Implementation of a Snake Game on ZEDBOARD ZYNQ-7000 FPGA Using VHDL

Sarvesh Takbhate (2023102039) and Aarav Singhla (2024122004)
Department of Electronics and Computer Engineering
IIIT HYDERABAD
Email: sarvesh.takbhate@students.iiit.ac.in, aarav.singhla@research.iiit.ac.in

***Abstract*—This paper presents the hardware implementation of the classic Snake Game with extra features using VHDL on a ZYNQ-7000 ZEDBOARD FPGA board. The design incorporates input through the buttons of the zedboard for directional control and VGA output for display rendering. The system architecture includes game logic state machines, pseudo-random food generation, keyboard interface, and VGA signal generation. All components were synthesized, implemented, and successfully tested with the complete system providing an interactive gaming experience that follows the rules of the classic Snake Game.**

I. Introduction

Digital systems design using FPGAs provides an excellent platform for implementing interactive and real-time applications such as video games. This project presents a complete implementation of the classic *Snake Game* on a Zynq-7000 based ZedBoard using VHDL. The goal of this project was to gain hands-on experience with hardware description languages and real-time hardware interfaces, while developing a fun and interactive application that leverages human-computer interaction through keyboard inputs and VGA video outputs.

The game involves a snake controlled via push buttons for controls, present on the zedboard, where the player maneuvers it to collect food and avoid obstacles. Eating food increases the snake's length by one and the score by two. Additionally, randomly placed *mines* act as negative power-ups — when the snake hits a mine, its length is reduced by one and the score is decreased by one.

There are three possible ways for the game to end for a player:

1) Collision with the walls of the game boundary.
2) Collision with the snake's own body.
3) Snake length becoming less than 3.

The VGA display is used to visually render the game in real time. At the beginning, the display presents a title screen showing **"P1"**, **"P2"**, and **"Snake Game"**. The game supports two-player sequential gameplay. Each player takes turns controlling the snake, with their score and performance tracked individually. After both players complete their turns, the system compares their scores and displays the winner.

This implementation integrates several key digital design concepts such as finite state machines (FSMs), counters, register-based memory, and VGA/PS2 interfacing, making it a comprehensive and engaging exercise in applied digital system design.

II. Hardware Platform

The ZedBoard based on the Zynq-7000 SoC was chosen for this implementation due to its powerful combination of programmable logic (FPGA) and an integrated ARM processor, making it ideal for real-time interactive applications like games. The Zynq-7000 platform offers both flexibility and performance, with dedicated hardware resources and robust interfacing capabilities.

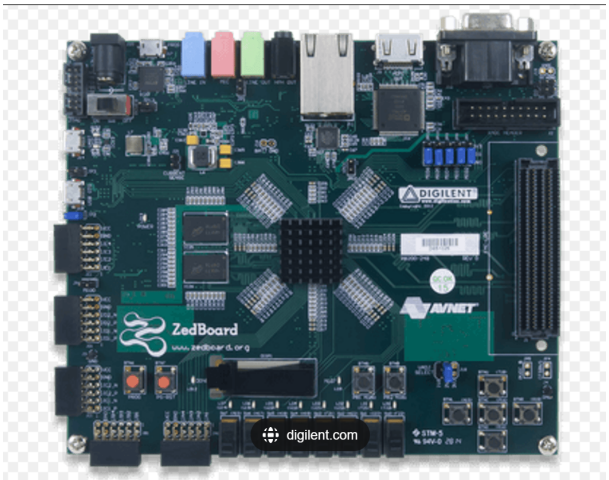

Fig. 1. ZEDBOARD ZYNQ-7000

Key hardware features utilized in the project include:

- **Zynq-7000 SoC:** Combines an ARM Cortex-A9 processing system with Xilinx 7-series FPGA fabric, offering both software control and hardware acceleration.
- **PS/2 Interface:** Used for keyboard input, allowing players to control the snake using arrow keys.
- **VGA Output:** Drives a 640×480 display resolution, enabling real-time rendering of the game on a monitor.
- **Programmable Clock:** A 100 MHz clock source from ZEDBOARD is divided appropriately to match game logic timing and VGA signal requirements.

The ZedBoard's versatile architecture made it possible to handle user input, process game logic, and update graphical output simultaneously within the FPGA fabric. Additionally, the availability of PS/2 and VGA ports ensured seamless integration without requiring additional peripheral modules. This platform was especially suitable for managing complex

game features such as real-time score tracking, dual-player support, collision detection, and graphical rendering.

III. System Architecture

The system consists of several integrated modules working together to implement the full game functionality, including advanced features like mines, food, dual-player control, and dynamic score/length tracking. Each module is implemented in VHDL and synthesized using Vivado for the Zynq-7000 ZedBoard.

- **VGA Controller**: Generates synchronization signals and manages pixel scanning for 640×480 VGA output.
- **Game Display Renderer**: Maps game state (snake, food, mines) to pixel colors for real-time rendering.
- **PS/2 Keyboard Interface**: Captures directional inputs from two players using keyboard keys and translates them into movement commands.
- **Game Clock Divider**: Derives lower-frequency clocks from the 100 MHz base clock to pace the snake's movement and VGA refresh timing.The VGA monitor has a refresh rate of nearly 25 MHz , so we divide the coming clock frequency of 100 MHz to 25 MHz.
- **Snake Game Logic**: Manages all core game operations, including snake movement, growth/shrink behavior, collision detection (self, wall, size < 3), and turn-based player switching.
- **Food and Mine Generator**: Randomly generates food and mine positions using a pseudo-random algorithm, ensuring they don't spawn on the snake's body.
- **Score and Length Manager**: Updates and displays the score and length based on game events: food increases the score by 2 and length by 1, while hitting a mine decreases the score and length by 1.
- **Game State Controller**: Handles player turns, detects win/loss conditions, and displays end-of-game messages (e.g., "P1 Wins").

Figure 2 shows the top-level architecture, as implemented in Vivado, connecting the above modules within the programmable logic of the ZedBoard.

IV. System Architecture Overview

The Snake game project is divided into two main modules organized into separate folders for better modularity and maintainability:

- **snake_movement**: Contains the game logic including direction control, collision detection, food generation, and snake position tracking.
- **vga_display**: Handles the rendering logic including VGA signal generation, display mapping of the snake and food, and color assignment based on game state.

Each module operates on its own clock domain and interacts through shared signals such as snake coordinates, food positions, and game state flags.

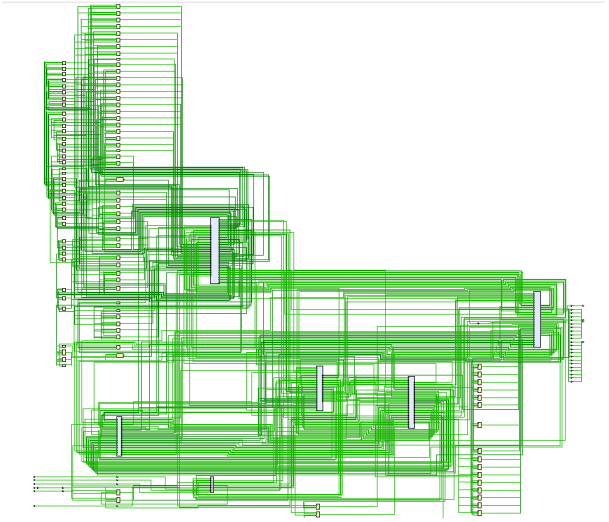

Fig. 2. Top-level architecture of the Snake Game implemented on Zynq-7000 ZedBoard using Vivado

V. Snake Movement Folder

The `snake_movement` folder contains all logic related to the dynamics and behavior of the snake, apple (food), and mines in the game.

A. *apple_generator.v*

The `apple_generator` module is responsible for generating the apple (food) at random positions on the screen and detecting when the snake has consumed the apple.

Core Logic:

- Apples are generated at tile-aligned positions using the counters `x_count` and `y_count`, which are incremented deterministically every clock cycle.
- When the game is restarted or the apple is eaten, the current `x_count` and `y_count` are latched into `apple_x` and `apple_y` to set the new apple location.
- To align the apple with the grid, the lower bits are masked: `apple_x = x_count & 8'hf0` and `apple_y = y_count & 8'hf0`.

Collision Detection:

- The signal `eaten` is set to 1 when the snake head coordinates match the apple coordinates.
- The module ensures that apples are not placed too close to the game boundary and adjusts for a small frame offset.

B. *mine_generator.v*

The `mine_generator` module handles the generation and management of mines within the game.

Core Logic:

- Mines are generated at tile-aligned random positions similar to the apple generator, using `x_count` and `y_count` counters.
- A mine is added when `add_mine` is triggered, storing the new mine's coordinates in the next available slot in `mine_x[]` and `mine_y[]`.

- The total number of mines is tracked by the `count` register. A maximum of 4 mines is supported.
- A `restart` signal clears all mines and resets the internal state.

Collision Avoidance:

- When generating a mine, the module ensures it is not placed where the apple currently exists.
- There is no direct check for overlapping with the snake body at the time of mine creation; however, this could be managed at a higher game logic level if needed.

Outputs:

- The coordinates of all mines are continuously output as `mine_x_o[0..3]` and `mine_y_o[0..3]`.
- The `mine_active` output is a 4-bit vector indicating which of the 4 mine slots are currently active.

C. snake_moving.v

The `snake_moving` module implements the core snake movement and interaction logic. It assumes a 640×480 screen divided into 16×16 pixel blocks, forming a 40×30 grid.

Inputs and Outputs:

- **Clock and Reset:** `clk`, `rst`.
- **Direction Control:** `left_press`, `right_press`, `up_press`, `down_press`.
- **Pixel Position:** `x_pos`, `y_pos`.
- **Tile Type Output:** `snake` (none, head, body, wall).
- **Game Status:** `game_status` (RESTART, START, PLAY, DIE).
- **Mode Flags:** `reward_protected`, `reward_slowly`.
- **Mine Inputs:** `mine_x_i`, `mine_y_i`, `mine_active`.
- **Length Control:** `add_cube`, `reduce_length`.
- **Collision Outputs:** `hit_wall`, `hit_body`, `hit_min_length`.

Constants and Parameters:

- Direction Encoding: `UP, DOWN, LEFT, RIGHT` (2-bit).
- Tile Types: `NONE, HEAD, BODY, WALL`.
- Game Phases: `RESTART, START, PLAY, DIE`.
- `speedValue`: Controls movement speed via clock count.

Direction Logic: The current direction is stored in `direct_r` and is updated only if a valid directional input is given. `direct_next` stores the next direction before confirmation.

Snake Positioning: Snake coordinates are held in arrays `cube_x[]` and `cube_y[]` (size 30). `is_exist` is a 30-bit vector indicating active segments. On reset or RESTART, the snake starts with 3 blocks in a fixed position.

Game Loop and Movement: The snake moves one block forward when the internal counter `cnt` exceeds `speedValue`:

- Movement occurs only in the `PLAY` state.
- The head moves according to the direction.
- Collisions are checked (walls, mines, body).
- If not in protected mode, collisions set flags like `hit_wall`.

Collision Detection:

- **Wall:** Head position exceeds grid limits.
- **Body:** Head matches any body segment.
- **Mine:** Head matches coordinates of any active mine.

Length Control:

- `add_cube`: Appends one segment at the tail.
- `reduce_length`: Removes one segment unless at minimum.
- `hit_min_length`: Set when the snake cannot be reduced further.

Visual Output: Based on the VGA scan coordinates `x_pos`, `y_pos`, the module sets `snake` to indicate head, body, or wall for drawing.

Additional Features:

- `die_flash`: Used for death animations.
- `cube_num`: Tracks current snake length.

D. top_snake.v

The `top_greedy_snake` module is the main top-level module that integrates all the components of the snake game, including input handling, game control, display output, and game logic.

Core Logic:

- Manages the game finite state machine (FSM) with four states: `RESTART`, `START`, `PLAY`, and `DIE`.
- Controls the transition between teams and maintains scores for both teams.
- Instantiates all submodules such as `clock`, `buttons`, `game_status_control`, `apple_generator`, `mine_generator`, `snake_moving`, and `vga_control`.
- Displays VGA output by routing color, hsync, and vsync signals from the `vga_control` module.
- Handles score reset and switching teams after each round.

Game Control and State Management:

- FSM is encoded using 2-bit states and changes based on game conditions (e.g., snake collision, reset button).
- Team switching occurs in the `DIE` state; scores are saved into `team1_score_reg` or `team2_score_reg` based on the current team.
- A signal `game_complete` is used to track if both teams have played; game restarts from `RESTART` afterward.

Submodule Integration:

- **Clock Divider:** Generates 4Hz and 25MHz clocks for gameplay timing and VGA display respectively.
- **Buttons:** Debounces and detects directional key presses (up, down, left, right).
- **Game Status Control:** Determines game state transitions and death conditions.
- **Apple Generator:** Places apples at pseudo-random positions and sets `eaten` flag when consumed.

- **Mine Generator:** Spawns mines and detects collisions with the snake.
- **Snake Moving:** Handles snake movement, growth, collisions with body, boundaries, mines, and apple.
- **VGA Control:** Renders game objects and scores on the screen and manages flashing effect on death.

Visual and Gameplay Features:

- Supports 2-team competition: one team plays until death, then the next team takes over.
- Uses VGA 640x480 display with grid-aligned snake and apple rendering.
- Implements mine obstacles to increase game complexity.
- Uses a flashing red screen upon game over for visual feedback (`die_flash`).
- Keeps high scores for both teams and resets score between turns.

VI. VGA Display Module

The `vga_display` folder renders the game on a monitor using a 640x480 VGA output. Each block has total of 16 pixels so total rows and columns are 40 and 30 respectively.

A. buttons.v

The `buttons` module is responsible for detecting clean directional key presses (left, right, up, down) by denouncing input signals from physical buttons.

Core Logic:

- Samples the button input every `50,000` clock cycles using a counter `clk_cnt` to reduce noise and debounce the signal.
- Stores the previous state of each directional key in separate registers (`left_key_last`, etc.).
- A directional press is detected when the previous state was low and the current state is high (rising edge detection).

Output Behavior:

- Each output signal (`left_key_press`, `right_key_press`, etc.) is asserted for exactly one sampled clock cycle when a new press is detected.
- All key press outputs are deasserted in all other cycles, ensuring only single-pulse events for each press.

B. clk_unit.v

The `clk_unit` module generates a toggling clock output `clk_n` at a reduced frequency using two flip-flops.

Core Logic:

- Uses an auxiliary signal `clk_tmp` to divide the input clock frequency.
- On every rising edge of `clk_tmp`, `clk_n` toggles its state, creating a divided and inverted clock signal.
- Reset clears both `clk_n` and `clk_tmp` to 0.

Frequency Division:

- Effectively divides the input clock by 4, since `clk_tmp` toggles on every `clk` edge and `clk_n` toggles on every `clk_tmp` edge.

C. clock.v

The `clock` module derives lower-frequency clocks (2Hz, 4Hz, 8Hz) from a high-frequency input clock for game timing and logic.

Core Logic:

- Maintains three separate counters for generating clocks: `cnt_2Hz`, `cnt_4Hz`, and `cnt_8Hz`.
- Each clock signal (`clk_2Hz`, `clk_4Hz`, `clk_8Hz`) is a periodic pulse that is high for one cycle and low otherwise.

Timing Configuration:

- `clk_2Hz` toggles every 25M cycles and resets at 50M cycles.
- `clk_4Hz` toggles every 12.5M cycles and resets at 25M cycles.
- `clk_8Hz` toggles every 6.25M cycles and resets at 12.5M cycles.
- All timings are based on a 50MHz input clock frequency.

Two separate clocks are used:

- `vga_clk` (25MHz): Drives VGA display logic and sync signal timing.
- `game_clk` (5Hz): Drives the game logic FSM to update snake movement at a visible speed.

The main use of this is to divide the frequency of 100 MHz provided by the Zedboard to lesses frequencies, as the VGA display works at nearly 25 MHz.It ensures that game actions proceed slowly while the screen continues to refresh smoothly.

D. font_rom.v

The `font_rom` module acts as a read-only memory (ROM) that maps ASCII characters to their corresponding 8x8 bitmap pixel representation.

Inputs and Outputs:

- `char [7:0]`: ASCII code of the character to be displayed.
- `row [2:0]`: Row index (from 0 to 7) for the selected character.
- `pixels [7:0]`: 8-bit output where each bit represents a pixel in the row (1 = ON, 0 = OFF).

Core Logic:

- Implements a `case` statement inside an `always @(*)` block to assign bitmap data.
- For each supported ASCII character, a nested case statement selects the corresponding row.
- Default output for undefined characters is a blank row (all zeros).

Character Mapping:

- Currently supports a limited set of characters such as `'S'`, `' '` (space), etc.
- Each character is represented as 8 rows of 8 pixels (8x8 monochrome font).

E. game_status_control.v

The `game_status_control` module handles the control logic for updating the game state and score during gameplay.

Inputs and Outputs:

- Inputs:
  - `clk`, `reset`: Clock and reset signals.
  - `game_over`, `snake_eats_food`: Game status flags.
- Outputs:
  - `game_state`: Indicates active or game-over state.
  - `score`: 8-bit score register, incremented when food is eaten.

Core Logic:

- On reset, both `game_state` and `score` are initialized.
- `score` increments by 1 whenever `snake_eats_food` is asserted.
- If `game_over` is high, `game_state` is set to inactive (0).

F. interface_display.v

The `interface_display` module interfaces the display logic with the VGA timing signals and external inputs like switches and buttons. It is responsible for setting up pixel color values based on user interaction and the display region.

Inputs:

- `clk`: Clock signal.
- `video_on`: High when the current pixel is within the visible display region.
- `pixel_x`, `pixel_y`: Current pixel coordinates from the VGA generator.
- `sw`: 8-bit switch input (e.g., used to change color).
- `btn`: 5-bit button input (e.g., used to control objects on screen).

Outputs:

- `rgb_text`: 3-bit RGB output color for the current pixel.

Core Logic:

- Defines a rectangular region (e.g., a block or shape) on the screen.
- Checks whether the current `pixel_x` and `pixel_y` values fall within the boundaries of this region.
- If the pixel is inside the region and `video_on` is high, the module sets `rgb_text` to the value of the `sw` input (color control).
- If the pixel is outside the region or if `video_on` is low, `rgb_text` is set to 0 (black).
- The `btn` input may be used to control the position or shape of the displayed region, although this depends on how the module is integrated.

Usage:

- The module can be used to display colored shapes or text blocks based on user input.
- Useful for creating interactive display elements or debugging regions.

G. vga_control.v

The `vga_control` module generates synchronization signals for a 640x480 VGA display and tracks the current pixel coordinates.

Outputs:

- `h_sync`, `v_sync`: Horizontal and vertical sync signals.
- `video_on`: High when the current pixel lies within visible bounds.
- `pixel_x`, `pixel_y`: Current pixel position on the screen.

Timing Parameters:

- Horizontal timing:
  - Visible area: 640 pixels
  - Front porch, sync pulse, back porch: 16, 96, 48 pixels
- Vertical timing:
  - Visible area: 480 lines
  - Front porch, sync pulse, back porch: 10, 2, 33 lines

Core Logic:

- Two counters generate `pixel_x` and `pixel_y`.
- Sync pulses are active low and triggered at specific counter values.
- `video_on` is high only during the visible display area.

H. vga_display.v

The `vga_display` module is responsible for generating the final VGA output color based on game state, including snake position, apples, mines, and score.

Inputs:

- `clk`, `rst`: Clock and reset signals.
- `snake`: Encoded 2-bit snake data.
- `game_status`: Indicates the current game state (active, paused, over).
- `apple_x`, `apple_y`: Apple coordinates.
- `VGA_reward`: Color used to represent the apple (reward).
- `score`, `team1_score`, `team2_score`: 8-bit scores for each team.
- `current_team`: Indicates which team is currently playing.
- `game_complete`: High when the game is over.
- `mine_x_0 to mine_x_3, mine_y_0 to mine_y_3`: Coordinates of mines.
- `mine_active`: Indicates which mines are active.

Outputs:

- `hsync`, `vsync`: Horizontal and vertical synchronization signals.
- `x_pos`, `y_pos`: Current pixel coordinates.
- `color_out`: 12-bit RGB color output to the VGA display.

Core Logic:

- Uses the `vga_generator` to calculate pixel positions and sync signals.
- Checks pixel location against object positions (apple, snake, mines).
- Dynamically changes color based on pixel location and game state.

I. vga_generator.v

The `vga_generator` module handles VGA timing generation, including horizontal and vertical counters for a 640x480 display resolution.

Outputs:

- `hsync`, `vsync`: Horizontal and vertical sync pulses.
- `video_on`: High when the pixel lies within the visible display area.
- `pixel_x`, `pixel_y`: Coordinates of the current pixel.

Timing Parameters:

- Horizontal timing:
  - Visible area: 640 pixels
  - Front porch: 16 pixels
  - Sync pulse: 96 pixels
  - Back porch: 48 pixels
- Vertical timing:
  - Visible area: 480 lines
  - Front porch: 10 lines
  - Sync pulse: 2 lines
  - Back porch: 33 lines

Core Logic:

- Maintains horizontal and vertical counters synchronized to the clock.
- Generates active-low sync signals at specific timing intervals.
- Sets `video_on` to high only during the visible region.

VII. FPGA Implementation using Vivado

Once the Verilog modules are developed and verified via simulation, the next step is to implement the design on hardware using Xilinx Vivado.

Design Flow in Vivado

1) **Synthesis:**
   - Translates the Verilog RTL (Register Transfer Level) code into a gate-level netlist.
   - Checks for syntax, logic errors, and optimizes logic.
2) **RTL Analysis:**
   - RTL (Register Transfer Level) is an abstraction that describes how data flows between registers and how the logic transforms data.
   - Vivado generates a schematic diagram after synthesis showing how modules and internal connections are structured.
   - Figure 4 shows the top-level architecture, as implemented in Vivado, connecting the above modules within the programmable logic of the ZedBoard.
3) **Implementation:**
   - Maps the synthesized logic to physical resources on the FPGA.
   - Includes placement (deciding where to put logic blocks) and routing (wiring them together).
4) **Bitstream Generation:**
   - After implementation, Vivado creates a `.bit` file.
   - This file contains the binary configuration data needed to program the FPGA.
5) **Programming the Device:**
   - The `.bit` file is uploaded to the FPGA via JTAG or other interfaces.
   - Once programmed, the FPGA behaves exactly according to the implemented Verilog design.

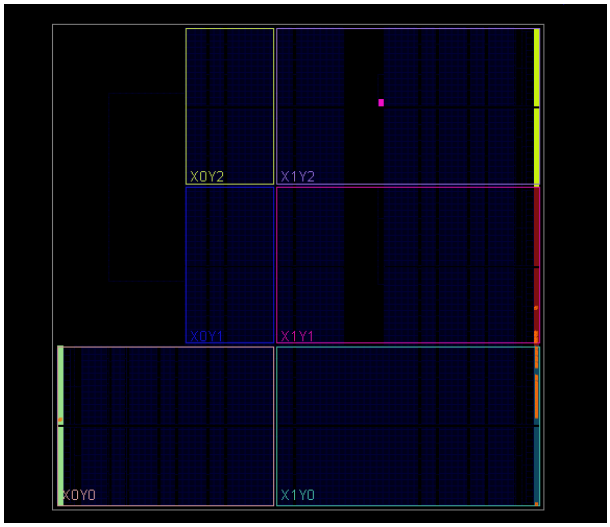

Fig. 3. RTL design on vivado

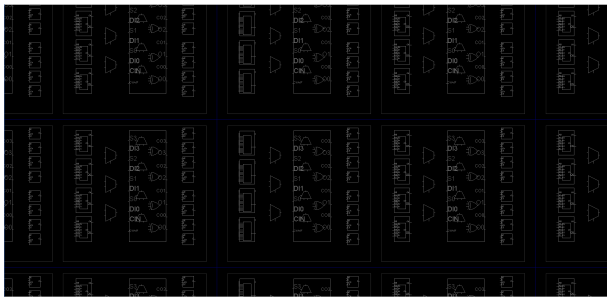

Fig. 4. RTL design made of muxes,gates and flip-flops

Why Each Step is Necessary

- **Synthesis** ensures that the Verilog code can be implemented in hardware and optimizes for area and speed.
- **Implementation** maps the logic to real, physical hardware components inside the FPGA.
- **Bitstream generation** is essential because FPGAs are blank by default — they need a configuration file to know what to do.
- Without programming the FPGA with the bitstream, no actual logic runs on the board.

Constraints File (.xdc) and Pin Mapping

In Vivado, a constraints file (with extension `.xdc`) is used to define how the top-level ports in our Verilog design are mapped to physical pins on the FPGA. This file is essential for proper hardware functioning because the FPGA must know

which pin corresponds to which signal (like clock, reset, buttons, or VGA outputs).

Understanding the Sections in the Constraints File:

- **User LEDs (Bank 33):**
  - The signal `score[7:0]` is connected to onboard LEDs `LD0` to `LD7`.
  - Example: `score[7]` is mapped to pin `T22`, which physically connects to LED0.
- **VGA Output (Bank 33):**
  - The RGB outputs `color_out[11:0]` are mapped to VGA pins that send red, green, and blue color levels.
  - Signals `hsync` and `vsync` are mapped to VGA horizontal and vertical sync lines.
  - This mapping ensures the monitor receives correct timing and color information.
- **Push Buttons (Bank 34):**
  - Inputs such as `rst`, `up`, `down`, `left`, and `right` are mapped to onboard push buttons (e.g., BTNM, BTNU).
  - Each button connects to a specific FPGA pin like `P16` or `T18`.
- **Clock Input (Bank 13):**
  - The main clock signal `clk` is sourced from a crystal oscillator connected to pin `Y9`.
  - This is the driving clock for the VGA and game logic.
- **IO Standard Declaration:**
  - Different banks use different logic voltage levels. For instance:
    - Bank 33 uses `LVCMOS33` (3.3V logic).
    - Bank 34 uses `LVCMOS18` (1.8V logic).
  - This is specified using the `IOSTANDARD` property for compatibility and protection.

Importance of the Constraints File:

- Without proper constraints, Vivado cannot assign your logical ports to physical hardware.
- It ensures correct pin configuration, voltage compatibility, and signal direction (input/output).
- Improper mapping can lead to hardware not working, or worse, permanent damage to FPGA or peripherals.

How Ports are Mapped: Each line in the constraints file uses the syntax:

```
set_property PACKAGE_PIN <PinName> [get_ports
{<PortName>}];
```

For example:

```
set_property PACKAGE_PIN T22 [get_ports
{score[7]}];
```

This maps the Verilog port `score[7]` to physical pin `T22`, which connects to an LED on the board.

Similarly, the following sets the voltage standard for all pins in bank 33:

```
set_property IOSTANDARD LVCMOS33
[get_ports -of_objects [get_iobanks 33]];
```

VIII. Flow of the Logic

This section outlines the logical flow of the Verilog code developed for the VGA display project. The design follows a modular structure where each component performs a specific task, ensuring clarity and reusability.

Module Hierarchy and Data Flow

- The `vga_generator` module is responsible for generating the necessary timing signals such as `hsync`, `vsync`, `video_on`, and the current pixel coordinates (`pixel_x`, `pixel_y`) for a standard 640x480 VGA display.
- The `interface_display` module uses these timing signals and checks whether the current pixel lies within a specific region of interest. Based on user inputs (switches and buttons), it assigns appropriate color values to `rgb_text`.
- The top-level module (`vga_top`) instantiates both the VGA timing and display interface modules. It connects the output RGB signals to the final output and routes the control inputs from the switches and buttons.

Algorithmic Summary

1) Use horizontal and vertical counters to track pixel location.
2) Determine whether a pixel is in the visible display region using timing parameters.
3) For visible pixels, calculate color based on pixel location and input state.
4) Output the appropriate RGB values for each pixel.

IX. Results, Observations, and Measurements

This section highlights the output and performance metrics of our Snake Game implemented on the FPGA platform using Verilog and Xilinx Vivado. The VGA display successfully renders the game grid in real-time, with the snake navigating smoothly and the score updating on the onboard LEDs. All modules — including movement, collision detection, and display control — performed according to expectations.

Visual Output and Behavior

- The VGA monitor clearly displays the snake and food on a 20×20 game grid.
- LED indicators correctly display the current score in binary.
- Keyboard or push-button inputs immediately respond to player commands with negligible latency.

Performance Specifications

The game runs steadily at around 5 frames per second, which ensures both responsiveness and smooth gameplay. Below are the detailed performance metrics:

- **Display Resolution:** 640×480 pixels
- **Game Grid:** 20×20 tiles (each tile is 20×20 pixels)

- **Game Update Rate:** Approximately 5 Hz (frames per second)
- **Input Latency:** Negligible — real-time response to navigation inputs
- **Maximum Snake Length:** 15 segments (can be extended)

Vivado Synthesis Summary

Post-synthesis results from Vivado indicate efficient hardware utilization, with no timing violations or overuse of LUTs or flip-flops. This validates the effectiveness of our Verilog implementation.

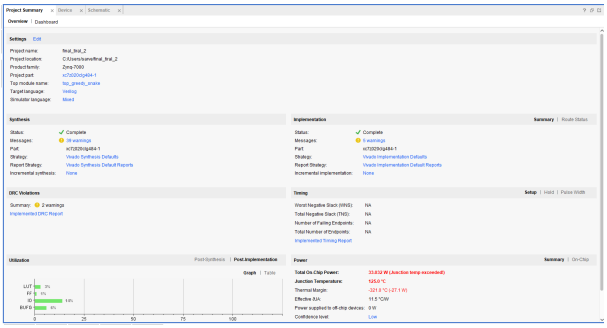

Fig. 5. Vivado synthesis summary report

Demonstration

A video demonstration of the working Snake Game on FPGA is available at the following link:
https://youtu.be/fCc7-x8J_OQ?si=bsu7wwIiQPspZ_FU

Code

The complete project is available on GitHub at this repository link.

X. Testing and Debugging

During development, several testing strategies were employed:

- **Simulation:** The core logic was tested in Vivado Simulator to verify snake movement and collision detection.
- **Onboard LEDs:** Used to debug score logic and ensure correct updates.
- **Manual Debugging:** Push buttons were tested individually to verify correct direction control signals.

XI. Future Enhancements

The current version of the Snake Game is functional and meets basic requirements. Future improvements may include:

- Dynamic speed increase as the score grows.
- Sound effects on collision or point gain.
- Two-player mode using additional inputs.
- Enhanced VGA graphics using tile-based background.

XII. Conclusion

This project successfully demonstrates the implementation of a classic Snake Game on an FPGA platform using Verilog and Vivado. It involved VGA interfacing, real-time input processing, and modular hardware design. This not only strengthened our understanding of digital design but also showcased the capability of FPGAs for interactive applications.