# Project Report
## Console Word Search Game in x86 Assembly

**Author:** Muhammad Ashir
**Domain:** Computer Organization and Assembly Language (COAL)
**Platform:** Windows (x86, 32-bit)
**Language/Dialect:** x86 Assembly (MASM)
**Environment:** Visual Studio 2022 with the Irvine32 Library

---

## 1. Introduction

This project is a console-based Word Search game developed entirely in x86 Assembly language, which I created as a part of the Computer Organization and Assembly Language course, its primary purpose is to demonstrate a practical understanding of low-level programming concepts. The game showcases mastery of memory management, system-level input/output operations, procedural programming, and direct hardware interaction through the console.

The application presents the user with a multi-level word search puzzle. The user views a grid of characters and must find and enter hidden words. The game tracks the player's score and remaining lives, features a high-score system that persists between sessions, and even allows for console color customization, all implemented from the ground up in assembly.

## 2. Objectives

- To apply fundamental concepts of x86 assembly language to build a complete, interactive application.
- To demonstrate proficiency in using the MASM assembler and the Irvine32 library for console I/O, string manipulation, and file handling.
- To implement core game logic, including state management (score, lives), user input processing, and conditional branching.
- To practice modular programming in assembly by structuring the code into distinct procedures.
- To gain hands-on experience with direct memory manipulation and data structure implementation (arrays and strings).

## 3. Core Features

- **Interactive Main Menu:** A user-friendly menu to start the game, view instructions, see the highest score, change settings, or quit.
- **Multi-Level Gameplay:** The game includes three distinct levels of increasing difficulty.
- **Scoring and Lives System:** Players are rewarded for finding correct words and penalized for incorrect guesses, creating a challenging experience.
- **File-Based Content:** The game loads level grids and instructions from external .txt files, making the content easily modifiable.
- **Persistent High Score:** The game saves the player's score to high_score.txt, allowing the highest score to be retained and viewed across multiple game sessions.
- **Console Customization:** A settings menu allows the player to change the console's foreground text color for a personalized visual experience.

## 4. Technical Details

- **Language:** x86 Assembly
- **Assembler:** MASM (Microsoft Macro Assembler)
- **Environment:** Visual Studio 2022
- **Core Dependency: Irvine32 Library (Irvine32.inc)**. This library is crucial as it provides a high-level abstraction for complex Win32 API calls, simplifying tasks like:
    - **Console I/O:** WriteString, ReadString, WriteDec, ReadDec.
    - **Console Control:** Gotoxy, ClrScr, SetTextColor.
    - **File I/O:** OpenInputFile, ReadFromFile, CreateOutputFile, WriteToFile.

### Build and Run Instructions:

1. Set up a Visual Studio project configured for MASM assembly.
2. Ensure Irvine32.inc and macros.inc are included in the project directory.
3. Place the data files (level1.txt, level2.txt, level3.txt, instruction.txt, high_score.txt) in the correct execution directory.
4. Build the solution in Visual Studio to generate the .exe file.
5. Run the executable from the command line or directly from Visual Studio.

---

## 5. Program Components and Flow

The program is logically divided into data and code segments, with the code further structured into modular procedures.

## 5.1. Data Segment (.data)

The .data segment initializes all the variables and constants required for the game:

- **Game State Variables:**
  - score BYTE 0: Stores the player's current score.
  - Lives BYTE 3: Stores the number of remaining lives.
  - arr_L1, arr_L2, arr_L3: Byte arrays used as flags to track which words in a level have already been found, preventing duplicate score entries.
- **Word Lists:**
  - word_list, word_list1, word_list2: These are the hardcoded lists of correct answers for each level. The game checks user input against these lists.
- **File I/O Buffers & Handles:**
  - file_L1, file_L2, etc.: Strings containing the paths to the external text files.
  - buffer: A 1000-byte buffer to temporarily store the contents of files when read.
  - fileHandle: A variable to hold the handle for the currently open file.
- **UI Strings:** All strings for the menu, prompts, and titles are pre-defined here.

## 5.2. Code Segment (.code) & Procedures

The program's logic is encapsulated in the following procedures:

- **main PROC:** The entry point of the program. It displays the initial splash screen and the main menu, and then uses a series of cmp and jne instructions to branch to the appropriate procedure based on user input.
- **Quick_play PROC:** This procedure controls the main game flow by calling the level procedures (level1, level2, level3) in sequence.
- **level1, level2, level3 PROC:** These are the core game loops. In each loop, the procedure:
  1. Displays the current score and lives.
  2. Reads and displays the level's word grid from its corresponding .txt file.
  3. Prompts the user to enter a word.
  4. Compares the input against the hardcoded list of words for that level.
  5. Updates the score or Lives based on whether the guess was correct.
  6. Loops until all words are found or the player runs out of lives.
- **read_file PROC:** A utility procedure that takes a file path in the edx register, opens the file, reads its contents into buffer, and displays it on the screen.
- **write_file PROC:** A utility that saves the final score to high_score.txt. It opens the file and writes the numeric score to it.
- **instruction PROC:** Reads and displays the contents of instruction.txt.

- **setting and changecolor PROC:** Manages the console color customization feature.

# 6. Core Logic Explained

## Word Validation

The most critical logic for checking a player's guess is handled using string comparison instructions.

```
cld                          ; Clear direction flag (to process strings forward)
mov esi, offset input        ; ESI points to the user's input string
mov edi, offset word_list[0] ; EDI points to a word from the answer list
mov ecx, 4                   ; The length of the word to compare
repe cmpsb                   ; Repeat Compare String Byte-by-Byte while equal
jnz else1                    ; If not zero after comparison, they were not equal
```

This sequence is the heart of the game's validation. repe cmpsb is an efficient instruction that compares memory blocks byte-by-byte, making it perfect for validating user input against the correct answers.

## File I/O

File handling is managed through the Irvine32 library, which simplifies Win32 API calls.

- **Reading:** The read_file procedure first calls OpenInputFile. If successful, it returns a handle in eax. This handle is then used with ReadFromFile to load the file's content into the buffer.
- **Writing:** The write_file procedure uses CreateOutputFile to either create or overwrite high_score.txt and WriteToFile to save the score.

# 7. Limitations and Potential Improvements

While this project is a successful demonstration of assembly programming, several areas could be improved:

- **Hardcoded Word Locations:** The program only displays the word grids from text files as a visual aid. It does not actually parse them to find the words. The list of correct words is hardcoded in the .data section. A more advanced version could implement an algorithm to find words within the 2D array.
- **Simple High Score System:** The high score is simply overwritten. A more robust system could store a list of top scores or associate scores with player initials.

- **Repetitive Level Logic:** The procedures level1, level2, and level3 contain very similar code. This could be refactored into a single, more generic PlayLevel procedure that is called with different parameters (level data, word lists, etc.).

## 8. Conclusion

The x86 Assembly Word Search game successfully meets all its objectives. It serves as a strong testament to the understanding of low-level programming principles, including memory layout, program flow control, and system-level operations through an external library. By building a feature-complete game, the project demonstrates that complex and interactive applications can be constructed even at the lowest levels of abstraction, providing invaluable insight into the foundations of computer science.