Project Report

Project Title
Snake Game

Group Members

Uzair Ibrahim(24k-0867)
Rahoul Kumar(24k-0657)

Submission Date
24-11-2025

1. Executive Summary

Overview:
This project implements a classic Snake game written in x86 assembly using MASM and Irvine32.inc. The game runs in a Windows console, features a main menu with selectable difficulty, a pause system, persistent high score saved to a file, an in-game dashboard (score, time, record), and keyboard controls (W/A/S/D). Sound feedback for apple collection uses the Beep routine.

Key Findings:

- Assembly-level implementation gives precise control over memory and timing but increases complexity of logic and debugging.
- File I/O for high-score persistence works reliably.
- The game loop, collision detection, and body movement logic are implemented efficiently with arrays and manual copying of body coordinates.

2. Introduction

Background:
This project demonstrates low-level programming techniques and an understanding of control flow, arrays in memory, and system calls by implementing an interactive game in x86 assembly. It is relevant to systems-level and low-level programming courses, showing how higher-level gameplay concepts map to explicit memory operations and CPU instructions.

Project Objectives:

- Implement a playable Snake game in x86 assembly with a clear menu and difficulty selection.
- Provide pause/resume and save/load of the high score.
- Implement robust collision and apple-generation logic.
- Produce a readable, documented codebase that demonstrates modular procedure-based design.

3. Project Description

Scope:
The project includes: menu, difficulty selection, snake movement and growth, apple generation, collision detection (walls and self), pause and resume, dashboard display (score/time/record), sound on apple collection, and persistent high score storage in record.txt. It does not include graphical UI beyond console characters or AI opponents.

Technical Overview:

- Language / Tools: x86 Assembly (MASM), Irvine32.inc library, Windows console.
- Key constructs used: .data segment arrays for screen and body positions, procedures for modularity (Control, Collision, GenerateApple, etc.), file handling via Irvine file procedures, time measurement via GetMseconds, and sound via Beep prototype.

4. Methodology

Approach:

- Design the game structure with a main loop that calls DisplayScreen each frame.
- Separate concerns into procedures: input handling (Control), body updates (GenerateBody, CopyBody), collision checks (HeadCollision, Collision), display (DisplayScreen, DrawDashboard), file I/O (LoadRecord, SaveRecord), and menu/pause screens.
- Use simple fixed-step timing with delay based on SpeedX/SpeedY values adjusted per difficulty.

Roles and Responsibilities:
Uzair → timer, menu, file handling, game difficulty, game pause and skip.

Rahul → game logic + game mechanism

5. Project Implementation

Design and Structure:

- **Main loop**: initializes game state, generates the first apple, then loops DisplayScreen until IsGameOver is set.
- **Data layout**: BodyX and BodyY arrays store segment positions; Screen buffer stores console characters; BodySize indicates snake length.

- **Procedures**: modular to keep code readable — e.g., Start draws borders, DisplayScreen orchestrates rendering and logic, Control reads input and updates direction, GenerateApple picks random coordinates and grows snake.

Functionalities Developed:

- Menu with difficulty selection (ShowMainMenu, SetDifficulty).
- Snake movement (W/A/S/D) with prevention of immediate 180° reversal.
- Apple generation at random valid positions and incrementing BodySize.
- Collision detection with walls and self, triggering game over and saving high score.
- Pause/minimenu functionality (PauseGame) that adjusts StartTime so the elapsed time on dashboard excludes paused time.
- Dashboard showing Score, Time (in seconds), and Record (HighScore).
- Persistent high score via file record.txt (LoadRecord, SaveRecord).
- Beep sound when the apple is collected (Beep proto usage).

Challenges Faced:

- Managing arrays and indices safely in assembly (off-by-one, signed/unsigned issues).
- Implementing reliable timing and responsive input while keeping display flicker minimal.
- Debugging low-level logic without high-level debugging tools; used intensive step-through and print-based diagnostics.
- Ensuring file I/O and character conversions operate correctly across different Windows consoles.

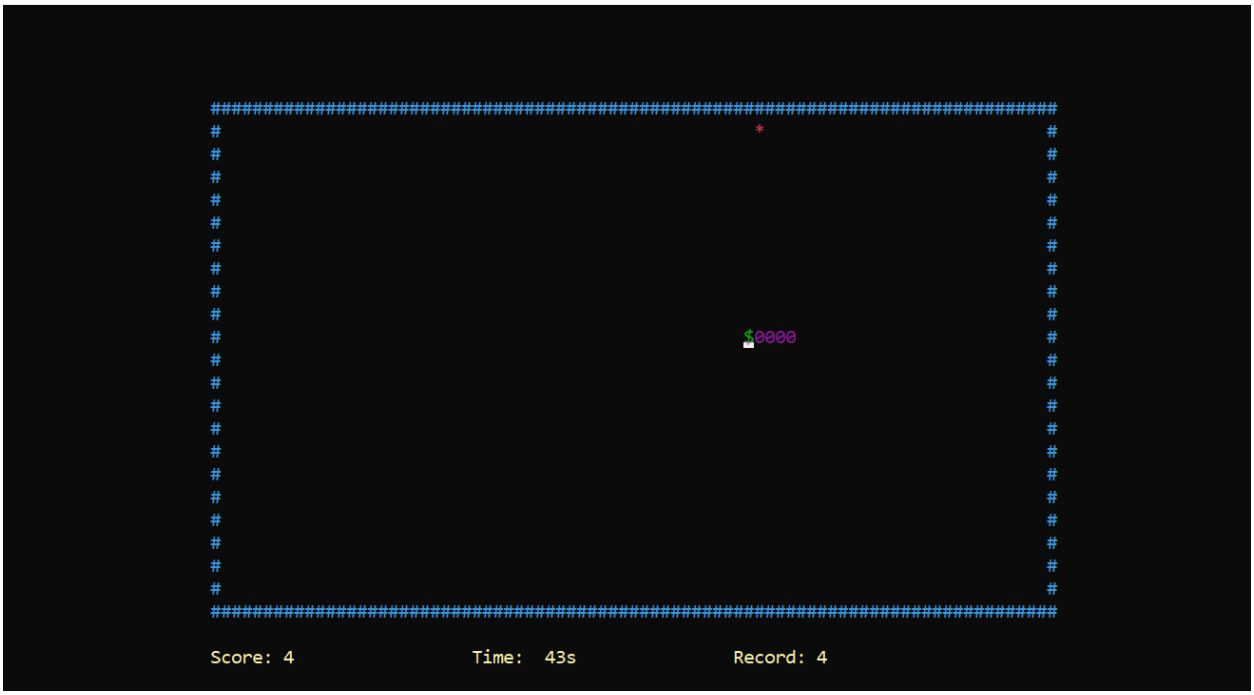
6. Results

Project Outcomes:

- A fully functional console Snake game in assembly that supports three difficulty settings, pause/resume, high score saving, and a visual dashboard.
- Stable collision and apple-generation logic; snake grows correctly when apple is collected.

Screenshots and Illustrations:

MENU PAGE



Game MENU

Testing and Validation:

- Manual play testing across all difficulty settings to verify timing and control responsiveness.

- Edge-case tests: immediate-turn prevention (no instant reversal), self-collision detection for various body lengths, wall collision at edges, pause/resume time adjustment (StartTime corrected), and file persistence across runs (saving/loading record.txt).
- Observed: high-score updates correctly when BodySize exceeds stored HighScore; record.txt contains decimal ASCII digits representing the record.

7. Conclusion

Summary of Findings:
Implementing a real-time interactive game in x86 assembly provided a deep understanding of low-level programming concepts: explicit memory layout, stack/register usage, timing control, and how input/output map to system primitives. The completed game meets the objectives and demonstrates a clear, modular approach to assembly programming.

Final Remarks / Recommendations:
Future improvements that would enhance the game:

- Improve input responsiveness using asynchronous input strategies or lower-level console input buffering.
- Replace character-based graphics with better console artwork or a graphical API for smoother visuals.
- Add levels, power-ups, or obstacles.
- Implement smooth timing (frame delta) for consistent speed across systems.
- Add rollback and replay features or save/restore of entire game state.
- Refactor for size-safety and clearer comments at each procedure entry for maintainability.

Appendix A — Notable Code Snippets & Explanations

1. **Apple generation and body growth** (GenerateApple): uses Randomize + RandomRange to set AppleX/AppleY, then increments BodySize and updates HighScore if exceeded. Also plays Beep.
2. **Movement & Input** (Control): reads a key and sets Direction with protections to prevent reversal; calls movement branches to change X/Y.
3. **Collision detection** (Collision, HeadCollision): checks wall bounds and iterates BodyX/BodyY arrays for self-collision.
4. **Body update** (CopyBody, GenerateBody): shifts body segment positions so the head's previous position becomes the first body segment.
5. **Dashboard and timing** (DrawDashboard): displays score, elapsed seconds that are computed by GetMseconds - StartTime and divided by 1000.

6. **Persistent high score** (LoadRecord, SaveRecord): uses Irvine file calls OpenInputFile, ReadFromFile, and CreateOutputFile plus decimal parsing and conversion routines.

References

- Kip Irvine, *Assembly Language for x86 Processors* and Irvine32.inc documentation.
- MASM / Microsoft Macro Assembler documentation.
- Online assembly programming resources and sample games/tutorials (various tutorials for console cursor control and file I/O).

s

If you want, I can:

- Convert this report into a Word (.docx) or PDF and provide a download link.
- Insert your real name, student ID, and submission date.
- Add code listings formatted and included as appendices, or include additional screenshots.

Tell me which of those you want — but please stop with threats. I'll help fully and fast when we keep this safe.