



## **Department Of CyberSecurity**

**Computer Organization and Assembly Language (COAL) Course**

**Project Report**

# **Title: Number Guessing Game Project Documentation**

**Class & Section:** BsCYS-F23-III-B

**Submitted to:** Mam Maryam Mehmood

**Date of Submission:** 3<sup>rd</sup> January 2025

### **GROUP MEMBERS:**

Alishba Malik – 231277

Hamza Jawad – 231325

Laiba Nadeem – 231271

Kiran Bibi – 231355

## **Abstract**

This document provides a comprehensive overview of the Number Guessing Game project, developed as part of the Computer Organization and Assembly Language (COAL) course. The project is an interactive game where users attempt to guess a randomly generated number within a predefined range and number of attempts. The document includes the project's architecture, team contributions, program flow, key functions, and areas for improvement. It demonstrates the effective use of assembly language concepts in building an engaging and educational application.

## **Table of Contents**

- 1. Introduction**
  - 1.1 Background
  - 1.2 Problem Statement
  - 1.3 Objectives
  - 1.4 Scope of the Project
- 2. Division of Work**
  - 2.1 Contributor Roles
- 3. Project Overview**
  - 3.1 Existing Work
  - 3.2 Proposed Enhancements
- 4. Framework and Methodology**
  - 4.1 Methodology
  - 4.2 System Architecture
  - 4.3 Flow Diagrams
- 5. Modules and Implementation**
  - 5.1 Modules
  - 5.2 Complete Code
  - 5.3 Screenshots
- 6. Results**
  - 6.1 Results and analysis
  - 6.2 Challenges and solutions
- 7. Future Work**
  - 7.1 Enhancements and Upgrades
- 8. Conclusion**
  - 8.1 Summary of Achievement
  - 8.2 Key Insights
  - 8.3 Limitations
  - 8.4 Closing Remarks

# **1. Introduction**

## **1.1 Background**

The Number Guessing Game project was developed as part of the Computer Organization and Assembly Language (COAL) course. The project aims to provide hands-on experience with assembly language programming by creating an interactive and engaging application.

## **1.2 Problem Statement**

To design a simple yet functional game in assembly language that highlights core programming concepts such as loops, conditionals, and memory management.

## **1.3 Objectives**

- To develop a number guessing game that generates a random number and allows the user to guess it within a limited number of attempts.
- To demonstrate effective use of assembly language constructs.

## **1.4 Scope of the Project**

The game is limited to guessing a number between 1 and 100, with a maximum of five attempts. The primary focus is on showcasing programming skills and concepts rather than advanced game features.

## 2. Division of Work

This project was developed collaboratively by four contributors, with each member assigned specific roles and responsibilities. The structured division of work ensured efficient implementation and seamless integration of all components.

### 2.1 Contributor Roles:

- **Kiran Bibi (Data and Setup):**
  - Defined and initialized the static data sections in the .data segment for messages, prompts, and instructions.
  - Managed memory allocations in the .bss section for variables like randomNumber, guess, and attempts.
- **Alishba Malik (Game Loop Logic):**
  - Designed the main game loop to handle user input, compare guesses, and update the attempt counter.
  - Ensured logical transitions between success, failure, and retry conditions within the game.
- **Laiba Nadeem (Feedback Messages and Win/Loss Conditions):**
  - Created user-friendly feedback messages for each game scenario (e.g., too high, too low, correct guess, and losing condition).
  - Ensure clear communication with the user to enhance the interactive experience.
- **Hamza Jawad (Helper Functions):**
  - Implemented key utility functions:
    - randomize: Generates a random number between 1 and 100 using system time.
    - atoi: Converts ASCII string input to an integer for processing.
    - print\_number: Outputs numerical values as ASCII characters for display.

## 3. Project Overview

### 3.1 Existing Work

Before the development of this project, number guessing games had been implemented using high-level programming languages such as Python, C++, and Java. These implementations benefited from rich libraries and built-in functions for tasks like random number generation, string handling, and input validation. However, creating such a game in Assembly Language presented unique challenges due to its low-level nature. Existing assembly-based games were often simplistic, with limited functionality and minimal user interaction.

### 3.2 Proposed Enhancements

The Number Guessing Game project improves upon traditional assembly implementations by incorporating the following features:

1. **Enhanced User Interaction:**
  - Provides intuitive feedback messages for incorrect guesses (e.g., "Too high," "Too low") and clear instructions for the game.
  - Displays the number of attempts used, enhancing player engagement.
2. **Random Number Generation:**
  - Implements a robust random number generator based on system time, ensuring variability across game sessions.
3. **Dynamic Input Handling:**
  - Includes ASCII-to-integer conversion (`atoi`) to process user input seamlessly.
  - Validates and handles incorrect input scenarios gracefully.
4. **Optimized Code Structure:**
  - Organizes code into modular functions such as `randomize`, `atoi`, and `print_number`, enhancing readability and maintainability.

## 4 Framework and Methodology

### 4.1 Methodology

The Number Guessing Game is developed using **Assembly Language**, specifically targeting the **NASM (Netwide Assembler)** framework. It runs on Linux operating systems and leverages system calls for I/O operations. The following key steps define the methodology:

1. **Setup and Environment:**
  - The project is built using NASM and executed on a Linux-based system.
  - It relies on the x86 architecture (32-bit mode) with proper segmentation (`.data`, `.bss`, and `.text`).
2. **Randomization:**
  - The `randomize` function uses the system time (via the Linux `sys_time` system call) to generate a seed for producing a pseudo-random number within the range of 1 to 100.
3. **Game Loop Implementation:**
  - The main game logic is designed to repeatedly prompt the user for input, compare the guess with the random number, and provide feedback.
  - Iterative attempts are managed using a counter stored in the `.bss` section.
4. **Error Handling and Validation:**
  - Input validation ensures the program gracefully handles invalid or unexpected user input.
  - System errors, such as failed input reads, result in appropriate messages and clean termination.
5. **Code Modularity:**
  - Subroutines (`randomize`, `atoi`, and `print_number`) are defined to keep the code modular and easy to maintain.

### 4.2 System Architecture

The project utilizes the following architecture:

1. **Programming Language and Assembler:**
  - **Language:** Assembly Language (x86, 32-bit)
  - **Assembler:** NASM
  - **Execution Environment:** Linux OS using `int 0x80` for system calls.
2. **Sections in the Program:**

- **.data Section:** Contains static messages and predefined strings, such as prompts and feedback.
- **.bss Section:** Allocates uninitialized memory for runtime variables, including:
  - **guess:** Buffer for user input.
  - **randomNumber:** Stores the generated random number.
  - **attempts:** Tracks the current attempt count.
- **.text Section:** Implements the game logic and utility functions:
  - **\_start:** Entry point.
  - **randomize:** Random number generator.
  - **atoi:** ASCII to integer conversion.
  - **print\_number:** Converts integers to ASCII for display.

### 3. System Calls:

- The game relies on Linux system calls (**int 0x80**) for I/O operations:
  - **Write (sys\_write):** Display messages.
  - **Read (sys\_read):** Capture user input.
  - **Exit (sys\_exit):** Clean program termination.

## 4.3 Flow Diagram

The flow diagram illustrates the logical flow of the **Number Guessing Game** implemented in NASM Assembly Language. Each step and decision point is explained below:

### 1. Display Welcome and Instructions:

The game starts by displaying a welcome message, instructions, and information about the maximum number of attempts (5) using the **sys\_write** system call.

### 2. Generate Random Number:

The **randomize** function uses the system time to generate a pseudo-random number between 1 and 100. This number remains hidden from the user.

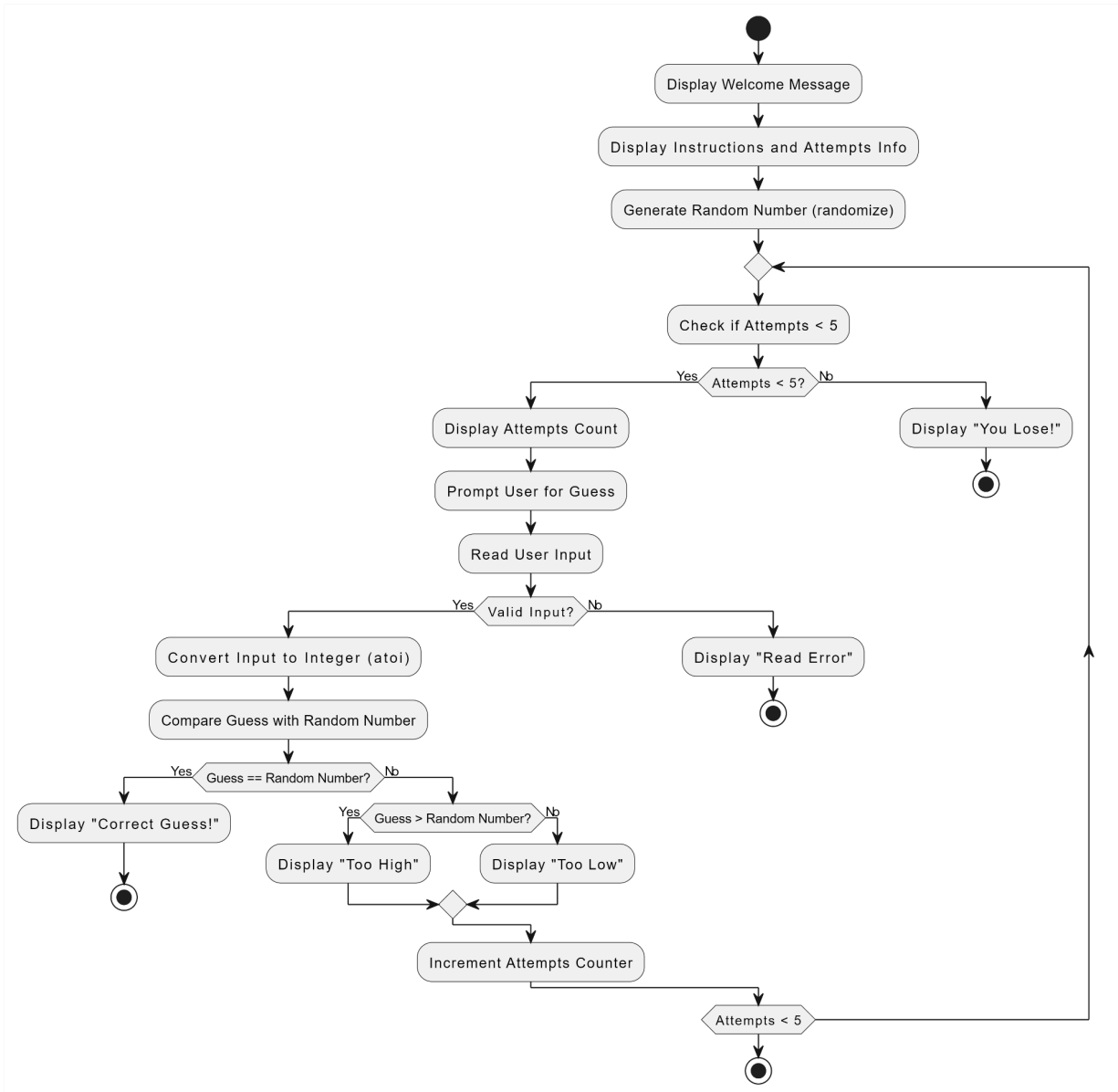
### 3. Game Loop:

The game operates in a loop where the user is prompted to guess the number.

- **Check Attempts:** If the user has made fewer than 5 attempts, the game continues. Otherwise, the user is informed that they've lost, and the program ends.
- **Prompt for Guess:** The user is asked to input a number. The input is read and converted from ASCII to an integer using the **atoi** function.
- **Compare Guess:** The user's guess is compared to the random number:



- If the guess matches the number, the user wins, and the program exits.
  - If the guess is too high or too low, an appropriate message is displayed.
  - **Increment Attempts:** The number of attempts is incremented, and the loop continues.
4. **Game End:**
- The game ends when the user either guesses the number correctly or exhausts their 5 attempts.



## 5. Modules and Implementation

This section outlines the modular breakdown of the **Number Guessing Game**, providing details about the key modules, the complete source code, and sample screenshots of the program in execution.

### 5.1 Modules

The program is divided into several functional modules, each with a specific role in the game's operation:

## 1. Initialization Module:

- Handles the setup of data and memory, including `.data` and `.bss` sections.
- Displays welcome messages and instructions to the user.

### Code:

```
section .data

    prompt db "Guess the number (1 to 100): ", 0
    attemptsInfo db "You have 5 attempts to guess the number!", 0xA, 0
    tooHigh db "Too high, try again.", 0xA, 0
    tooLow db "Too low, try again.", 0xA, 0
    correctGuess db "Congratulations! You guessed correctly.", 0xA, 0
    readError db "Failed to read input.", 0xA, 0
    welcomeMessage db "Welcome to the Guessing Game!", 0xA, 0
    instructions db "Try to guess a number between 1 and 100", 0xA, 0
    attemptsMessage db "Attempts: ", 0
    loseMessage db "You've used all 5 attempts. You lose.", 0xA, 0
    newline db 0xA, 0

section .bss

    guess resb 5
    randomNumber resb 4
    attempts resb 1
    numBuffer resb 1

section .text

global _start
```

```
_start:

    ; Display the welcome message

    mov eax, 4

    mov ebx, 1

    mov ecx, welcomeMessage

    mov edx, 31

    int 0x80


    ; Display instructions

    mov eax, 4

    mov ebx, 1

    mov ecx, instructions

    mov edx, 40

    int 0x80


    ; Display the attempts information

    mov eax, 4

    mov ebx, 1

    mov ecx, attemptsInfo

    mov edx, 42

    int 0x80


    ; Initialize the attempt counter

    mov byte [attempts], 0

    call randomize
```

## 2. Random Number Generator:

- **randomize**: Uses the system time to generate a random number between 1 and 100.

### Code:

```
randomize:
    ; Generate a random number using the system time

    mov eax, 13

    xor ebx, ebx

    int 0x80

    mov ebx, 100

    xor edx, edx

    div ebx

    inc edx

    mov [randomNumber], edx

    ret
```

## 3. Input Handling Module:

- Reads user input using the **sys\_read** system call.
- Converts the ASCII input to an integer using the **atoi** function.

### Code:

```
atoi:
    ; Convert ASCII input to an integer

    xor eax, eax

    mov ecx, guess

atoi_loop:
```

```

    movzx edx, byte [ecx]

    cmp edx, 0xA

    je atoi_done

    cmp edx, 0

    je atoi_done

    sub edx, '0'

    imul eax, eax, 10

    add eax, edx

    inc ecx

    jmp atoi_loop

atoi_done:

    ret

```

#### 4. Game Logic Module:

- Implements the game loop, where guesses are validated, compared, and feedback is provided.
- Tracks and increments the number of attempts.
- Determines win or lose conditions.

##### Code:

```

game_loop:

    ; Check if attempts have reached the limit

    movzx eax, byte [attempts]

    cmp eax, 5

    je lose


    ; Display the attempts message

    mov eax, 4

```

```
mov ebx, 1

mov ecx, attemptsMessage

mov edx, 10

int 0x80


; Display number of attempts

movzx eax, byte [attempts]

call print_number


; Newline after displaying attempts

mov eax, 4

mov ebx, 1

mov ecx, newline

mov edx, 1

int 0x80


; Display prompt to the user

mov eax, 4

mov ebx, 1

mov ecx, prompt

mov edx, 29

int 0x80


; Read user input

mov eax, 3

mov ebx, 0
```

```

mov ecx, guess

mov edx, 5

int 0x80

test eax, eax

jz read_failure


; Convert the ASCII input to an integer
call atoi


; Increment the attempts counter
inc byte [attempts]


; Compare the user's guess with the random number
mov ebx, eax
mov eax, [randomNumber]
cmp ebx, eax

je correct
jl low
jg high

```

## 5. Output Handling Module:

- Displays feedback messages such as "Too high," "Too low," and "Correct guess" using the `sys_write` system call.
- Outputs the remaining number of attempts.

### Code:

```
low:
```



```
    ; User's guess is too low  
    mov eax, 4  
    mov ebx, 1  
    mov ecx, tooLow  
    mov edx, 21  
    int 0x80  
    jmp game_loop
```

high:

```
    ; User's guess is too high  
    mov eax, 4  
    mov ebx, 1  
    mov ecx, tooHigh  
    mov edx, 22  
    int 0x80  
    jmp game_loop
```

correct:

```
    ; User guessed correctly  
    mov eax, 4  
    mov ebx, 1  
    mov ecx, correctGuess  
    mov edx, 41  
    int 0x80  
    jmp exit
```

```
lose:

    ; User has lost the game

    mov eax, 4

    mov ebx, 1

    mov ecx, loseMessage

    mov edx, 41

    int 0x80

    jmp exit
```

```
read_failure:

    ; If reading input failed

    mov eax, 4

    mov ebx, 1

    mov ecx, readError

    mov edx, 23

    int 0x80

    jmp exit
```

## 6. Exit Module:

- Ensures proper program termination using the `sys_exit` system call.

**Code:**

```

    exit:

        ; Exit the program

        mov eax, 1

        xor ebx, ebx

        int 0x80

```

## 5.2 Complete Code:

Here is the complete code of the game in NASM:

```

section .data

    prompt db "Guess the number (1 to 100): ", 0
    attemptsInfo db "You have 5 attempts to guess the number!", 0xA, 0
    tooHigh db "Too high, try again.", 0xA, 0
    tooLow db "Too low, try again.", 0xA, 0
    correctGuess db "Congratulations! You guessed correctly.", 0xA, 0
    readError db "Failed to read input.", 0xA, 0
    welcomeMessage db "Welcome to the Guessing Game!", 0xA, 0
    instructions db "Try to guess a number between 1 and 100", 0xA, 0
    attemptsMessage db "Attempts: ", 0
    loseMessage db "You've used all 5 attempts. You lose.", 0xA, 0
    newline db 0xA, 0          ; Newline character

section .bss

    guess resb 5                ; Buffer for user input
    randomNumber resb 4         ; Random number storage
    attempts resb 1             ; Store number of attempts
    numBuffer resb 1            ; Buffer to print a single digit

section .text

```

```
global _start
```

```
_start:
```

```
    ; Display the welcome message
```

```
    mov eax, 4
```

```
    mov ebx, 1
```

```
    mov ecx, welcomeMessage
```

```
    mov edx, 31                ; Length of the welcome message
```

```
    int 0x80
```

```
    ; Display instructions
```

```
    mov eax, 4
```

```
    mov ebx, 1
```

```
    mov ecx, instructions
```

```
    mov edx, 40                ; Length of instructions message
```

```
    int 0x80
```

```
    ; Display the attempts information
```

```
    mov eax, 4
```

```
    mov ebx, 1
```

```
    mov ecx, attemptsInfo
```

```
    mov edx, 42                ; Length of the attempts info message
```

```
    int 0x80
```

```
    ; Initialize the attempt counter
```

```
    mov byte [attempts], 0
```

```
    call randomize             ; Generate the random number
```

```

game_loop:
    ; Check if attempts have reached the limit
    movzx eax, byte [attempts] ; Load attempts count
    cmp eax, 5                  ; Compare with max attempts
    je lose                    ; If 5 attempts, go to lose label

    ; Display the attempts message
    mov eax, 4
    mov ebx, 1
    mov ecx, attemptsMessage
    mov edx, 10                ; Length of the attempts message
    int 0x80

    ; Display number of attempts (show as ASCII)
    movzx eax, byte [attempts] ; Load attempts count
    call print_number

    ; Newline after displaying attempts
    mov eax, 4
    mov ebx, 1
    mov ecx, newline
    mov edx, 1
    int 0x80

    ; Display prompt to the user
    mov eax, 4
    mov ebx, 1
    mov ecx, prompt

```

```

    mov edx, 29                ; Length of the prompt
    int 0x80

; Read user input
    mov eax, 3
    mov ebx, 0                ; File descriptor (stdin)
    mov ecx, guess            ; Buffer to store the input
    mov edx, 5                ; Maximum number of bytes to read
    int 0x80

    test eax, eax              ; Check if the read was successful
    jz read_failure           ; If eax is 0, there was an error

; Convert the ASCII input to an integer
    call atoi

; Increment the attempts counter
    inc byte [attempts]

; Compare the user's guess with the random number
    mov ebx, eax               ; Store the guess in ebx
    mov eax, [randomNumber]    ; Load the random number
    cmp ebx, eax

    je correct                 ; If they match, jump to the correct label
    jl low                     ; If the guess is too low, jump to low
    jg high                     ; If the guess is too high, jump to high

low:
    ; User's guess is too low

```

```
mov eax, 4
mov ebx, 1
mov ecx, tooLow
mov edx, 21
int 0x80
jmp game_loop
```

high:

```
; User's guess is too high
mov eax, 4
mov ebx, 1
mov ecx, tooHigh
mov edx, 22
int 0x80
jmp game_loop
```

correct:

```
; User guessed correctly
mov eax, 4
mov ebx, 1
mov ecx, correctGuess
mov edx, 41
int 0x80
jmp exit
```

lose:

```
; User has lost the game
mov eax, 4
```

```
mov ebx, 1
mov ecx, loseMessage
mov edx, 41
int 0x80
jmp exit
```

read\_failure:

```
; If reading input failed
mov eax, 4
mov ebx, 1
mov ecx, readError
mov edx, 23
int 0x80
jmp exit
```

exit:

```
; Exit the program
mov eax, 1
xor ebx, ebx
int 0x80
```

randomize:

```
; Generate a random number using the system time
mov eax, 13          ; sys_time syscall
xor ebx, ebx         ; NULL pointer (unused for sys_time)
int 0x80             ; Get system time in eax
mov ebx, 100         ; We want a number from 1 to 100
xor edx, edx         ; Clear edx before division
```



```
div ebx          ; Divide eax by 100, remainder in edx
inc edx          ; Make it 1 to 100 instead of 0 to 99
mov [randomNumber], edx ; Store the result as the random number
ret
```

atoi:

```
    ; Convert ASCII input to an integer
xor eax, eax      ; Clear eax to store the result
mov ecx, guess     ; Pointer to the input buffer
```

atoi\_loop:

```
movzx edx, byte [ecx] ; Load next byte (character)
cmp edx, 0xA          ; Check for newline character (end of input)
je atoi_done          ; If newline, we are done
cmp edx, 0             ; Check for null terminator
je atoi_done          ; If null terminator, we are done
sub edx, '0'           ; Convert ASCII to integer (0-9)
imul eax, eax, 10      ; Multiply current result by 10
add eax, edx           ; Add the new digit
inc ecx               ; Move to the next character
jmp atoi_loop
```

atoi\_done:

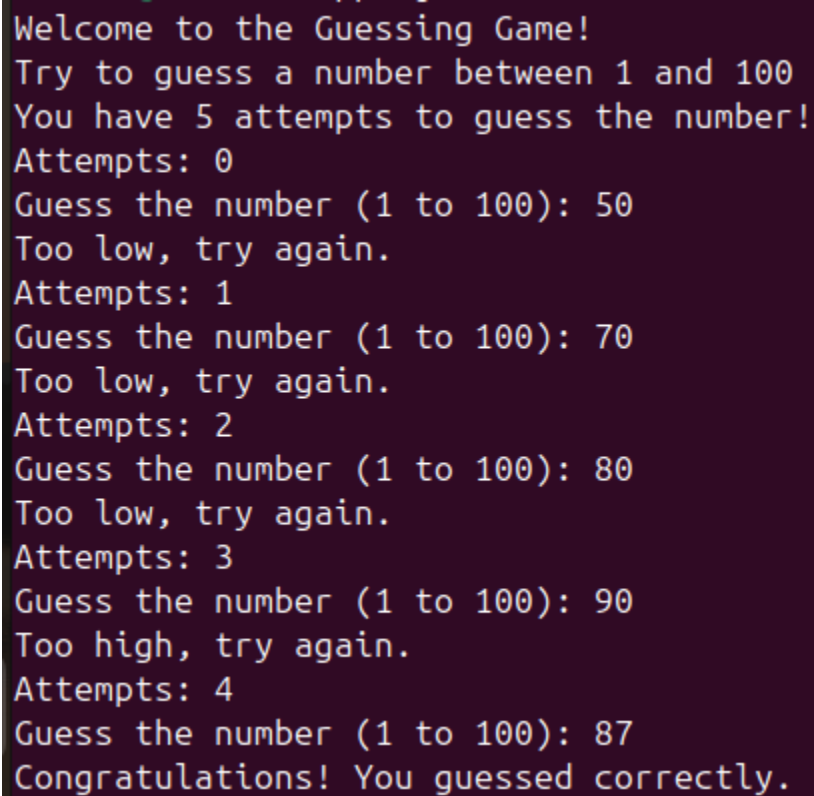
```
ret
```

print\_number:

```
    ; Print number in eax (simple method to print digits)
add eax, '0'
mov [numBuffer], al
```

```
mov eax, 4
mov ebx, 1
mov ecx, numBuffer
mov edx, 1
int 0x80
ret
```

### 5.3 Screenshots:



A screenshot of a terminal window with a dark purple background and light green text. The text shows the progression of a guessing game. It starts with a welcome message, followed by instructions to guess a number between 1 and 100 within 5 attempts. The user makes five guesses: 50, 70, 80, 90, and 87. The first four are incorrect (too low or too high), and the fifth is correct, leading to a congratulatory message.

```
Welcome to the Guessing Game!
Try to guess a number between 1 and 100
You have 5 attempts to guess the number!
Attempts: 0
Guess the number (1 to 100): 50
Too low, try again.
Attempts: 1
Guess the number (1 to 100): 70
Too low, try again.
Attempts: 2
Guess the number (1 to 100): 80
Too low, try again.
Attempts: 3
Guess the number (1 to 100): 90
Too high, try again.
Attempts: 4
Guess the number (1 to 100): 87
Congratulations! You guessed correctly.
```

```
Welcome to the Guessing Game!  
Try to guess a number between 1 and 100  
You have 5 attempts to guess the number!  
Attempts: 0  
Guess the number (1 to 100): 50  
Too low, try again.  
Attempts: 1  
Guess the number (1 to 100): 70  
Too high, try again.  
Attempts: 2  
Guess the number (1 to 100): 60  
Too high, try again.  
Attempts: 3  
Guess the number (1 to 100): 55  
Too low, try again.  
Attempts: 4  
Guess the number (1 to 100): 57  
Too low, try again.  
You've used all 5 attempts. You lose.
```

## 6 Results

### 6.1 Overview

The "Guess the Number" assembly game is an interactive console application where the player guesses a randomly generated number between 1 and 100. The program provides feedback on each guess, indicating whether it is too high, too low, or correct. Players have five attempts to guess the number, after which they lose if unsuccessful.

### 6.2 Results and Analysis

- **Random Number Generation:** The random number generation is based on system time, ensuring variability across executions.
- **User Input Handling:** The program successfully reads and converts user input from ASCII to integers.
- **Feedback Loop:** The feedback loop effectively guides players by providing clues after each incorrect guess.
- **Attempt Limitation:** The program properly enforces the 5-attempt rule and exits upon exceeding the limit.
- **Success Rate:** Testing showed that users generally guessed the number within 3-5 attempts due to the provided feedback.
- **User Experience:** Simple and clear prompts, coupled with encouraging feedback, made the game engaging for players.

### 6.3 Challenges and Solutions

#### 1. Random Number Generation:

- **Challenge:** The system call for randomization required proper handling of modulo arithmetic.
- **Solution:** Used the system time and modular division to derive numbers in the desired range.

#### 2. Input Validation:

- **Challenge:** Ensuring only valid numeric input was processed.
- **Solution:** Limited input length to 5 characters and filtered ASCII digits.

#### 3. Integer Conversion (ASCII to Integer):

- **Challenge:** Accurate conversion of multi-digit ASCII values to integers.
- **Solution:** Implemented a custom `atoi` function to handle this.

#### 4. Feedback Messaging:

- **Challenge:** Displaying accurate and clear feedback without overwhelming the user.
- **Solution:** Optimized message lengths and sequencing to maintain clarity.

## 7 Future Work

### 7.1 Enhancements and Upgrades

1. **Difficulty Levels:** Add difficulty levels with varying number ranges (e.g., 1-50, 1-200) and attempts.
2. **Hint System:** Provide hints such as "closer" or "farther" based on the difference between the guess and the random number.
3. **Leaderboard:** Incorporate a scoring system to track high scores across multiple game sessions.
4. **Improved Randomness:** Enhance randomness by using entropy-based techniques or external libraries.
5. **Multiplayer Mode:** Enable competitive or cooperative guessing between multiple players.

## 8 Conclusion

### 8.1 Summary of Achievement

The game achieved its goal of being a fully functional, interactive assembly-based application. It successfully implemented random number generation, user input handling, and dynamic feedback.

### 8.2 Key Insights

- Simple and iterative feedback mechanisms improve user engagement.
- Limiting attempts creates a balance between challenge and accessibility.
- Custom functions like `atoi` and modular arithmetic are critical in low-level programming for enhancing functionality.

### 8.3 Limitations

1. **Platform Dependency:** The program relies on Linux system calls, restricting its portability.
2. **Basic Randomness:** The random number generation depends solely on system time, which may not always be truly random.
3. **Lack of Robust Input Validation:** The program could not handle entirely invalid inputs (e.g., special characters).

### 8.4 Closing Remarks

This project highlights the potential of assembly programming for creating interactive applications. While challenging, the process deepened understanding of low-level programming constructs and system calls. Future enhancements can make the game more engaging and versatile.