

1/13/2024

**PROJECT PROPOSAL**

**COAL-LAB**

**Submitted to:** MAAM MARIA ALI

**Submitted by:** Musharib Rehman [032]

Muhammad Rizwan [409]

Awais Anser [181]

Umair Noor [339]

Abdullah Rashid [079]

**“BANK MANAGEMENT SYSTEM”**

# INTRODUCTION

This assembly code is designed to simulate a basic banking system where users can log in, perform various transactions like depositing money, withdrawing money, transferring funds, checking balance, and leave the program. Users are prompted to enter their ID and password for authentication. The system has a predefined set of account IDs (IDS1 and IDS2) and corresponding passwords (PASSWORDS1 and PASSWORDS2). After successful authentication, users can choose from various banking operations presented in a menu.

# PROCEDURE

* **Data Segment:**

The DATA SEGMENT section declares various data structures, constants, and messages used in the program.

It includes account IDs (IDS1 and IDS2), corresponding passwords (PASSWORDS1 and PASSWORDS2), messages for user prompts and responses, and variables like ACCOUNT\_BALANCE, IDINPUT, PASSINPUT, and TRIES\_LEFT.

* **Code Segment:**

The CODE SEGMENT contains the main logic and program flow.

The program starts at the START label and initializes the data segment (MOV AX, DATA and MOV DS, AX).

The user is prompted to choose between login and signup options. If the user selects login, they need to enter their ID and password for authentication. If they choose signup, they provide a new ID and password.

* **User Authentication (Login):**

The program enters a loop labeled AGAIN where the user is prompted to enter their ID. The entered ID is compared with predefined account IDs (IDS1 and IDS2).

If a match is found, the program proceeds to ask for the corresponding password. The entered password is then compared with the predefined passwords (PASSWORDS1 and PASSWORDS2).

If the ID and password match, the user is granted access (ALLOWED\_MESSAGE), and the program proceeds to the main menu. Otherwise, the user may have multiple attempts (TRIES\_LEFT), and if exhausted, access is denied (DENY\_ACCESS).

* **User Authentication (Signup):**

In the signup process, the user is prompted to enter a new ID and password. The entered ID and password are then stored for future authentication.

* **Main Menu:**

After successful login or signup, the program displays a menu with various options such as depositing money, withdrawing money, transferring funds, checking balance, about us, references for project and leaving the program.

The user's choice is obtained using the SCAN\_NUM function, and the program proceeds to execute the chosen operation based on the user input.

* **Banking Operations:**

The program includes operations like depositing money (ADD\_MONEY), withdrawing money (WITHDRAW\_MONEY), transferring funds (TRANSFER\_MONEY), and checking balance (CHECK\_BALANCE).

Each operation involves user input, and the program performs corresponding calculations and updates the ACCOUNT\_BALANCE accordingly.

* **Error Handling:**

The program includes basic error handling. For example, if the user enters the wrong password multiple times, they may be denied access (DENY\_ACCESS). If there are insufficient funds during a fund transfer, an error message is displayed (TRANSFER\_ERROR).

* **Exit Program:**

The program includes a LEAVE\_PROGRAM option to exit the program gracefully.

* **Code Ending:**

The program ends with the END START statement.

# BENEFITS

**Educational Value:**

The code serves as an educational tool for learning assembly language programming, particularly for the x86 architecture. It provides a practical example that demonstrates essential concepts like input/output operations, control flow, and data manipulation in assembly language.

**Understanding Low-Level Operations:**

Assembly language allows programmers to interact directly with the hardware, gaining a deeper understanding of low-level operations. This can be valuable for those interested in systems programming, embedded systems, or understanding the internals of computer architecture.

**Simulation and Testing:**

The code is designed to run on the emu8086 emulator, which allows for simulation and testing of x86 assembly programs in a controlled environment. This can be useful for learners who want to experiment with assembly code without requiring physical hardware.

**Teaching Security Concepts:**

The code includes a simple user authentication mechanism, introducing the concept of secure user access. This can be a starting point for discussions on security practices, password protection, and potential vulnerabilities in real-world systems.

**Concepts of Banking Operations:**

The code demonstrates basic banking operations such as depositing money, withdrawing money, transferring funds, and checking balances. This can be useful for illustrating how assembly language can be applied to implement real-world functionality.

**Error Handling:**

The inclusion of error-handling mechanisms, such as limiting the number of password attempts and displaying appropriate messages for denied access, highlights the importance of error checking and handling in programming.

**Customizable for Learning:**

Learners can modify and expand upon the code to experiment with additional features, improve security measures, or add more sophisticated banking operations. This hands-on experience can enhance comprehension and problem-solving skills.

# LIMITATIONS

**Security Concerns:**

The code implements a basic user authentication system, but it lacks robust security measures. Real-world banking systems require sophisticated security mechanisms to protect user data, prevent unauthorized access, and handle encryption for secure communication.

**Lack of Input Validation:**

The code does not perform extensive input validation. In a production environment, input validation is crucial to prevent various types of attacks, such as buffer overflows, injection attacks, or unintended behavior due to invalid inputs.

**Limited Error Handling:**

While the code includes some error-handling mechanisms, it does not cover all potential error scenarios. Robust error handling is crucial in production systems to gracefully handle unexpected situations and provide meaningful feedback to users.

**Limited Functionality:**

The provided code covers only basic banking operations. Real-world banking systems require a more extensive set of features, such as account management, transaction logging, user profiles, and integration with other banking services.

**Platform-Specific Code:**

The code is specific to the emu8086 emulator and x86 architecture. Real-world applications need to be platform-independent and adhere to industry standards. Assembler code, in general, lacks portability across different architectures.

**Scalability Issues:**

The code may not scale well for a large number of users or transactions. Scalability is a crucial consideration for systems that need to handle a significant volume of users and transactions concurrently.

**Maintenance Challenges:**

Assembly code can be challenging to maintain and modify, especially as the complexity of the system increases. For large-scale projects, higher-level programming languages with more abstraction and maintainability are preferred.

**Limited User Interface:**

The user interface in this code is console-based and lacks the graphical interface commonly found in modern applications. Real-world banking systems often require user-friendly interfaces to accommodate a diverse range of users.

**Complexity:**

Assembly language, by its nature, is low-level and requires a deep understanding of hardware architecture. This complexity may pose challenges for developers, and modern high-level languages are often preferred for productivity and code readability.

# CONCLUSION

In conclusion, the provided assembly code for a simple banking system implemented in emu8086 serves as an educational tool, offering insights into x86 assembly language programming and foundational concepts. The code demonstrates user authentication, menu-driven interactions, and basic banking operations. However, it has notable limitations that make it unsuitable for real-world banking applications.

The key takeaways from this code are:

1. Educational Value:

The code provides a practical example for learners to understand assembly language concepts, such as low-level operations, user input/output, and control flow.

2. Limited Functionality: The code covers basic banking operations, but real-world banking systems require more sophisticated features, security measures, and scalability.

3. Security Concerns:

The code lacks robust security measures, input validation, and mechanisms for handling sensitive user data securely.

4. Platform Dependency:

It is specific to the emu8086 emulator and x86 architecture, making it non-portable and unsuitable for deployment in diverse computing environments.

5. Scalability and Maintenance Challenges:

Assembly code is difficult to scale for large user bases and complex functionalities, and it poses challenges for maintenance and modifications.

6. User Interface:

The console-based user interface is rudimentary, lacking the graphical interfaces commonly expected in modern applications.

7. Preparation for Further Learning:

The code can serve as a starting point for learners to delve deeper into assembly language or explore related topics in systems programming.

In real-world scenarios, the development of banking systems would be better approached using higher-level programming languages, industry-standard security practices, and adherence to regulatory requirements. The limitations of this code underscore the importance

E:\LGU\2nd Semester\Z-Extras\Endground.jpg