: valid=1, left=1 // vec 169-171: Move LEFT

0_0_0_0_0_0_0_0_1_00_01_0_0  // vec 170: release (x=0 immediately)

0_0_0_0_0_0_0_0_1_00_01_0_0  // vec 171: hold x=0

0_0_1_1_0_0_0_0_1_00_01_0_0  // vec 172: valid=1, up=1


0_0_0_0_0_0_0_0_1_00_00_0_0  // vec 173: release (y=0 immediately) // vec 172-174: Move UP

0_0_0_0_0_0_0_0_1_00_00_0_0  // vec 174: hold y=0

0_0_0_0_0_0_0_0_1_00_00_0_0  // vec 175: hold all // vec 175: Final Safety Line (Prevents Ghost Errors)
```

```
module TestbenchSudokuEngine();

  parameter NUM_VECTORS = 175;  // Total vectors in SudokuEngine.tv (0 to 175)

  logic I_CLK, I_RESET;
  logic I_SEL, I_VALID, I_UP, I_DOWN, I_LEFT, I_RIGHT, I_ENTER;
  logic I_READY, I_WON, I_LOST;
  logic I_READY_EXP, I_WON_EXP, I_LOST_EXP;
  logic [1:0] I_CUR_X_EXP, I_CUR_Y_EXP;
  logic [3:0] I_CUR_X, I_CUR_Y;
  logic [31:0] vectornum, errors;

  logic [14:0] testvectors[0:NUM_VECTORS-1];

  // Unused outputs
  logic [3:0] I_VAL;
  logic I_CHKW, I_CHKL;
  logic I_MATCH [0:8][0:8];
  logic [3:0] I_GRID [0:8][0:8];
  logic I_FIXED [0:8][0:8];

  // Debug Signals
  logic [1:0]  dbg_state;
  logic [6:0]  dbg_load_counter;
  logic [3:0]  dbg_load_row, dbg_load_col;
  logic        dbg_up_evt, dbg_down_evt, dbg_left_evt, dbg_right_evt, dbg_enter_evt;
  logic [3:0]  dbg_current_x, dbg_current_y, dbg_current_val;
  logic        dbg_engine_ready, dbg_game_won, dbg_game_lost;
```

```
assign dbg_state        = dut.state;
assign dbg_load_counter = dut.load_counter;
assign dbg_load_row     = dut.load_row;
assign dbg_load_col     = dut.load_col;
assign dbg_up_evt       = dut.up_evt;
assign dbg_down_evt     = dut.down_evt;
assign dbg_left_evt     = dut.left_evt;
assign dbg_right_evt    = dut.right_evt;
assign dbg_enter_evt    = dut.enter_evt;
assign dbg_current_x    = dut.current_x;
assign dbg_current_y    = dut.current_y;
assign dbg_current_val  = dut.current_val;
assign dbg_engine_ready = dut.engine_ready;
assign dbg_game_won     = dut.game_won;
assign dbg_game_lost    = dut.game_lost;

sudoku_engine dut (
    .clk(I_CLK),
    .reset(I_RESET),
    .puzzle_selector({1'b0, I_SEL}),
    .cmd_number(4'b0000),
    .cmd_up(I_UP),
    .cmd_down(I_DOWN),
    .cmd_left(I_LEFT),
    .cmd_right(I_RIGHT),
    .cmd_enter(I_ENTER),
    .cmd_valid(I_VALID),
    .check_enable(1'b0),
    .current_x(I_CUR_X),
    .current_y(I_CUR_Y),
```

```
        .current_val(I_VAL),
        .game_won(I_WON),
        .game_lost(I_LOST),
        .engine_ready(I_READY),
        .check_win(I_CHKW),
        .check_lose(I_CHKL),
        .cell_match(I_MATCH),
        .grid_out(I_GRID),
        .fixed_mask_out(I_FIXED)
    );

    // Clock generation
    always begin
        I_CLK = 0; #5; I_CLK = 1; #5;
    end

initial
        begin
            // UPDATED: Initialize memory to X
            for (int i=0; i<256; i=i+1) testvectors[i] = 15'bx;

            // NEW FIX: Initialize the expected signal so we don't quit early
            I_READY_EXP = 0;

            $readmemb("SudokuEngine.tv", testvectors);
            vectornum = 0; errors = 0;

                    // Pre-load first vector before simulation starts
        {I_RESET, I_SEL, I_VALID, I_UP, I_DOWN, I_LEFT, I_RIGHT, I_ENTER,
         I_READY_EXP, I_CUR_X_EXP, I_CUR_Y_EXP, I_WON_EXP, I_LOST_EXP} =
testvectors[0];
```

```
    end



  // Main test logic - runs on every positive clock edge
  always @(posedge l_CLK) begin
    #1;  // Wait for signals to settle


    // Check for errors on the current vector
    if (l_READY !== l_READY_EXP) begin
      $display("Error vec %0d: ready=%b, expected=%b", vectornum, l_READY,
l_READY_EXP);
      errors = errors + 1;
    end
    if (l_CUR_X[1:0] !== l_CUR_X_EXP) begin
      $display("Error vec %0d: curX=%0d, expected=%0d", vectornum, l_CUR_X[1:0],
l_CUR_X_EXP);
      errors = errors + 1;
    end
    if (l_CUR_Y[1:0] !== l_CUR_Y_EXP) begin
      $display("Error vec %0d: curY=%0d, expected=%0d", vectornum, l_CUR_Y[1:0],
l_CUR_Y_EXP);
      errors = errors + 1;
    end
    if (l_WON !== l_WON_EXP) begin
      $display("Error vec %0d: won=%b, expected=%b", vectornum, l_WON,
l_WON_EXP);
      errors = errors + 1;
    end
    if (l_LOST !== l_LOST_EXP) begin
      $display("Error vec %0d: lost=%b, expected=%b", vectornum, l_LOST,
l_LOST_EXP);
      errors = errors + 1;
```

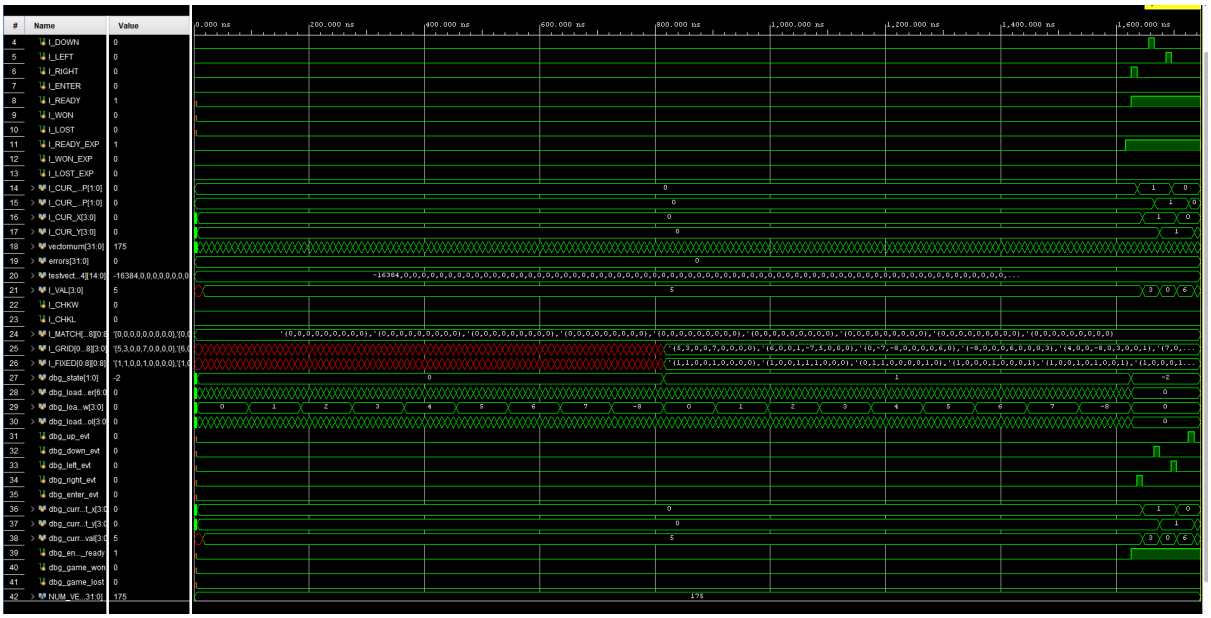
```
        end

        // Move to next vector
        vectornum = vectornum + 1;

        // Check if we've processed all vectors
        if (vectornum >= NUM_VECTORS) begin
            $display("--------------------------------------------");
            if (errors == 0)
                $display("SUCCESS: All %0d vectors passed!", NUM_VECTORS);
            else
                $display("COMPLETED: %0d vectors, %0d errors", NUM_VECTORS, errors);
            $display("--------------------------------------------");
            $finish;
        end

        // AI Acknowledgement: Added for debugging timing error --Load next vector
        {I_RESET, I_SEL, I_VALID, I_UP, I_DOWN, I_LEFT, I_RIGHT, I_ENTER,
         I_READY_EXP, I_CUR_X_EXP, I_CUR_Y_EXP, I_WON_EXP, I_LOST_EXP} =
testvectors[vectornum];
    end

endmodule
```

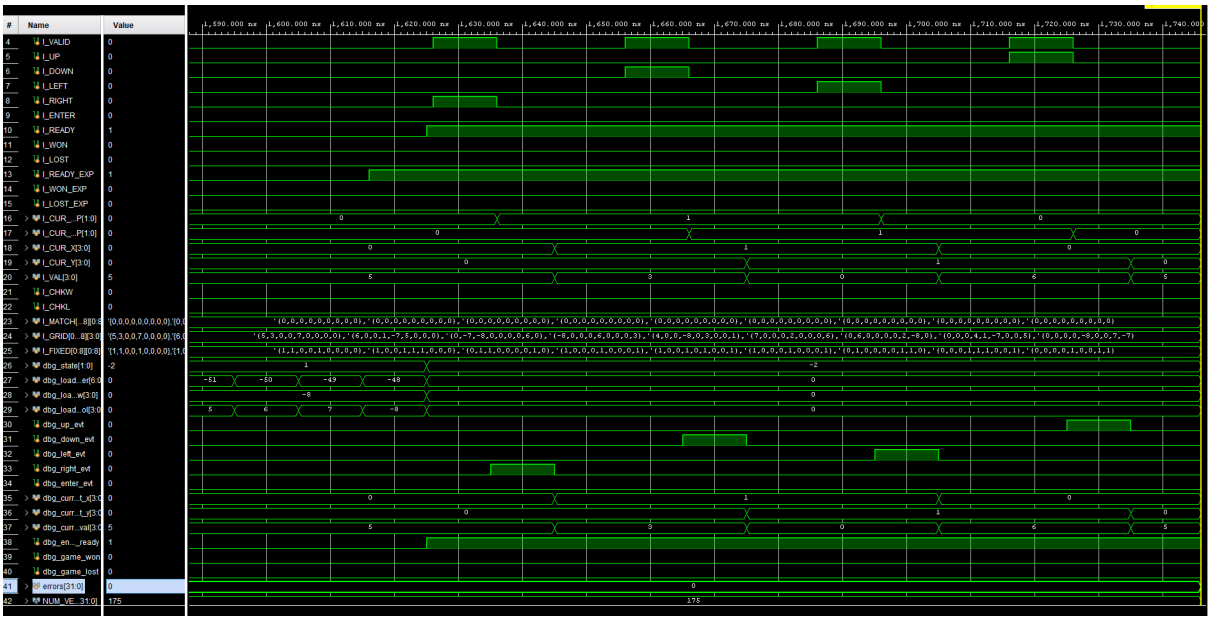
Simulation Module: TestbenchSudokuEngine

Parameters: NUM_VECTORS = 175

Key Debug Signals: dbg_state, dbg_load_counter, dbg_engine_ready

```
source TestbenchSudokuEngine.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without a wave window. If
#   }
# }
# run 1000ns
WARNING: Too many words specified in data file SudokuEngine.tv
INFO: [USF-XSim-96] XSim completed. Design snapshot 'TestbenchSudokuEngine_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:02 ; elapsed = 00:00:08 . Memory (MB): peak = 3152.988 ; gain = 0.000
run all
------------------------------------------------
SUCCESS: All 175 vectors passed!
------------------------------------------------
$finish called at time : 1746 ns : File "C:/Users/suprbludude/Downloads/CSCEA342_SudokuProject/sudoku_1_0_0/sudoku
remove_forces { {/TestbenchSudokuEngine/l_CLK} {/TestbenchSudokuEngine/l_RESET} }
```




Name
Value
i_VALID
i_UP
i_DOWN
i_LEFT
i_RIGHT
i_ENTER
i_READY
i_WON
i_LOST
i_READY_EXP
i_WON_EXP
i_LOST_EXP
i_CUR_...P[1:0]
i_CUR_...P[1:0]
i_CUR_X[3:0]
i_CUR_Y[3:0]
i_VAL[3:0]
i_CHKW
i_CHKL
i_MATCH[...8][0:8]
i_GRID[0...8][3:0]
i_FIXED[0:8][0:8]
dbg_state[1:0]
dbg_load...ed[6:0]
dbg_loa...w[3:0]
dbg_load...o[3:0]
dbg_up_evt
dbg_down_evt
dbg_left_evt
dbg_right_evt
dbg_enter_evt
dbg_curr...t_x[3:0]
dbg_curr...t_y[3:0]
dbg_curr...val[3:0]
dbg_en..._ready
dbg_game_won
dbg_game_lost
errors[31:0]
NUM_VE...31:0]
175

6. Final File Listing

File	Lines	Description
top.sv	278	Top-level integration module
sudoku_engine.sv	221	Game state machine and logic
sudoku_draw.sv	453	VGA pixel rendering with UI panels
letter_font.sv	266	8×12 letter bitmaps
digit_font.sv	168	8×12 digit bitmaps
sevenseg_sudoku.sv	75	Sudoku-specific 7-seg logic
sevenseg.sv	75	Low-level driver
clock_divider.sv	26	Clock generation
conditioner.sv	38	Input debouncing
vga_controller.sv	67	VGA timing generation
puzzles.mem	7	Puzzle data file
Basys3_Master.xdc	295	Constraints file

7. AI Usage Acknowledgment

In accordance with academic integrity requirements, the following documents all instances where AI assistance (Claude by Anthropic) was used during this project's development.

7.1 Report Generation

This report was generated by Claude AI (Anthropic) from a draft outline and the complete project source files provided by the authors.

7.2 Summary of AI-Assisted Development Tasks

1. **Architecture Simplification:** Proposed removing the RAM/shadow register architecture to fix the VGA display delay bug.
2. **Engine Consolidation:** Integrated loader, gameplay, and checker logic into a single `sudoku_engine` module.
3. **Correctness Checking:** Added `check_enable`, `check_win`, `check_lose`, and `cell_match` outputs for SW2 feature.
4. **UI Enhancement:** Added right-side status boxes (WIN, LOSE, ENTER 2) to `sudoku_draw`.
5. **Font Extension:** Added letters I, N, R, O to `letter_font.sv` for WIN/LOSE/ENTER display.
6. **Cursor Tracking:** Added automatic number selection update when the cursor moves in NUMBER mode.
7. **Seven-Segment Integration:** Guided integration of flashing behavior and mode-aware display.
8. **General Debugging:** Various bug fixes, including ROM loading, display timing, and state machine transitions.
9. **Code Documentation:** Added comments.
10. **Report Generation:** Created this technical documentation.
11. **Code Cleanup:** Cleaned up overall code structure and organization.
12. **VGA Pipeline Explanations:** AI assisted in documenting the VGA display pipeline, including timing, layering, pixel-level priority resolution, and 8×8 bitmap character rendering.
13. **Color Priority Clarification:** Helped articulate the color-layer priority rules used in the Sudoku renderer and worked through visibility bugs involving digits, highlights, and flashing overlays.
14. **Flashing System Integration:** Supported the implementation and documentation of the flash counter, `flashOn` logic, and its interaction with selected-cell highlighting and digit rendering.

15. **Bitmap Rendering Documentation:** Helped formalize the explanation of the 8×8 bitmap system, including local coordinate mapping, scaling, row/column indexing, and glyph authoring notes.
16. **VGA Documentation Formatting:** Organized and formatted Sections to present a clear, structured, and academically styled pipeline explanation suitable for a capstone project report.
17. **Seven-Segment Display Logic:** Assisted in structuring the multiplexing logic and the context-sensitive display modes (navigation vs. entry) to ensure visual consistency with the VGA output.
18. **Test Bench & Timing Validation:** Modified the `TestbenchSudokuEngine` and associated test vectors for debugging to verify engine state transitions and signal timing. Also used to determine the correct flashing interval rate for the cursor to ensure it was human-readable.

7.3 Human Contributions

The following work was performed primarily by the human engineers:

- Original project concept and requirements definition.
- Initial architecture design and module planning.
- VGA controller timing implementation.
- Digit and letter font bitmap design.
- Puzzle selection and puzzle data.
- Clock divider implementation.
- Button conditioner/debouncer logic.
- Hardware testing and validation on Basys3.
- All design decisions and architectural choices.
- Final integration and verification.
- Complete VGA rendering logic (`sudoku_draw.sv`) including grid line detection, box boundary rendering, and digit positioning.
- Manual creation and testing of all 8×8 digit and letter bitmaps.
- Hardware debugging of the VGA output on the physical Basys3 board.

8. References and Coursework Attribution

This project builds upon foundational digital logic modules and starter code provided by **Professor Neumeyer** for the **CSCE A342 - Digital Circuits** course at the University of Alaska Anchorage.

8.1 Course Materials and Starter Code

The following modules were adapted from course assignments and instructor-provided templates:

- **Conditioner.sv**: Input signal debouncing and synchronization logic used for all button inputs.
- **seven_seg.sv**: The low-level seven-segment display multiplexing driver, originally developed for the *Seven Segment RAM Assignment*. The team repurposed the module to drive the Sudoku status display.
- **vga_controller.sv**: The VGA timing logic (horizontal/vertical sync and counter generation) was adapted from the VGApulses module provided in earlier VGA lab exercises (referenced from oneFile.sv).
- **Basys3_Master.xdc**: The pin constraint file is based on the standard master file provided by Digilent and the instructor. The file was customized for this project's specific I/O requirements.

8.2 Previous Assignments

Concepts and logic from previous coursework contributed directly to this final implementation:

- **Keyboard & Input Processing:** Techniques for signal synchronization and state decoding were adapted from the *PS2 Decoder* and *Keyboard to Seven Segment* labs (ps2_decoder.sv, keyboard_to_seven_seg.sv).
- **Test Bench Verification:** The structure of the simulation test bench (TestbenchSudokuEngine.sv) follows the verification methodologies introduced in the *Rotary Encoder* assignment (TestbenchRotaryToLED.sv).