Department of Computer Science

Bahria University Islamabad

PROJECT REPORT

Number guessing Game

Instructor: Ms. Nabia Khalid

Group Members:

Laiba Sajjad (01-136242-017)

Muhammad Uzair Attiq (01-136242-029)

Contents

Project Overview

The Enhanced Number Guessing Game is developed in **x86 Assembly Language** using **MASM** and the **Irvine32 library**. The project demonstrates low-level programming concepts while providing an interactive gaming experience.

Key features include:

- Generating a **random secret number** between 1 and 100
- Accepting user guesses via keyboard input
- Providing real-time feedback ("Too High", "Too Low", "Correct!")
- Optional **tutorial mode** for beginners
- Intelligent **hint system** (even/odd) after multiple wrong guesses
- Score calculation based on attempts and lives remaining
- Tracking session history for up to 20 games per user
- Maintaining high scores and win/loss statistics

This project emphasizes **core Assembly concepts**, such as registers, memory addressing, loops, conditional jumps, procedures, and bitwise operations, making it an ideal educational tool for COAL students.

Project Scope

Included Features

- Keyboard input handling using ReadInt
- Random number generation using RandomRange
- Input validation for numeric range (1–100)
- Tutorial mode with step-by-step instructions
- Hint system using bitwise operations (AND eax, 1)
- Score calculation: score = (100 - attempts) * lives remaining
- User session tracking with arrays for scores, attempts, and lives used
- Modular procedure-based program design
- Display of session history and high scores

Excluded Features

- Graphical user interface or animations
- Persistent storage of user data (scores reset after program ends)
- Multiplayer or network functionality
- Sound effects or audio feedback

Functional Requirements

Function	***Description***
Random number generation	Generates a secret number between 1–100 for each game session
User input handling	Prompts user for guesses and validates numeric range
Life-based gameplay	Player has 5 lives; each incorrect guess reduces one life
Hint system	Displays whether the secret number is even or odd after 3 wrong guesses
Score calculation	(100 - attempts) * lives remaining; only positive scores
User management	Add new users, select existing users, track statistics (wins/losses, high score, history)

Data Segment

The data segment contains all **variables, constants, messages, and arrays** used in the game:

Variables:

- secretNumber : stores the secret number
- attemptCount : counts the number of attempts in current game
- score : stores the player's score
- lifeCount : tracks remaining lives
- playAgain : flag for replaying the game
- tutorialMode : flag for enabling tutorial mode
- currentUserId : stores the selected user index

```
; per-game variables
attemptCount    DWORD 0
score           DWORD 0
playAgain       DWORD 0
tutorialMode    DWORD 0
lifeCount       DWORD MAXLIVES
secretNumber    DWORD 0
gameWonFlag     DWORD 0
```

Constants:

- MAXLIVES = 5 : maximum lives per game
- MAXUSERS = 10 : maximum users supported
- MAXGAMES = 20 : maximum games tracked per user
- NAMELENGTH = 20 : maximum characters per user name

```
; constants
MAXUSERS        = 10
MAXGAMES        = 20
NAMELENGTH      = 20
MAXLIVES        = 5         ; Maximum number of lives per game
```

Messages:

- Welcome, instructions, hints, correct/wrong messages
- Tutorial prompts
- User management messages (Add/Select user, menu options)

```
.data
    ; messages
    msgInvalidGuess   BYTE "Invalid input! Please enter a numeric guess (1-100).",0
    msgWelcome        BYTE "=== Number Guessing Game ===",0
    msgStartNew       BYTE "Generating secret number...",0
    msgEnterGuess     BYTE "Enter your guess (1-100): ",0
    msgHigh           BYTE "Too High! Try Again.",0
    msgLow            BYTE "Too Low! Try Again.",0
    msgCorrect        BYTE "Correct! You guessed the number!",0
    msgGameOver       BYTE "Game Over! You've run out of lives.",0
    msgSecretNumber   BYTE "The secret number was: ",0
    msgAttempts       BYTE " Number of attempts: ",0
    msgScore          BYTE " Your score: ",0
    msgPlayAgain      BYTE "Do you want to play another game? (1 = Yes, 0 = No): ",0
    msgInvalidInput   BYTE "Invalid input. Enter 1 for Yes, 0 for No.",0
    msgHighScoreNow   BYTE "Current high score: ",0
    msgUserHighScore  BYTE "Your high score: ",0
    msgNewHigh        BYTE "Congratulations! New high score!",0
    msgHintEven       BYTE "Hint: The number is even.",0
    msgHintOdd        BYTE "Hint: The number is odd.",0
    msgLivesRemaining BYTE "Lives remaining: ",0
    msgOutOfLives     BYTE "You are out of lives!",0
    msgLifeLost       BYTE "Life lost! ",0

    ; Tutorial messages
    msgTutorialPrompt BYTE "Do you want to enable tutorial mode? (1 = Yes, 0 = No): ",0
    msgTutorialContent BYTE "Tutorial: Guess the number between 1 and 100.",0
    msgTutorialContent2 BYTE "You have only 5 lives! Each wrong guess costs 1 life.",0
    msgTutorialContent3 BYTE "Your score = (100 - attempts) * lives remaining.",0
    msgTutorialContent4 BYTE "Hints will be provided after 3 wrong guesses.",0
```

Arrays/Structures:

- users[MAXUSERS] – stores user data, including name, scores, attempts, lives used, gameCount, highScore, wins, losses

```
; user database
users           USER MAXUSERS DUP(<>)
userCount       DWORD 0
currentUserId   DWORD 0FFFFFFFFh  ; Initialize as invalid
```

Code Segment & Procedures

Main Procedure

- Initializes the **random number generator** using Randomize
- Calls UserManager to handle menu and user selection
- Exits the program gracefully

UserManager

- Displays main menu:
    1. Add New User
    2. Select Existing User
    3. Start Game
    4. Exit
- Calls corresponding procedures based on user input

```
; UserManager - handles user selection/creation
UserManager PROC
MenuLoop:
    call Clrscr
    mov edx, OFFSET msgUserManager
    call WriteString
    call CrLf
    call CrLf

    mov edx, OFFSET msgAddUser
    call WriteString
    call CrLf
    mov edx, OFFSET msgSelectUser
    call WriteString
    call CrLf
    mov edx, OFFSET msgStartGameOpt
    call WriteString
    call CrLf
    mov edx, OFFSET msgExitUserMgr
    call WriteString
    call CrLf
    call CrLf

    mov edx, OFFSET msgChoice
    call WriteString
    call ReadInt
```

AddUser

- Adds a new user to the users array
- Initializes statistics (gameCount, highScore, gamesWon, gamesLost)
- Validates that maximum user limit is not exceeded

```
; AddUser - adds a new user
AddUser PROC
    mov eax, userCount
    cmp eax, MAXUSERS
    jl CanAddUser

    ; Max users reached
    call CrLf
    mov edx, OFFSET msgInvalidInput
    call WriteString
    call CrLf
    mov edx, OFFSET msgPressAnyKey
    call WriteString
    call ReadChar
    ret
```

SelectUser

- Displays all existing users with their statistics
- Allows selection by user number
- Updates currentUserId for gameplay

```
; SelectUser - displays and selects existing user
SelectUser PROC
    mov eax, userCount
    cmp eax, 0
    jg ShowUsers

    ; No users exist
    call CrLf
    mov edx, OFFSET msgNoUsers
    call WriteString
    call CrLf
    mov edx, OFFSET msgPressAnyKey
    call WriteString
    call ReadChar
    mov currentUserId, 0FFFFFFFFh
    ret
```

StartGame

- Checks if a user is selected
- Displays tutorial instructions if enabled
- Calls GuessGame to execute gameplay
- Displays session history after each game
- Prompts user for replay

```
; StartGame - starts the game if a user is selected
StartGame PROC
    ; Check if a user is selected
    mov eax, currentUserId
    cmp eax, 0FFFFFFFFh
    jne UserSelected

    ; No user selected
    call CrLf
    mov edx, OFFSET msgNoUserSelected
    call WriteString
    call CrLf
    mov edx, OFFSET msgPressAnyKey
    call WriteString
    call ReadChar
    ret
```

GuessGame

- Resets game variables (ResetGame)
- Generates a random secret number (GenerateNumber)
- Game loop:
    1. Prompt user for a guess (GetUserGuess)
    2. Compare guess with secret number (CheckGuess)
    3. Display hints after 3 wrong guesses
    4. Update lives and score
    5. Repeat until correct guess or lives reach zero
- Update user statistics (wins/losses, high score)
- Store session data (attempts, score, lives used)

```
; GuessGame - main game procedure with limited lives
GuessGame PROC USES ebx esi edi
    LOCAL localLifeCount:DWORD
    LOCAL localSecretNumber:DWORD

    ; Get current user's data
    mov eax, currentUserId
    cmp eax, 0FFFFFFFFh
    je EndGame  ; No user selected

    ; Calculate user offset
    mov eax, TYPE USER
    mov ebx, currentUserId
    mul ebx
    mov esi, OFFSET users
    add esi, eax
```

GenerateNumber

Generates a random number in the range 1–100 using RandomRange

```
; GenerateNumber: returns random number 1..100
GenerateNumber PROC
    mov eax, 100
    call RandomRange
    inc eax
    ret
GenerateNumber ENDP
```

GetUserGuess

- Reads user input via ReadInt
- Validates numeric input and ensures it is within 1–100

```
; GetUserGuess - prompt & read int (1-100) with validation
GetUserGuess PROC
ReadGuess:
    mov edx, OFFSET msgEnterGuess
    call WriteString
    call ReadInt
    jno CheckRange  ; Check for overflow

InvalidInput:
    mov edx, OFFSET msgInvalidGuess
    call WriteString
    call CrLf
    jmp ReadGuess

CheckRange:
    cmp eax, 1
    jl InvalidInput
    cmp eax, 100
    jg InvalidInput
    ret
GetUserGuess ENDP
```

CheckGuess

- Compares the user’s guess with the secret number
- Returns 0 if correct, 1 if incorrect
- Displays feedback: “Too High” or “Too Low”

```
; CheckGuess(secret, guess, lifeCountPtr) - returns 0 if correct, 1 if wrong
CheckGuess PROC USES ebx esi, secretNum:DWORD, guess:DWORD, lifeCountPtr:DWORD
    mov eax, guess
    mov ebx, secretNum
    cmp eax, ebx
    je Correct

    ; Wrong guess - show message based on high/low
    ja TooHigh

    ; Too low
    mov edx, OFFSET msgLow
    call WriteString
    mov edx, OFFSET msgLifeLost
    call WriteString
    call CrLf
    mov eax, 1
    ret
```

PrintHistory

- Displays the complete game history for the current user
- Shows game number, status (Won/Lost), attempts, lives used, and score

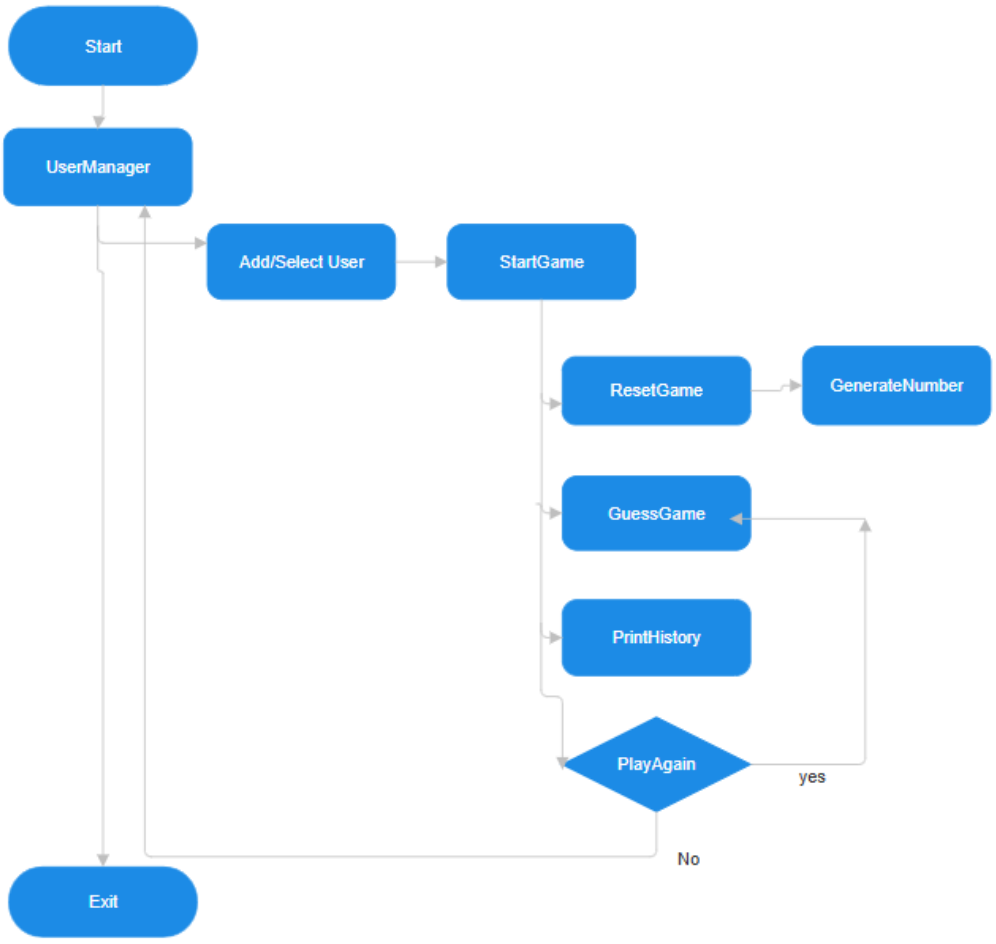
```
; PrintHistory - displays game history for current user
PrintHistory PROC USES esi
    ; Check if user is selected
    mov eax, currentUserId
    cmp eax, 0FFFFFFFFh
    je NoHistory

    ; Calculate user offset
    mov eax, TYPE USER
    mov ebx, currentUserId
    mul ebx
    mov esi, OFFSET users
    add esi, eax

    mov eax, (USER PTR [esi]).gameCount
    cmp eax, 0
    je NoHistory

    call CrLf
    mov edx, OFFSET msgWelcome
    call WriteString
    call CrLf
```

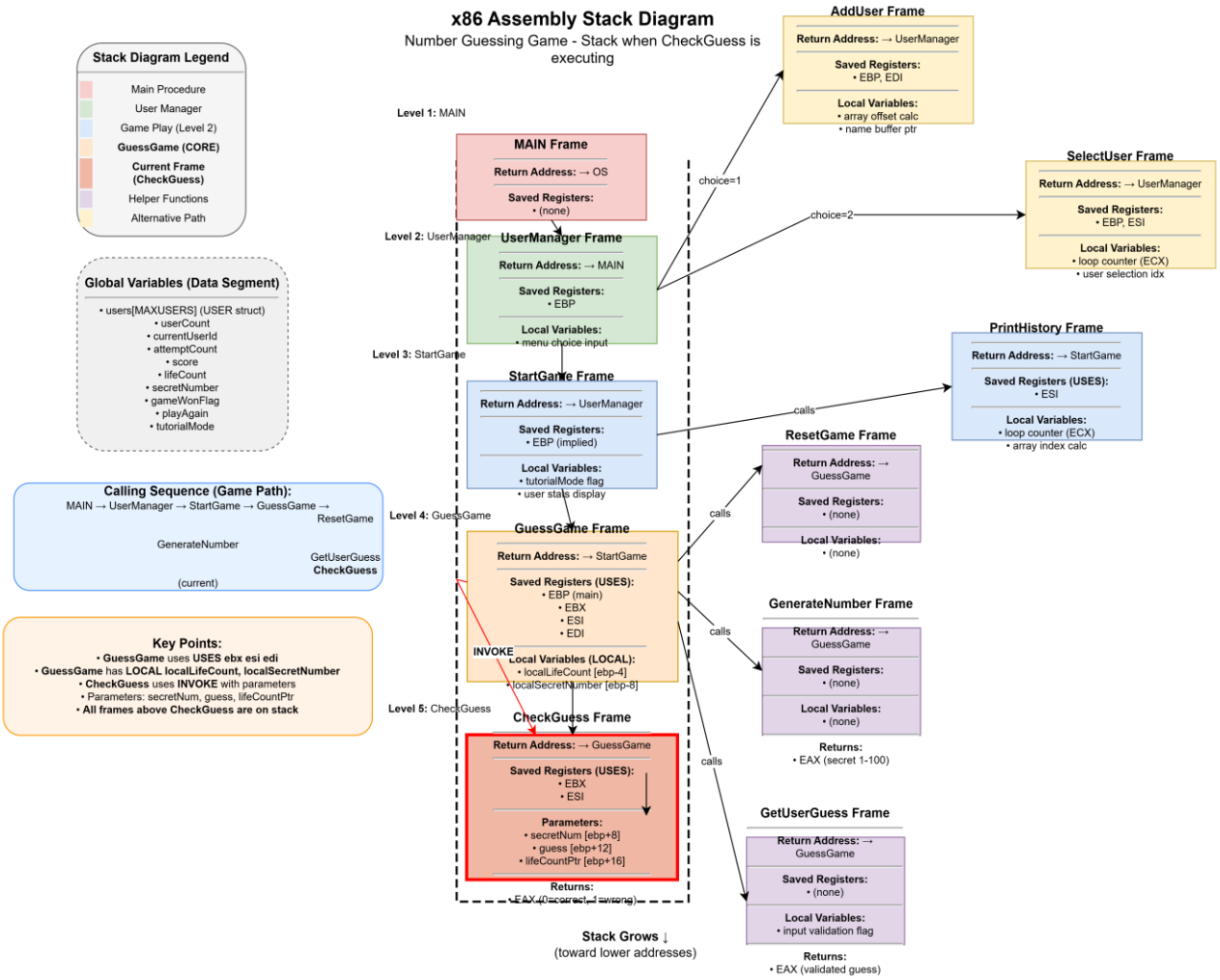
Program Flow



Key Assembly Concepts Demonstrated:

- **Registers:** EAX, EBX, ECX, EDX, ESI, EDI for computation and memory addressing
- **Memory Addressing:** Accessing structures and arrays using offsets
- **Loops & Conditional Jumps:** LOOP, JMP, JE, JNE, JL, JG, JGE, JLE
- **Procedures:** PROC/RET, INVOKE, parameter passing
- **Arithmetic Instructions:** MOV, ADD, SUB, MUL, DIV
- **Bitwise Operations:** AND for hint system (even/odd)
- **Input/Output:** ReadInt, WriteString, WriteDec, WriteChar using Irvine32
- **Error Handling:** Input validation and feedback

Stack Diagrams



Conceptual stack overview:

Step 1: main calls StartGame / GuessGame

- **Action:** main calls UserManager → eventually StartGame → GuessGame.
- **Stack after CALL GuessGame**

Stack (top --> bottom)	**Description**
return address	Address to resume after GuessGame call in StartGame
saved EBP	Base pointer for GuessGame
local variables	localLifeCount, localSecretNumber

USES ebx esi edi saves these registers automatically on stack

Step 2: GuessGame calls ResetGame

- **Action:** Reset game variables for new round.
- **Stack during call**

Stack (top to Bottom)	Description
return address	Address to resume after ResetGame
saved registers?	N/A for ResetGame (simple PROC, just ret)

After ret from ResetGame, stack **pops return address**, execution resumes in GuessGame.

Step 3: GuessGame calls GenerateNumber

- **Action:** Generate a random secret number (returned in EAX).
- **Stack:**

Stack (top to bottom)	Description
return address	Address to resume in GuessGame after GenerateNumber
saved registers?	N/A (no USES)

- GenerateNumber returns with EAX = secret number.
- Stack **pops return address**, resumes in GuessGame.
- localSecretNumber ← EAX

Step 4: GuessGame calls GetUserGuess inside game loop

- **Action:** Prompt user input.
- **Stack:**

Stack (top to bottom)	Description
return address	Resume in GuessGame after GetUserGuess

saved registers?	N/A

- User enters number → returned in EAX → stored in EDI as guess.
- Stack pops return address → resumes in GuessGame.

Step 5: GuessGame calls CheckGuess

- **Action:** Compare user guess vs secret number. Parameters passed:
    - secretNum → EBX
    - guess → EDI
    - lifeCountPtr → address of localLifeCount
- **Stack:**

Stack (top to bottom)	Description
return address	Resume in GuessGame after CheckGuess
saved registers	EBX / ESI (USES)
parameters	secretNum, guess, lifeCountPtr (passed by INVOKE) 1. lifeCountPtr (rightmost parameter) - pushed first 2. guess - pushed second 3. secretNum (leftmost parameter) - pushed third

- After return: EAX = 0 if correct, 1 if wrong.
- Stack pops back → execution continues in GuessGame.

Step 6: Game loop repeats / ends

- Life count decremented if wrong, loop continues.
- Once game ends:
    - GuessGame returns → stack **pops saved registers and return address** → resumes in StartGame.
    - StartGame then may call PrintHistory etc., each pushing return address for the call.

Step 7: Return to StartGame → UserManager → main

- All return addresses popped in order.
- Stack restored to **main state**, no local GuessGame variables remain.

Conclusion

The Enhanced Number Guessing Game demonstrates the practical application of **x86 Assembly programming concepts** in an interactive and educational context. By integrating **random number generation, user input validation, score calculation, tutorial guidance, and hint systems**, the project provides a complete gaming experience while showcasing low-level programming skills.

Through this project, we have successfully explored:

- **Procedural programming** in Assembly using PROC, RET, and INVOKE
- **Memory management and arrays** for user and session tracking
- **Conditional logic and loops** to implement game rules, input validation, and feedback
- **Bitwise operations** for hint generation
- **I/O handling** using the Irvine32 library for keyboard and screen interactions

The project also emphasizes **user engagement and learning**, allowing multiple users, session history tracking, and guided tutorial features for beginners. Overall, this Number Guessing Game serves as both a **practical implementation of Assembly concepts** and a **fun, educational tool** for understanding low-level programming logic, structure, and problem-solving techniques.