FPGA experiment report

Snake game

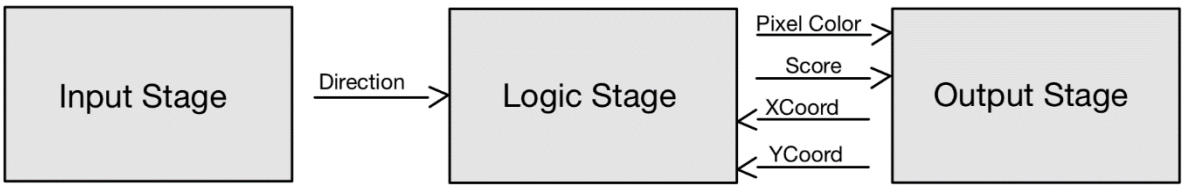
Student 1 name: Bshara Rhall ID: 208264085
Student 2: name: Lior Daniel ID: 208829259

Abstract
In this experiment, we implemented a classic Snake game on a Basys3 FPGA using fully parameterized Verilog modules for game control, body management, food generation, and collision detection. The Game_FSM orchestrates state transitions, VGA_Interface renders an 800×600 display at 72 Hz, and PS/2 input handles user commands. Simulation and on-board testing confirmed correct snake movement, growth, collision handling, and score tracking. The modular architecture readily supports future enhancements such as adjustable game speed.

Motivation
Implementing Snake on the Basys3 FPGA unifies several prior lab exercises, VGA video timing, PS/2 keyboard interfacing, and seven-segment display control into one cohesive, parameterized Verilog design. By centralizing frame-tick generation, direction latching, and game-flow management within a single FSM, this project reinforces advanced digital-design concepts such as finite-state machines, synchronous logic, and modular reuse. Additionally, the configurable grid dimensions and snake length exemplify how parameterization enhances adaptability and paves the way for future features like variable speed.

Global Architecture

High Level block diagram of the Top module



The block diagram above shows the system partitioned into three functional stages:

- **Input Stage**
  Handles all user inputs: the reset button (Debouncer), PS/2 clock and data lines (Ps2_Interface), and translation of scan codes into a 2-bit direction command (Direction_Decoder).
- **Logic Stage**
  Core game control and state management: the Game_FSM generates frame ticks, latches new directions, and issues update_snake, score_inc, and reset_game pulses;

Snake_Logic maintains and shifts the snake's body on a 100×75 grid; Food_Generator produces pseudo-random food positions, Collision_Detector flags food_eaten and collision events, Score_Counter tallies points.

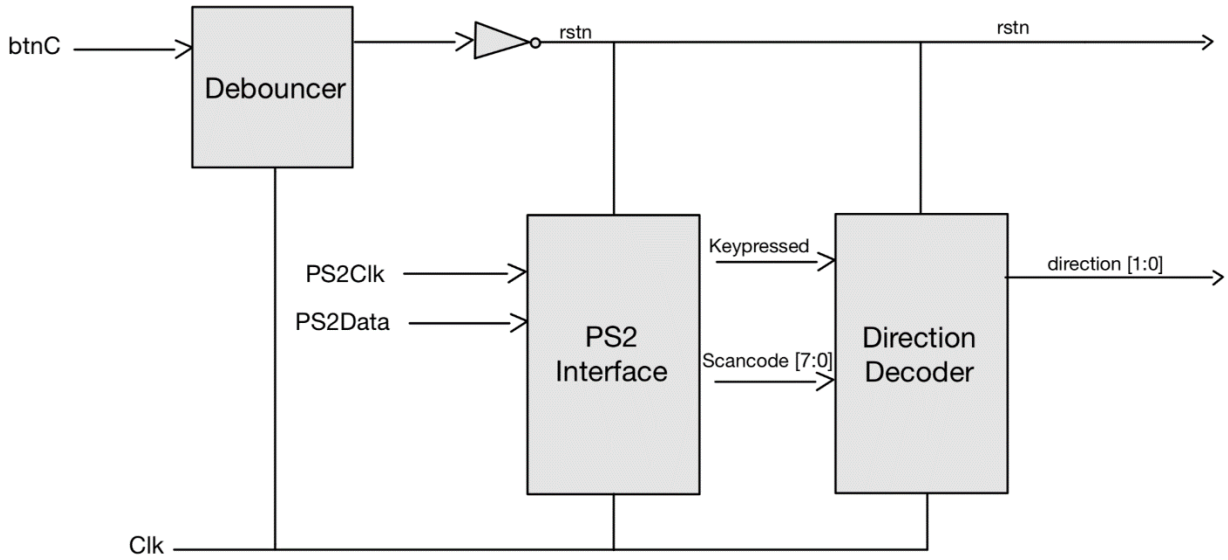
- **Output Stage**
  Renders game state and score: Snake_Renderer merges the grid (snake + food) into a 12-bit pixel_color bus; VGA_Interface drives an 800X600 72 Hz display, Seg_7_Display shows the current score on four seven-segment digits. Utility modules (CSA, FA, Lim_Inc) support arithmetic operations in rendering and counting.

1 Input Stage

1.1 Description

- **Debouncer** filters the mechanical, **active-high** btnC, producing reset_db. In the top module, reset_db is inverted to yield an **active-low** rstn.
- **Ps2_Interface** monitors the PS/2 lines at the falling edge of PS2Clk, shifting in 10 bits (start + 8 data + parity). When complete, it extracts an 8-bit scancode, filters out 0xE0/0xF0 prefixes, and asserts a one-cycle keyPressed (scancode_valid) in the 100 MHz clock domain.
- **Direction_Decoder** samples scancode only when scancode_valid is high, maps arrow-key make-codes to a 2-bit direction (00=Up, 01=Right, 10=Down, 11=Left), and blocks any immediate 180° reversal.

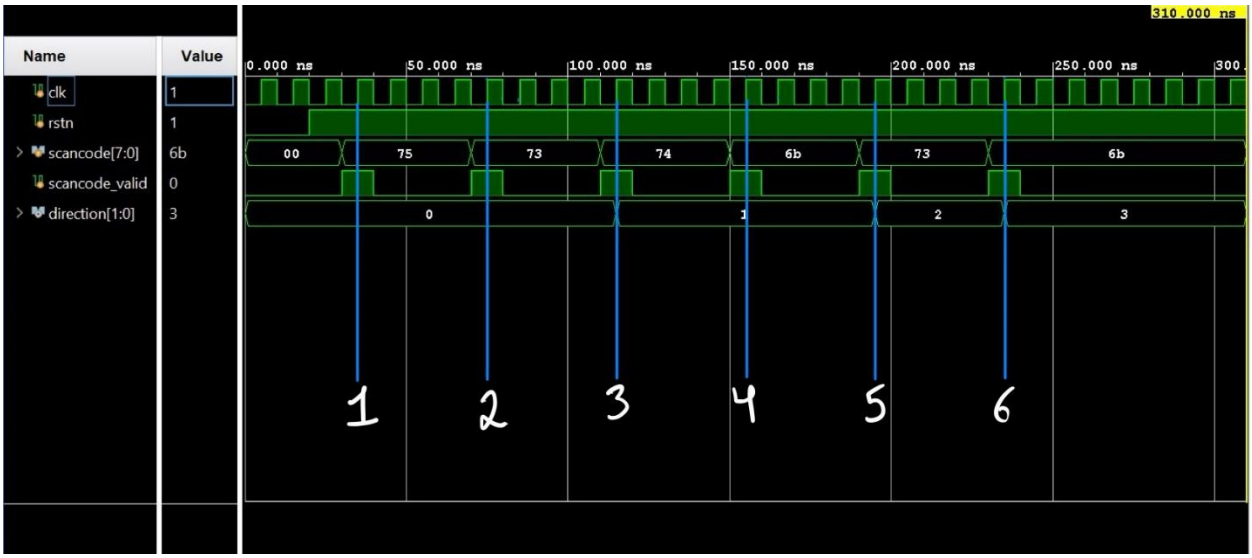
1.2 Block Diagram



As shown above, btnC is debounced and inverted to generate an active-low rstn signal that resets all modules. The PS2 Interface samples PS2Clk/Data to serially shift in, filter, and validate each scan code pulsing scancode_valid with the resulting 8-bit code which the Direction Decoder then translates into a 2-bit direction_out.

1.3 Testbench & Simulation

Direction_Decoder:



The provided simulation applies a sequence of PS/2 scan codes and checks that the 2-bit direction output behaves correctly:

1. **(1)** The first scancode_valid pulse carries 0x75 (‘8’). At this marker, direction updates to 00 (UP).
2. **(2)** The next pulse is 0x73 (‘5’). Because this would reverse UP→DOWN, it is ignored and direction remains 00.
3. **(3)** A subsequent pulse of 0x74 (‘6’) is valid, so direction changes to 01 (RIGHT).
4. **(4)** The following 0x6B (‘4’)—which would reverse RIGHT→LEFT—is blocked, leaving direction at 01.
5. **(5)** The next 0x73 (‘5’) is now allowed (RIGHT→DOWN), so direction becomes 10 (DOWN).
6. **(6)** Finally, 0x6B (‘4’) is accepted, and direction updates to 11 (LEFT).

These checks confirm both correct mapping of PS/2 codes to movement commands and enforcement of the no-immediate-reversal rule.

```
# run 1000ns
direction = 00 (expected 00 = UP)
direction = 00 (expected 00 = still UP)
direction = 01 (expected 01 = RIGHT)
direction = 01 (expected 01 = still RIGHT)
direction = 10 (expected 10 = DOWN)
direction = 11 (expected 11 = LEFT)
$finish called at time : 310 ns : File "C:/Users/ASUS/Desktop/ׁׁsnake codes/Direction Decoder tb.v" Line 64
```

2 Logic Stage

2.1 Description

- **Game_FSM**:
  The Game_FSM module centrally manages the Snake game's timing and control signals. It divides the 100 MHz system clock down to a frame rate (~20 Hz) and sequences the core game operations—snake movement, scoring, and game-over behavior—according to three states: **IDLE**, **RUN**, and **GAMEOVER**.

Key I/O Signals:

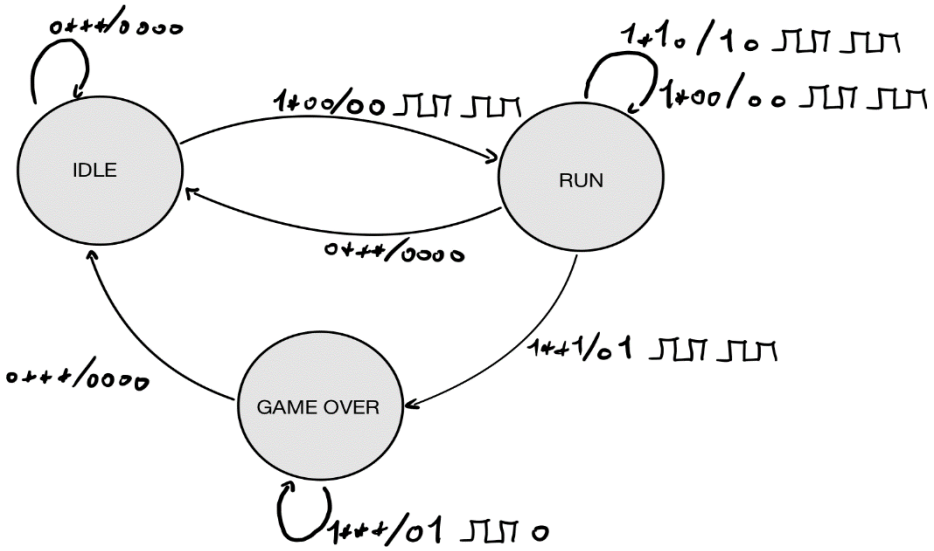
- **Inputs:**
  - clk (100 MHz system clock)
  - rstn (active-low reset)
  - direction_in[1:0] (latched movement command from PS/2)
  - food_eaten (pulse when the snake's head meets food)
  - collision (high when the snake collides with wall or itself)
- **Outputs:**
  - frame_tick (one-clock pulse at each new frame)
  - update_snake (pulse to advance the snake body)
  - score_inc (pulse to increment the score once per frame after eating)
  - reset_game (high to indicate Game Over state)
  - direction_out[1:0] (current movement direction forwarded downstream)

Operation:

1. **Clock Division & Frame Tick**
   A 25-bit tick_counter increments every clk cycle. When it reaches the parameter TICKS_PER_FRAME (default 5 000 000), it resets to 0 and asserts frame_tick for one cycle thereby producing a steady 20 Hz timing reference for all game updates.
2. **State Machine**
   - **IDLE:** On reset (rstn = 0), the FSM stays in IDLE with all control pulses deasserted. When rstn goes high, it transitions to RUN.
   - **RUN:**
     - At each frame_tick, it pulses update_snake to shift the snake.
     - If a food_eaten event occurred since the last frame, it emits score_inc once on that tick.
     - If collision is high at any time, it moves to GAMEOVER.
     - It continuously forwards direction_in to direction_out.
   - **GAMEOVER:**
     - Asserts reset_game high to indicate Game Over State.
     - Remains here until rstn is deasserted, returning to IDLE on the next reset.

This structure ensures that all game updates, movement, growth, scoring, and termination occur synchronously on well-defined frame boundaries, with clean handoff of user direction commands and reliable collision handling.

FSM Diagram:



rstn, direction_in [1:0], food_eaten, collision **/** score_inc, reset_game, frame_tick, update_snake

The Game_FSM is a three-state synchronous Mealy machine: in **IDLE** it waits for reset release (rstn=1) before entering **RUN**, where every frame_tick pulses update_snake (and, if eaten_latched is high, also pulses score_inc), a collision on a frame_tick still pulses update_snake but transitions the FSM to **GAMEOVER**, in which reset_game remains asserted regardless of further ticks until a reset (rstn=0) returns it to **IDLE**. Because its outputs (update_snake, score_inc, reset_game) are produced based on both the current state and input signals at transition time, it implements a Mealy finite-state machine.

- **Snake_Logic**

Maintains and updates the snake's body on a GRID_W×GRID_H grid in a 2-stage pipeline: it samples move and eat pulses, computes the new head position, shifts (and grows) the body array, updates head and length outputs, and flattens the internal array into a wide bus for rendering.

Key I/O Signals

- **Parameters**
  - GRID_W, GRID_H (grid dimensions)
  - MAX_LEN (maximum segments)
  - INIT_LEN (initial length)
  - POS_BITS (bits to index the grid)
- **Inputs**
  - clk (100 MHz system clock)
  - rstn (active–low reset)

    - update_snake (one-cycle pulse per frame)
    - food_eaten (one-cycle growth pulse)
    - direction_in[1:0] (00=Up, 01=Right, 10=Down, 11=Left)
- **Outputs**
    - snake_head[POS_BITS-1:0] (flat index of current head)
    - snake_length[$clog2(MAX_LEN):0] (current segment count)
    - snake_body_flat[POS_BITS*MAX_LEN-1:0] (flattened body indices)

Operation

1. **Reset & Initialization**
    - On rstn=0, the module centers the head at (GRID_W/2, GRID_H/2) by setting head_x and head_y accordingly, and clears the stage-1 latches upd_s1 and eat_s1.
    - It sets snake_length to INIT_LEN, computes the flat center index for snake_head, and clears the grow_req flag.
    - The first INIT_LEN entries of the internal snake_body array are initialized to consecutive grid cells extending left from center, and the remaining entries are zeroed.
2. **Pipeline Stages**
    - **Stage 1**

      On each rising clock, the pulses update_snake and food_eaten are latched into upd_s1 and eat_s1.

      If update_snake is high, the head coordinates (head_x, head_y) advance one cell in the direction given by direction_in, wrapping at the grid edges.

    - **Stage 2 Index, Shift & Grow**

      On each rising clock, the grow_req flag is set whenever eat_s1 is high and cleared when upd_s1 occurs. If upd_s1 is asserted, the module computes $new\_head = head\_y * GRID\_W + head\_x$, shifts the entire snake_body array down by one and inserts new_head at index 0, increments snake_length if grow_req is set and snake_length < MAX_LEN, and finally updates snake_head to new_head.

3. Flatten for Rendering

A generate loop packs each snake_body[idx] into its slice of the wide snake_body_flat bus (since Vivado cannot export 2-D ports), giving the renderer a single, frame-aligned vector of all segment indices.

Example: Screen-Grid Mapping
Suppose we simplify to a 4×4 grid with GRID_W=4, GRID_H=4, and INIT_LEN=3.
thus, our screen is

Y/X	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

Suppose on reset:
The body array is [9, 5, 1], and the direction is right. Therefore for the next head:
head_x=2, head_y=2 → new_head = 2*4+2 = 10
and body array becomes [10, 9, 5].

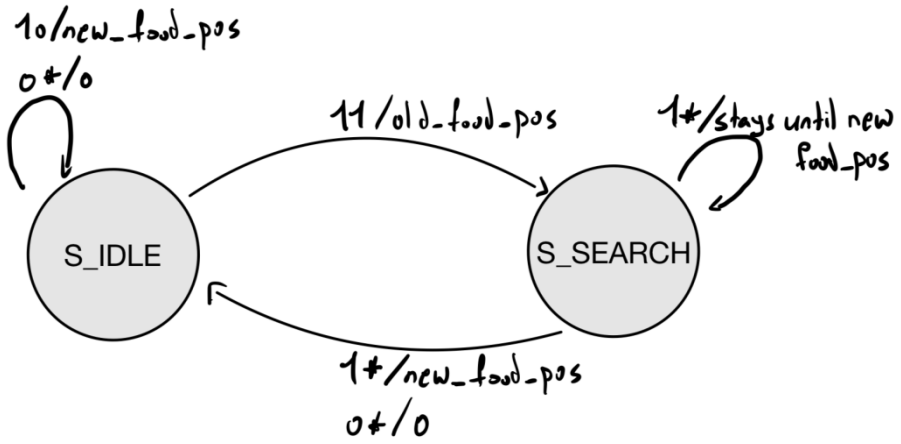
- **Food_Generator**
  Generates a new pseudo-random food position whenever the snake eats, ensuring it falls within the grid and does not overlap the snake's body.
  Key I/O Signals
  - **Parameters**
    - GRID_W, GRID_H (grid dimensions)
    - MAX_LEN (max segments)
    - POS_BITS (bits to index GRID_W×GRID_H)
  - **Inputs**
    - clk, rstn (active-low reset)
    - food_eaten (one-cycle pulse when snake head equals food)
    - snake_body_flat[POS_BITS*MAX_LEN-1:0] (flattened body indices)
    - snake_length[$clog2(MAX_LEN):0] (current snake length)
  - **Output**
    - food_pos[POS_BITS-1:0] (current food index)

  Operation
  Hardware does not provide true randomness, so we implement a **13-bit Galois LFSR** to generate a maximal-length pseudo-random sequence. On each clock cycle in the search state, the register shifts left by one bit, and the new LSB is the XOR of selected taps (bits 12, 3, 2, and 0). This "feedback" bit insertion yields a long, uniform sequence with minimal logic and low latency.
  The module's two-state FSM works as follows:
  1. **S_IDLE:**
     - After reset, food_pos initializes to a known value (zero) and the LFSR seed is set (13'h1).
     - When food_eaten pulses, transition to **S_SEARCH**.

2. **S_SEARCH:**
    - Each cycle, the LFSR advances, producing a new candidate index idx.
    - A combinational collides(idx) function checks the first snake_length entries of snake_body_flat.
    - If idx < GRID_W*GRID_H and !collides(idx), the FSM returns to **S_IDLE** on the next cycle and updates food_pos = idx.
    - Otherwise, it remains in **S_SEARCH** and tries the next LFSR output.

This design efficiently “randomizes” food placement purely in hardware no multipliers, dividers, or memory while guaranteeing the food never appears under the snake.

FSM Diagram:



rstn, food_eaten **/** food_pos

As seen above: in **S_IDLE** the food position is held until food_eaten=1, then it jumps to **S_SEARCH**, where each lfsr_nxt is tried until it’s in bounds and collision-free upon which food_pos is updated and it returns to **S_IDLE**. the food_pos register is only loaded when we are in **S_SEARCH and** the inputs (lfsr_nxt, collision flags) satisfy the acceptance test, its next value is a function of **both** the current state and those inputs, therefore it’s a two-state synchronous Mealy FSM.

- **Collision_Detector**

  Detects when the snake eats food or collides with a wall or itself, producing two one-cycle outputs:
    - **food_eaten**: asserted whenever the snake’s head index matches the current food position
    - **collision**: asserted on any wall hit or self-intersection

  Key I/O Signals
    - **Parameters:** MAX_LEN (must match Snake_Logic), POS_BITS (index width), GRID_W, GRID_H (board size)
    - **Inputs:**
        - snake_head [POS_BITS-1:0] (flat index of head)

- snake_body_flat [POS_BITS*MAX_LEN-1:0] (flattened body indices)
- snake_length [6:0] (current number of segments)
- food_pos [POS_BITS-1:0] (flat index of food)
- direction_in [1:0] (movement command)

- **Outputs:**
  - food_eaten (high when head == food_pos)
  - collision (high on wall hit or self-collision)

Operation

1. **Food Detection:**
   assign food_eaten = (snake_head == food_pos);
   A direct comparison flags any time the head reaches the food cell.
2. **Wall Collision:**
   Compute coordinates:
   $$head\_x = snake\_head \,\%\, GRID\_W;$$
   $$head\_y = snake\_head \,/\, GRID\_W;$$
   If the head is at the left edge and moving LEFT, or the right edge and moving RIGHT, or the top/bottom edges and moving UP/DOWN respectively, assert a wall collision.
3. **Self-Collision**:
   A generate/for loop compares snake_head against each live segment in snake_body_flat [1 … snake_length–1]. If any match is found, a col_self flag is set.
4. **Final Collision Signal:**
   assign collision = col_self || wall_collision;
   Combining wall and self flags yields the overall collision output.

This module therefore guarantees that every move is checked for both eating and crashing, feeding clean, single-cycle pulses back to **Game_FSM** for scoring or game-over handling.

- **Score_Counter**
  Maintains the player's score in BCD format (four digits, 0–9999), incrementing by one on each valid food-eaten event and rolling over cleanly using dedicated limit-adders.
  Key I/O Signals
  - **Inputs:**
    - clk (100 MHz system clock)
    - rstn (active-low reset)
    - score_inc (one-cycle pulse from Game_FSM on food eaten)
  - **Output:**
    - score[15:0] (16-bit BCD: [15:12]=thousands, [11:8]=hundreds, [7:4]=tens, [3:0]=ones)

Operation

1. **Cascaded BCD Incrementers**
   Four instances of the Lim_Inc module (parameterized with L=10) implement each BCD digit's increment and carry behavior:
   - **Ones digit (inc0)**
     - Input a = score[3:0], carry-in ci = score_inc.
     - Outputs ones_next and carry-out c1 when a + ci == 10.
   - **Tens digit (inc1)**
     - Input a = score[7:4], carry-in ci = c1 from ones.
     - Outputs tens_next and carry-out c2.
   - **Hundreds digit (inc2)**
     - Input a = score[11:8], carry-in ci = c2.
     - Outputs hundreds_next and carry-out c3.
   - **Thousands digit (inc3)**
     - Input a = score[15:12], carry-in ci = c3.
     - Outputs thousands_next (rolls over to 0 after 9) and c4 (unused).
2. **Register Update**
   An always @(posedge clk or negedge rstn) block resets score to 16'd0 when rstn=0. On any score_inc event, it loads the concatenated {thousands_next, hundreds_next, tens_next, ones_next} at the next clock edge, ensuring all four digits update simultaneously.

This design cleanly isolates each decimal digit's logic, handles carry propagation via dedicated limit-adders, and guarantees rollover behavior at 9999 without additional logic.

- **Font_ROM**
  Provides a tiny, combinational ROM storing 8×8 monochrome bitmaps for the characters in "GAME OVER," enabling the Snake_Renderer to display crisp text without large memory.

  Key I/O Signals
  - **Inputs:**
    - char_idx [3:0] — selects which letter (0='G', 1='A', 2='M', 3='E', 4=space, 5='O', 6='V', 7='E', 8='R')
    - row_idx [2:0] — selects the row (0..7) within the 8×8 bitmap
  - **Output:**
    - bits [7:0] — one bit per pixel (MSB = leftmost) for the selected row of the letter

  Operation

  An always @(*) block uses a two-level case statement: the outer case on char_idx selects which character's ROM to read, and the inner case on row_idx outputs the corresponding 8-bit pattern. Any undefined char_idx values fall through to a 0 (bg_color) output. This fully unrolled, nested-case approach synthesizes to a small lookup table of registers or distributed RAM, providing immediate combinational access to any letter row.

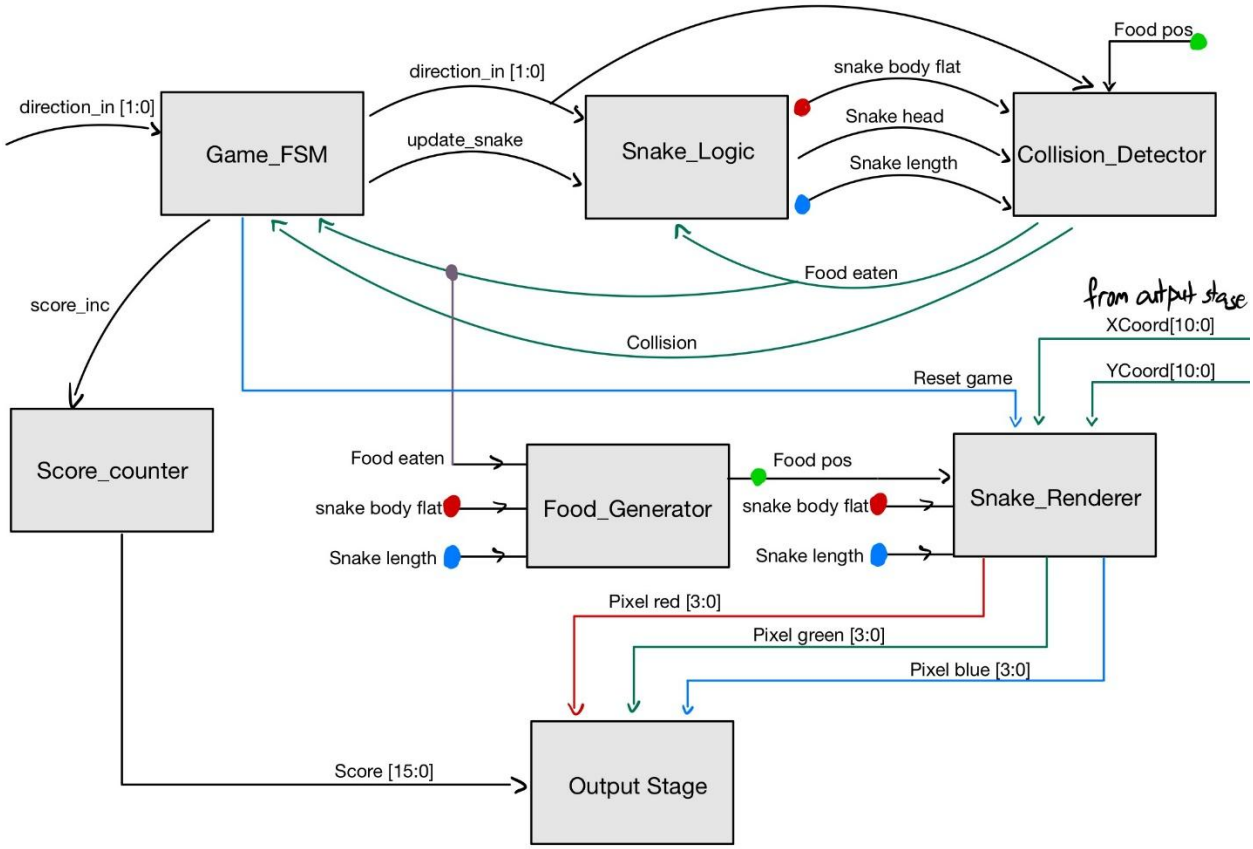
- **Snake_Renderer**
  Renders each cell of the 100×75 game grid to VGA pixels, combining the current snake body and food position into colored output during play, and, upon game over, displays a centered, fading “GAME OVER” message using the built-in 8×8 Font_ROM.
  Key I/O Signals
  - **Inputs:**
    - **clk** 100 MHz system clock
    - **rstn** Asynchronous, active-low reset
    - **XCoord, YCoord** Current cell coordinates (0…99, 0…74)
    - **snake_body_flat** Flattened list of all potential segment positions
    - **snake_length** Number of active segments (1…MAX_LEN)
    - **food_pos** Index of the food cell
    - **game_over** High when the game has ended
  - **Outputs:**
    - **pixel_red, pixel_green, pixel_blue** 4-bit RGB color for the current cell

  Operation
  1. **Grid-to-Linear Mapping & Snake Detection**
     - Convert the 2D coordinates (XCoord, YCoord) into a single linear index.
     - Compare this index against each of the active snake segment positions, if any match, flag that the current cell is occupied by the snake.
  2. **Fade Generator**
     - A free-running 27-bit counter (free_counter) increments on each rising clk (reset by rstn); its upper four bits (free_counter[26:23]) form a 4-bit fade level, which when game_over is asserted its applied as the gray-scale text color (txt_color = fade) and its bitwise inverse as the background (bg_color = ~fade), producing a smooth fade transition of the gray/white background and text.
  3. **Text Window Placement**
     - Determine the pixel region needed to display “GAME OVER” (9 characters × 8×8 cells each).
     - Center that region within the 100×75 grid by calculating its start coordinates once at reset.
  4. **Per-Cell Color Selection**
     - **During play (game_over = 0):**
       - If the cell matches a snake segment → output bright green.
       - Else if it matches the food position → output bright red.
       - Otherwise → output black.
     - **On game over (game_over = 1):**
       - Alternate background and text colors based on the fade signal (light gray background/dark gray text vs. dark gray background/light gray text).

- If the current cell lies within the previously centered text window, look up the corresponding bit in the Font_ROM to decide whether to draw a character pixel (text color) or background.
- All other cells simply show the current background color.

This organization cleanly separates the normal rendering path from the game-over display while minimizing storage and logic by reusing a small Font_ROM and a single fade counter.

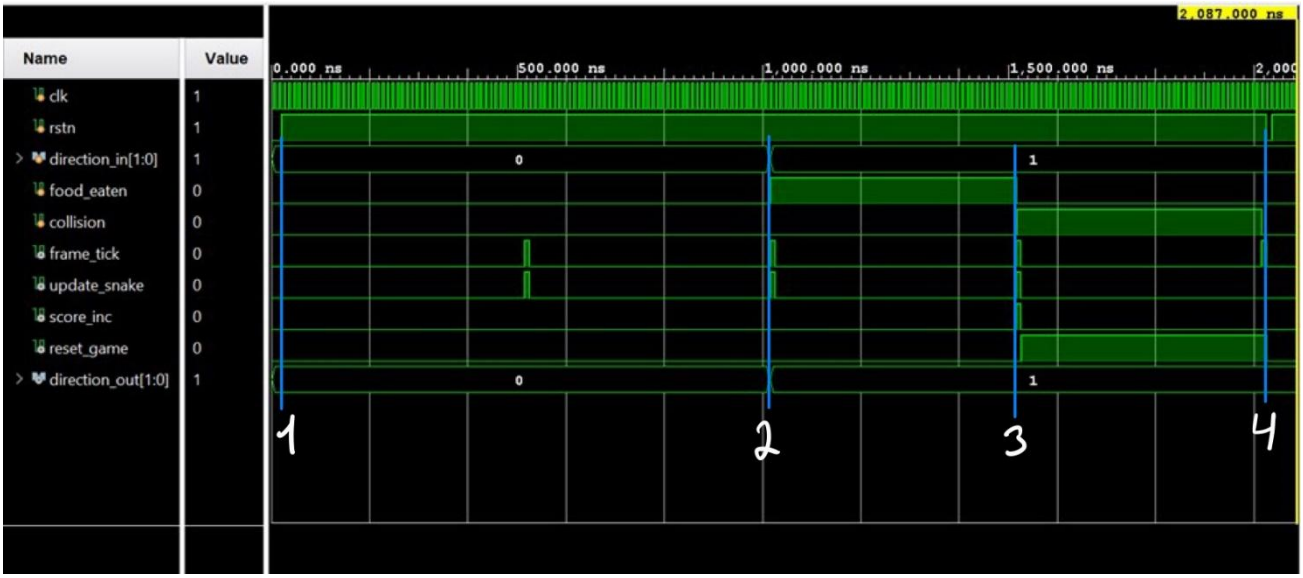
2.2 Block Diagram



As shown in the high-level block diagram, Game_FSM generated the update_snake, score_inc and reset_game pulses in response to collision and food_eaten flags from Collision_Detector. Snake_Logic then updated the head position and body vector on each movement tick, while Collision_Detector compared those signals against the food_pos and board boundaries. Upon food consumption, Food_Generator using the snake_body_flat and length vectors to avoid occupied cells produced a new food_pos, and Snake_Renderer combined XCoord/YCoord (from the output stage) inputs with the snake and food data to output 4-bit RGB pixel streams. Simultaneously, Score_Counter incremented the 16-bit score on each score_inc, and both pixel data and score fed into the Output Stage to drive the video and score displays.

2.3 Testbench & Simulation

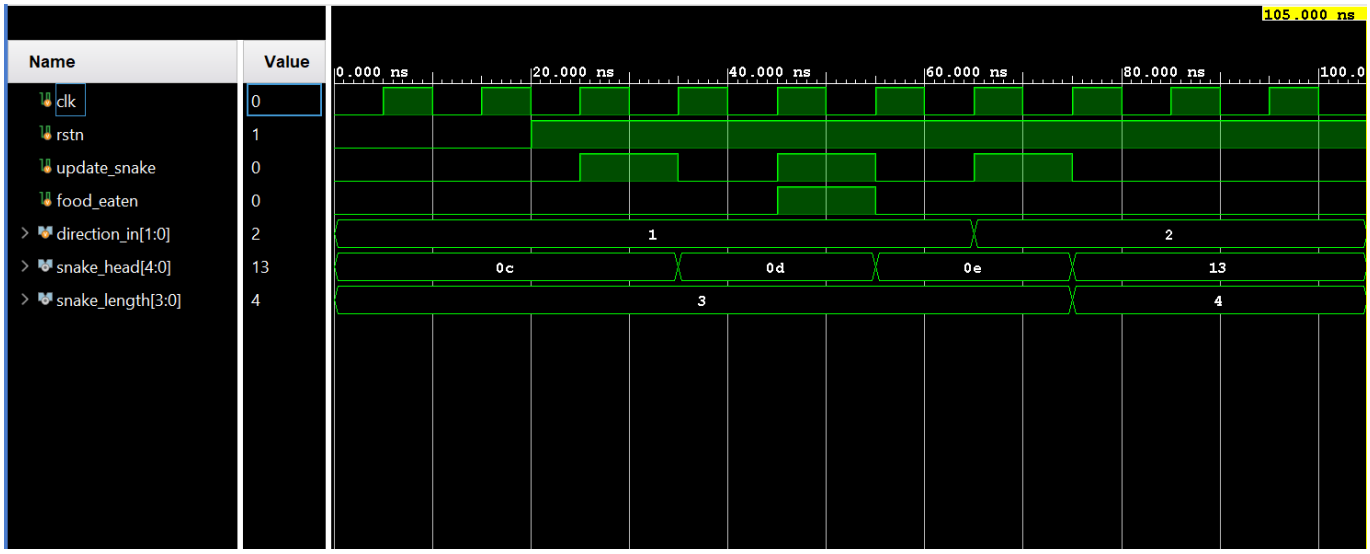
Game_FSM:



The provided simulation applies a sequence of timed inputs to the Game_FSM and checks that its key outputs behave correctly:

1. **(1) Reset Release**
   At 20 ns, rstn goes from 0→1. This should move the FSM out of IDLE into RUN, with reset_game low and no spurious pulses on score_inc or update_snake.
2. **Frame-Tick Verification**
   The testbench then waits for two rising edges of frame_tick. With TICKS_PER_FRAME=50, these should appear roughly every 500 ns, confirming the reduced-tick simulation is running as expected.
3. **(2) Food-Eaten → Score Increment**
   direction_in is set to 01 and food_eaten asserted just before the next frame_tick. On that tick, score_inc must pulse high for one cycle, then return low—demonstrating correct score-increment timing.
4. **(3) Collision → Game-Over**
   With collision high at the following frame tick, the FSM must assert reset_game (GAMEOVER) on that tick and hold it until an external reset, confirming collision detection and state transition.
5. **(4) Return to Idle via Reset**
   The testbench then pulses rstn low for 10 ns and back high. Immediately after, reset_game must go low again and the FSM re-enter IDLE (briefly) then RUN showing proper reset behaviour.

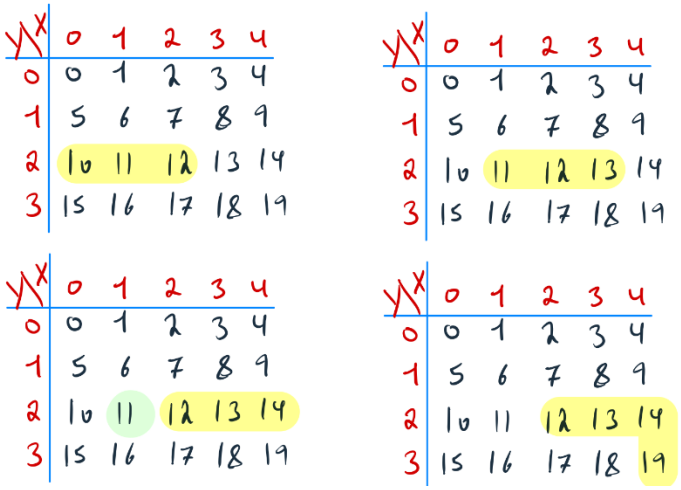
Snake_Logic:



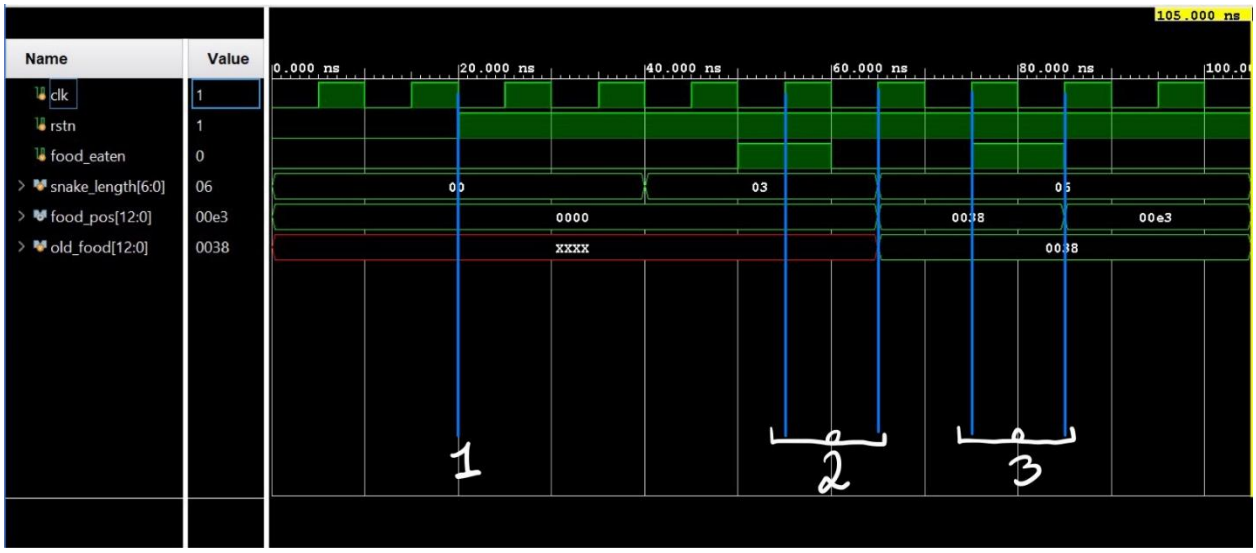

```
# run 1000ns
INIT : head=12, len=3, body=[12,11,10]
MOVE1: head=13, len=3, body=[13,12,11,10]
GROW1: head=14, len=3, body=[14,13,12,11]
MOVE2: head=19, len=4, body=[19,14,13,12]
$finish called at time : 105 ns : File "C:/Users/ASUS/Desktop/┐┐snake codes/Snake Logic tb.v" Line 102
```

Upon reset the snake is initialized at the center of our 5×4 grid: the head index is 12 (cell (2,2)), the length is set to 3, and the three body segments occupy the cells immediately to the left (indices 12, 11, 10).
On the first update_snake pulse (no eating), the head moves one step to the right (to index 13), the entire body shifts down by one slot, and the length remains at 3.
When we assert both update_snake and food_eaten, the head advances to index 14 and the body shifts again, but the length still stays at 3—this cycle simply captures the "grow" request in an internal flag.
Finally, changing direction to Down and pulsing update_snake once more moves the head to index 19 and, because the pending grow flag is now acted upon, increments the length to 4 while performing the usual body shift.

Screen Display →



Food_Generator:

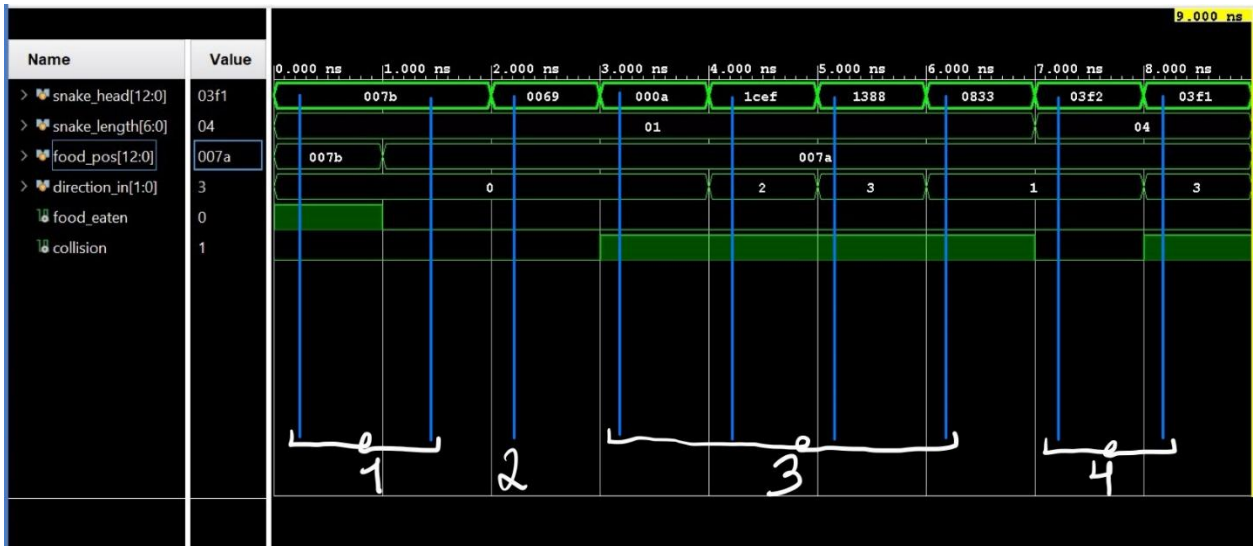



```
# run 1000ns
PASS: food_pos == 0 after reset
PASS: first food_pos = 56 (valid & avoids initial snake)
PASS: multi seg new food_pos = 227 (avoids old & {10,20,30,40,50})
passed all tests
$finish called at time : 85 ns : File "C:/Users/ASUS/Desktop/ˆ¶snake_codes/Food_Generator_tb.v" Line 111
```

1. **Reset Initialization**
    - On releasing rstn, food_pos returned to 0, proving the generator always starts from a known “no-food” state.
2. **First-Food Placement**
    - With the snake at its initial three segments {0,1,2}, pulsing food_eaten produced food_pos = 56.
    - Since 56 lies within the 0…7499 grid range and is not one of the occupied cells, the module correctly chose a legal, unoccupied location.
3. **Multi-Segment Collision Avoidance**
    - After marking cells {56,10,20,30,40,50} as occupied (the previous food plus five more), another food_eaten pulse yielded food_pos = 227.
    - Because 227 ∉ {56,10,20,30,40,50}, this confirms the LFSR+collision logic reliably skips *all* occupied positions.

Together, these self-checking steps demonstrate that **Food_Generator** (a) resets properly, (b) generates valid food positions on an initial non-empty snake, and (c) never collides with any segment of the snake body.

Collision_Detector:



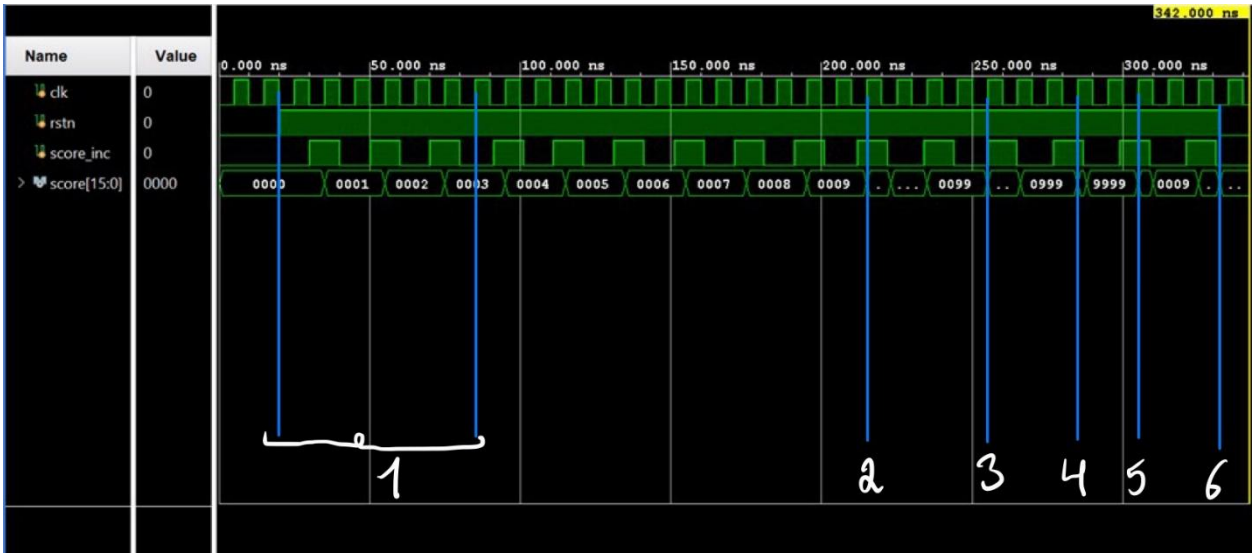

```
# run 1000ns
PASS: food_eaten asserted when head==food (head=123)
PASS: food_eaten deasserted when head!=food (head=123, food=122)
PASS: no collision for interior cell 105
PASS: top-wall collision detected (head=10)
PASS: bottom-wall collision detected (head=7407)
PASS: left-wall collision detected (head=5000)
PASS: right-wall collision detected (head=2099)
PASS: no self-collision for head=1010 vs body={1009,1008,1007}
PASS: self-collision detected when head=1009 vs body[1]=1009
passed all tests
$finish called at time : 9 ns : File "C:/Users/ASUS/Desktop/ٹ ٹsnake_codes/Collision_Detector_tb.v" Line 149
```

1. **Food-eat detection**
    - When snake_head == food_pos the bench saw food_eaten = 1. when they differed, food_eaten = 0.
      This proves the module correctly flags a food-eat event only on an exact match.
2. **Interior no-collision**
    - Placing the head at cell 105 (well inside the board) with no other segments yielded collision = 0.
      This confirms that the detector doesn't falsely report a collision when the snake is neither touching a wall nor itself.
3. **Wall collisions**
    - Head at 10 moving "up" (y=0) → top-wall collision
    - Head at 7407 moving "down" (y=74, GRID_H–1) → bottom-wall collision
    - Head at 5000 moving "left" (x=0) → left-wall collision
    - Head at 2099 moving "right" (x=99, GRID_W–1) → right-wall collision
      Each of the four edge cases correctly produced collision = 1, demonstrating reliable wall-hit detection for all directions.
4. **Self-collision**
    - A contiguous 4-segment snake at {1007,1008,1009,1010} produced no collision when the head was at 1010.

- For the next step, moving the head into 1009 (its own “neck”) set collision = 1. This verifies that the module only flags a self-collision when the head index exactly matches one of the existing body segments.

Taken together, these tests give full confidence that **Collision_Detector** (a) flags food-eaten events properly, (b) ignores harmless interior moves, (c) catches every wall hit, and (d) only reports self-collision when the head truly overlaps its own body.

Score_Counter:



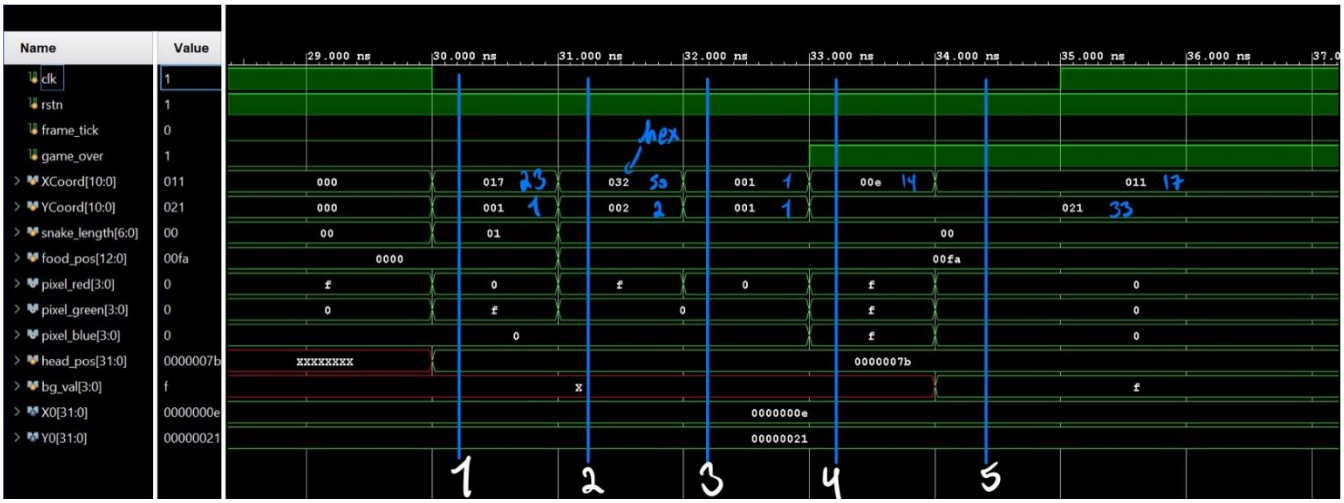

```
# run 1000ns
Step 1: score = 3 (expected 0003)
Step 2a: score = 9 (expected 0009)
Step 2b: score = 10 (expected 0010)
Step 3a: score = 99 (expected 0099)
Step 3b: score = 100 (expected 0100)
Step 4: score = 1000 (expected 1000)
Step 5: score = 0 (expected 0000)
Step 6 (reset): score = 0 (expected 0000)
$finish called at time : 342 ns : File "C:/Users/ASUS/Desktop/¶¶snake_codes/Score_Counter_tb.v" Line 76
```

1. **Basic Unit to Digit Counting (0000 -> 0001 -> 0002 -> 0003)**
   - Starting from reset (score = 0000), three score_inc pulses produce 0001, 0002 and 0003.
   - Confirms the ones digit properly increments on each pulse when no carry is needed.
2. **Ones to Tens Rollover (0003 -> 0009 -> 0010)**
   - Six additional pulses advance the count from 0003 up to 0009.
   - A seventh pulse rolls the ones digit from 9 to 0 and carries into the tens digit, yielding 0010.
   - Verifies that a carry-out from the ones place increments the tens place correctly.
3. **Tens to Hundreds Rollover (0098 -> 0099 -> 0100)**

- The counter is forced to 0098 and two more pulses produce 0099 then 0100.
- Confirms the tens place rolls over at 9 and carries into the hundreds digit.

4. **Hundreds to Thousands Rollover (0999 -> 1000)**
   - With the score forced to 0999, one pulse produces 1000.
   - Demonstrates that carries ripple all the way from hundreds into the thousands digit.
5. **Full to Scale Rollover (9999 -> 0000)**
   - Forcing 9999 and issuing one score_inc causes the counter to wrap around to 0000.
   - Validates the final carry from the thousands digit resets the entire 4-digit BCD value.
6. **Asynchronous Reset (X -> 0000)**
   - After generating additional pulses, driving rstn = 0 asynchronously clears score back to 0000.
   - Ensures the active-low reset always returns the counter to a known zero state regardless of prior value.

Snake_Renderer:




```
# run 1000ns
PASS: snake pixel at (23,1) = green
PASS: food pixel at (50,2) = red
PASS: background pixel at (1,1) = black
PASS: game-over background grayscale = f
PASS: game-over text grayscale inverse = 0 (bg=f)
passed all tests
$finish called at time : 35 ns : File "C:/Users/ASUS/Desktop/¶¶snake_codes/Snake_Renderer_tb.v" Line 156
```

1. **Normal Gameplay – Snake Pixel** (snake head at 123)
   With game_over = 0 and only one segment at flat index 123 (cell coordinates (23,1)), the renderer produced a **green** pixel (R=0, G=15, B=0) at that location. This confirms that any grid cell matching a snake segment is drawn in green.

2. **Normal Gameplay – Food Pixel**
   With no snake segments and food_pos = 250 (cell (50,2)), the renderer output a **red** pixel (R=15, G=0, B=0) at exactly that coordinate. This verifies that the food cell is displayed in red when the game is running.

3. **Normal Gameplay – Background Pixel**
   At a cell with neither snake nor food ((1,1)), the renderer drove the pixel to **black** (R=0, G=0, B=0). This demonstrates that empty cells default to a black background.

4. **Game-Over Screen – Background Color**
   Enabling game_over = 1 with the “off” state (fade=0), at any background position where the font ROM bit = 0, all three color channels matched ({R=15, G=15, B=15}), confirming a uniform grayscale fill for non-text areas.

5. **Game-Over Screen – Text Color**
   Still with game_over = 1 and the “off” state (fade=0), at a position where the font ROM bit = 1, the renderer again produced equal color channels but inverted relative to the background value confirming that letters pixels render as the inverse grayscale shade.

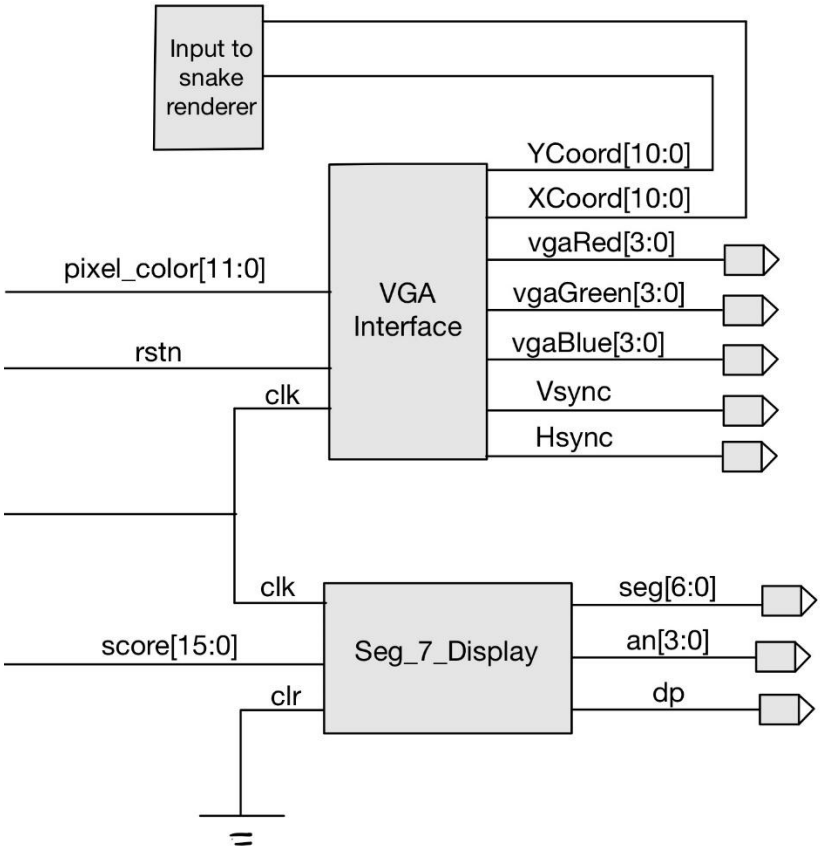
These five checks together validate both the normal rendering logic (snake, food, empty cells) and the Game-Over text’s blinking logic via correct grayscale fills and inversion based on ROM bits.

3. Output Stage
3.1 Description

• **VGA_Interface** generates the 800×600 72 Hz video signals from a 12-bit pixel bus. It samples pixel_color[11:0] each 100 MHz clock, drives horizontal/vertical counters to produce Hsync and Vsync, and outputs vgaRed[3:0]/vgaGreen[3:0]/vgaBlue[3:0] during the visible window (otherwise black). It also divides the raw pixel counters by 8 to yield 11-bit XCoord and YCoord for downstream rendering.

• **Seg_7_Display** multiplexes four seven-segment digits to show the 16-bit BCD score. A small refresh counter cycles through an[3:0]. on each active digit it selects the corresponding 4-bit nibble of score, looks up the 7-segment pattern, drives seg[6:0], and holds dp low.

3.2 Block Diagram



As shown above, the 12-bit pixel_color output from the Snake Renderer, together with rstn and clk, feeds into the VGA Interface, which generates 11-bit XCoord/YCoord back to the renderer

for address lookup and drives the 4-bit vgaRed, vgaGreen, vgaBlue channels plus Hsync and Vsync to the display. Meanwhile, the 16-bit score value, clocked alongside a clear input, goes into the Seg_7_Display module, which time-multiplexes and outputs the seven-segment patterns (seg[6:0]), digit enables (an[3:0]), and decimal point (dp).

3.3 Testbenches & Simulations

Both modules were tested in previous labs (lab 3, lab 6).

Game Over Display

When the Game_FSM asserts reset_game after a collision, Snake_Renderer disables normal snake/food pixels and enters its Game-Over rendering path, using a free-running 27-bit counter on the 100 MHz clock to generate a 4-bit fade level—no extra clock domains or flip-flops needed.

Meanwhile, for Game-Over fading we run:

$$free\_counter <= free\_counter + 1$$
$$fade \quad = free\_counter[26:23]$$

Fade step rate: each increment of fade occurs every 2^{23} clock cycles yielding:

$$\mathrm{f}_{\mathrm{fade}_{\mathrm{step}}} = \frac{100\mathrm{MHz}}{2^{23}} = 11.92\ [\mathrm{Hz}] \rightarrow 84\ [ms]$$

Full fade cycle: all 16 gray levels wrap around every 2^{27} clock cycles yielding:

$$\mathrm{f}_{\mathrm{fade}_{\mathrm{cycle}}} = \frac{100\mathrm{MHz}}{2^{27}} = 0.745\ [\mathrm{Hz}] \rightarrow 1.34\ [s]$$

Afterward we assign:

$$txt\_color = \{fade, fade, fade\}$$
$$bg\_color = \sim txt\_color$$

so text and background smoothly invert through 16 intensity steps.

To center “GAME OVER,” the renderer computes its text window in grid-cell units where each cell is 8×8 pixels. $TEXT\_W = 9 \times 8 = 72$ cells wide and $TEXT\_H = 8$ cells tall, and it calculates:

$$X0 = \frac{GRID_W - TEXT_W}{2} = 14$$

$$Y0 = \frac{GRID_H - TEXT_H}{2} = 33$$

Here, X0 and Y0 represent the top-left cell coordinates (column and row) of the centered text region on your 100×75 grid. For every cell whose (XCoord,YCoord) lies inside that 72×8 block it derives:

which of the 9 letters: $char\_idx = (XCoord - X0) / 8$

which of the 8 bitmap rows: $row\_idx = (YCoord - Y0)$
MSB-first bit in that row: $bit\_idx = 3'd7 - ((XCoord - X0)\ \%\ 8)$

These indices drive the combinational Font_ROM to output an 8-bit font_bits row for the selected letter. If font_bits[bit_idx] is 1, the pixel is drawn in the current fade color, otherwise it's drawn in the inverted fade background. Because the counter, coordinate math, ROM lookup, and color selection all occur in the 100 MHz domain, the "GAME OVER" message appears perfectly centered with a smooth fading effect and minimal resource overhead.

Key Parameters in the Top Module

- **GRID_W, GRID_H** → define the range of XCoord/YCoord and feed into centering math X0, Y0 (100X75 in our case).

- **TICKS_PER_FRAME** → sets frame_tick rate, driving update_snake, score_inc, reset_game and the renderer's blink (we chose 25'd5_000_000 which yielding $\mathrm{F_{frame}} = \frac{100\mathrm{Mhz}}{5000000} = 20\ [\mathrm{Hz}]$ for smooth snake movement).

- **MAX_LEN** → bounds the loop in the renderer's snake-hit test (snake_body_flat comparisons) and size of snake_length (set to 100 as specified).

- **POS_BITS** → determines the bit-width of snake_body_flat and the width of snake_length (chosen to cover all 100X75=7500 grid cells).

Utilization of Previous Labs

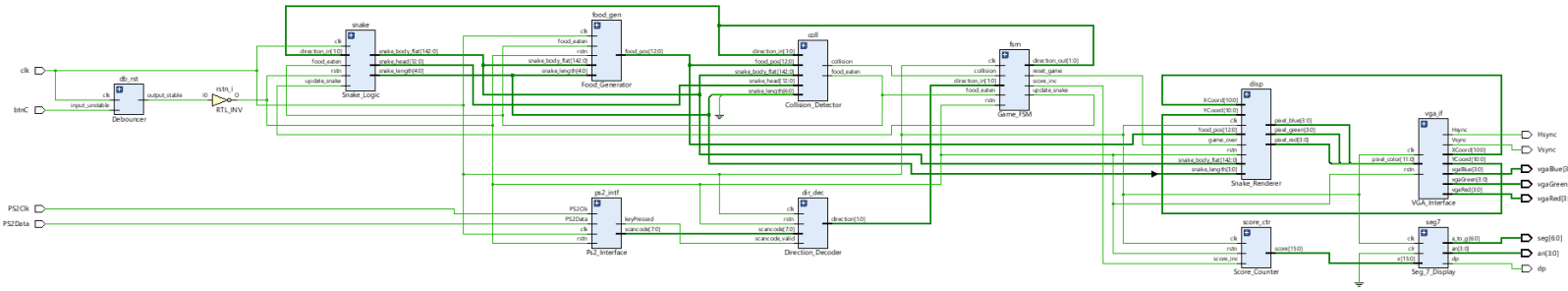
We reused and adapted several modules from earlier exercises:

- The **Debouncer** and **Seg_7_Display** from Labs 3,4 provided the reset filter and score display logic with minimal changes.
- The **PS2_Interface** from Lab 5 were integrated unchanged to handle keyboard input.
- Our **VGA Interface** come straight from Lab 6's VGA experiment.
- The **Lim_Inc** BCD counter from Lab 2 forms the backbone of our **Score_Counter**, cascading its ripple-carry behavior to four digits.

Results:

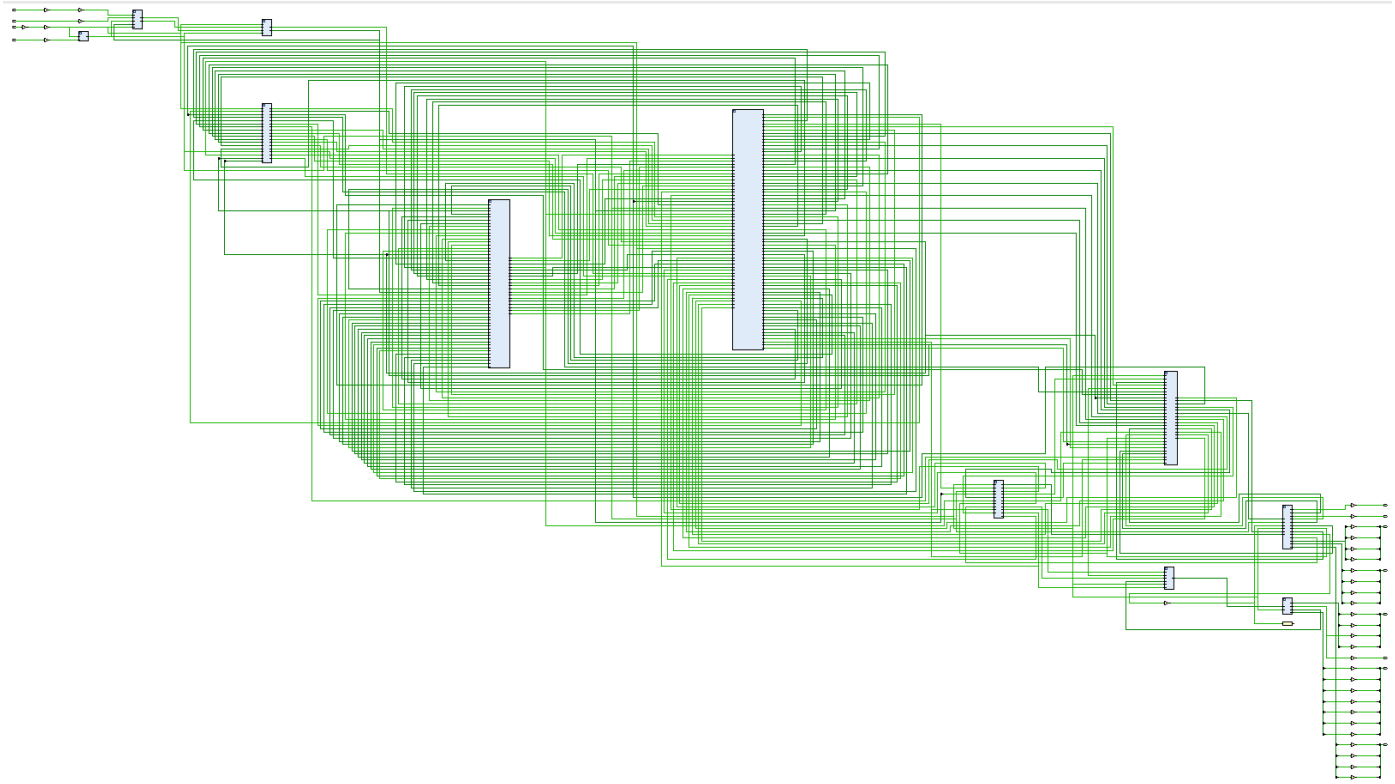
In its completed form the Snake game runs at a 20 Hz update rate on an 800 × 600 VGA display, with smooth head movement, correct body-shifting and growth upon eating food, and a four-digit BCD score that increments on each food event. Arrow-key input via PS/2 and 180° reversals are blocked. Collisions with walls or self-segments immediately assert reset_game, freezing the snake and switching to a centered "GAME OVER" message that smoothly fades through 16 gray-level steps, text intensity driven by the top four bits of a free-running 27-bit counter on the 100 MHz clock, with the background as its bitwise inverse. The 7-segment display holds the final score until the next reset.

Elaborated design schematic



The Vivado schematic matches our block diagram: inputs (Debouncer, PS2_Interface, Direction_Decoder) feed the **logic stage** (Game_FSM, Snake_Logic, Food_Generator, Collision_Detector, Score_Counter and Snake_Renderer), and their outputs (score, pixe_color) then drive the VGA_Interface and Seg_7_Display in the output stage.
The debounced, active-low rstn signal feeds into all modules reset ports.

Synthesis



At a high level the post-synthesis schematic looks densely packed, but zooming in reveals the IO buffers on every top-level pin—clock, rstn, PS/2, VGA and 7-segment lines guarding the interface between the FPGA fabric and the external world.

Implementing the design

Critical Warnings:

After implementation Vivado reported “methodology rule violations” because both **PS2Clk** (our PS/2 interface clock) and **pix_clk** (our VGA pixel clock) are being used directly to drive registers. In this design those two clock domains are cleanly isolated and operate exactly as intended, so these warnings are benign and do not impact functionality.

Design Timing Summary:

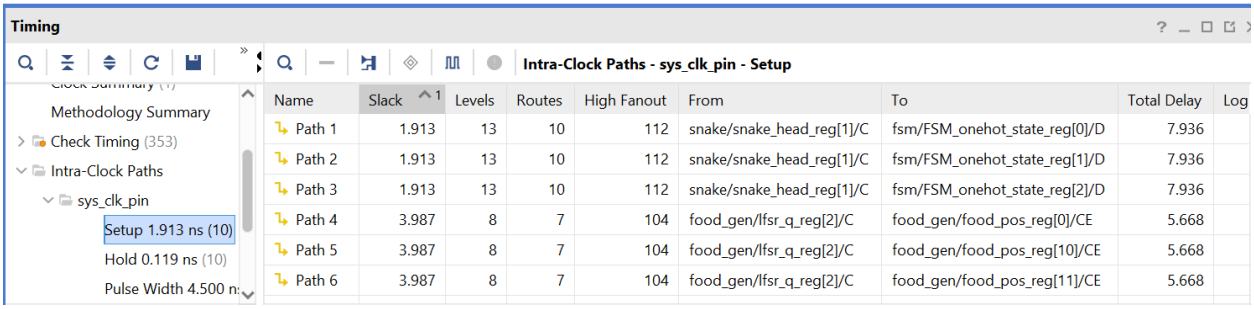
Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	1.913 ns	Worst Hold Slack (WHS):	0.119 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	4203	Total Number of Endpoints:	4203	Total Number of Endpoints:	1444

All user specified timing constraints are met.

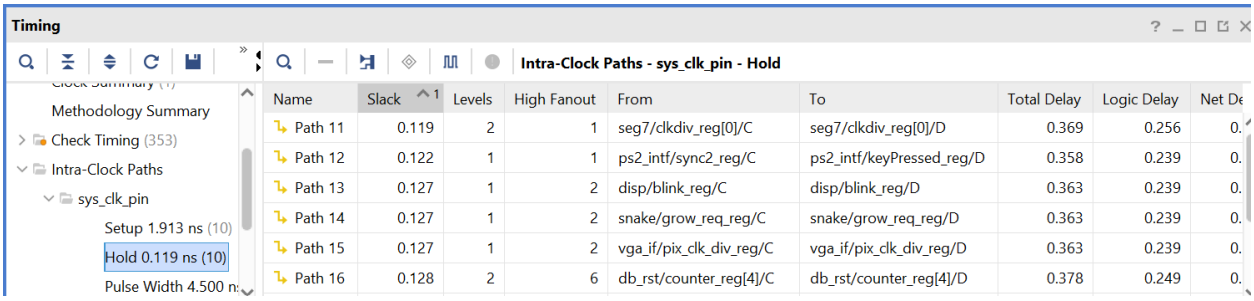
The timing summary reports a positive worst negative slack of 1.913 ns, a hold slack of 0.119 ns, and a pulse-width slack of 4.500 ns with zero failing endpoints, confirming that all timing constraints are met.

Setup:



The worst setup path runs from the snake_head_reg output in the Snake_Logic block into the one-hot state register in Game_FSM (e.g. snake/snake_head_reg[1]/C → fsm/FSM_onehot_state_reg[0]/D). Its total delay is about 7.936 ns, comfortably below the 10 ns period of a 100 MHz clock, yielding a positive slack of 1.913 ns and no setup violations.

Hold:



The worst hold path is internal to the Seg_7_Display's clock divider register (seg7/clkdiv_reg[0]/C → seg7/clkdiv_reg[0]/D), with a total delay of roughly 0.369 ns. With a hold slack of 0.119 ns, the design meets all hold timing requirements as well.

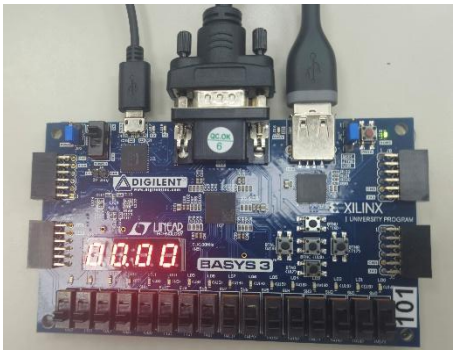
Setting up constraints

We assigned In the fpga_lab_constraints.xdc file only the I/Os we actually use: the 100 MHz system clock, the btnC reset button, the seven-segment display lines, the VGA HSYNC/VSYNC and RGB channels, and the PS/2 clock/data pair for keyboard input.

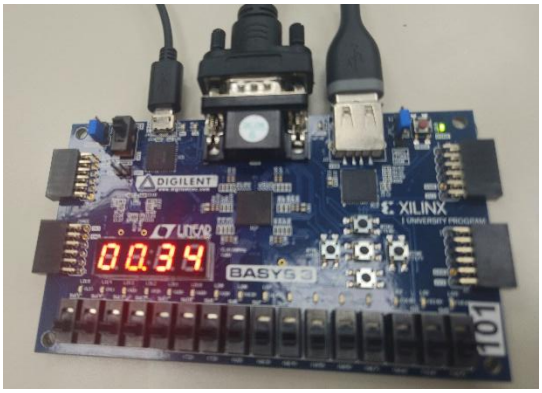
FPGA programming and debugging

We generated the bitstream file, successfully programmed the FPGA board, and verified the design through successful testing.

At the starting point:



right before collision:



After collision, GAME OVER, while holding the score: