VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE & ENGINEERING

COMPUTER ARCHITECTURE

Assignment Report

Instructor: Nguyen Thien An

Students: Pham Anh Tai – 2350023
Tran Hoang Phuong – 2352959
Luc Duy Minh Quan – 2352994

HO CHI MINH, 6th May 2025

Contents

1 Introduction

Gomoku

Gomoku, also known as Five in a Row or Caro, is a classic abstract strategy board game originating from Japan's Edo period. Players alternate turns placing black and white stones on a grid, aiming to create an unbroken line of exactly five stones horizontally, vertically, or diagonally.

This report documents the implementation of the Gomoku game using MIPS32 Assembly language. The project focuses on designing a playable two-player Gomoku system at the assembly level, handling game board initialization, player input, move validation, win condition checking, and board display. By working at a low level, this implementation emphasizes memory management, control flow, and efficient data processing without the abstractions available in high-level languages.

The goal of this project is to demonstrate the ability to build a functional and interactive game solely through MIPS32 Assembly, highlighting both the possibilities and challenges of low-level programming.

2 Algorithm

The program simulates the two-player board game **Gomoku (Five in a Row)** on a 15×15 grid, allowing players to alternately input their moves and checking for a winning or tie condition after each move. The high-level algorithm is as follows:

1. **Initialize the Board:**
   - Set up a 15×15 board (`board[225]`) with all cells initially empty (represented by the `'.'` character).
2. **Start the Game Loop:**
   - Repeatedly display the board after each move.
   - Prompt the current player to input their move (row, column).
3. **Input and Validate Move:**
   - Parse the player's input string (row and column separated by a comma).
   - Validate if the input coordinates are within the bounds ($0 \rightarrow 14$) and the selected cell is empty.

- If the input is invalid, prompt the player to input again:
  `"Invalid input. Please enter coordinates again.\n"`

4. **Update the Board:**
   - Mark the board with `X` for Player 1 and `O` for Player 2 at the specified coordinates.
5. **Check for a Win:**
   - After each move, check four possible directions from the last move:
     - Horizontal (row)
     - Vertical (column)
     - Diagonal from top-left to bottom-right
     - Diagonal from top-right to bottom-left
   - A player wins if exactly five consecutive stones of their color are aligned in any direction.
6. **Handle Game End:**
   - If a player wins, display the winner and prompt whether to play again.
   - If all 225 moves are made and no player wins, declare a tie.
7. **Save Result:**
   - Save the final board state and the result (win/tie) to a file `result.txt` before exiting.
8. **Restart or Exit:**
   - If the user chooses to play again, reinitialize the board and restart.
   - Otherwise, exit the program.
9. **Flowchart:**

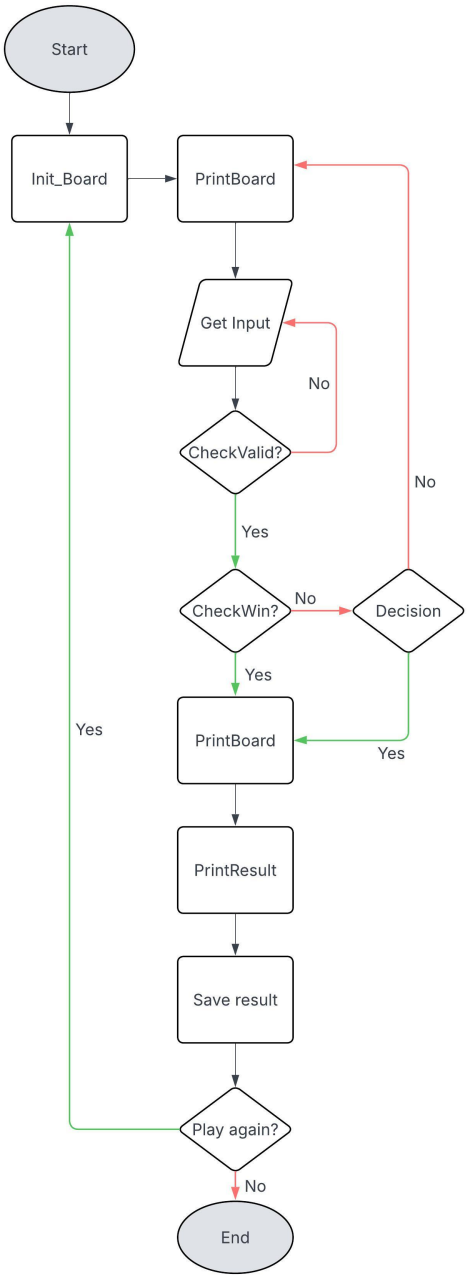


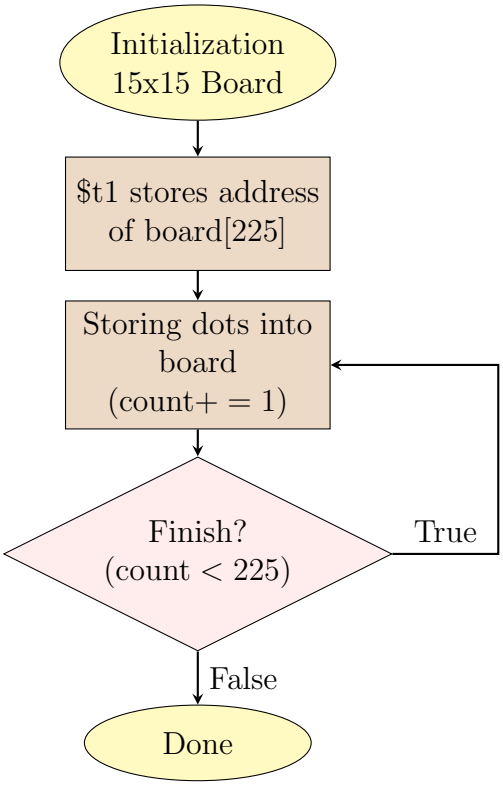
Figure 1: Flowchart

3 Implementation

The program is structured into distinct procedures, organized as follows:

1. **Board Initialization (`init_board`):**

   - This function prepares the game board for a new session. It ensures that every cell on the 15×15 grid is marked as empty, giving players a clean and consistent playing field. This is an essential setup step that must occur before any gameplay begins.
   - The function operates by looping exactly 225 times—once for each cell on the game board. During each iteration, it assigns a specific character (a dot `'.'`) to the current cell. This character represents an empty position, indicating that no player has placed a mark there yet.
   - Internally, the board is stored as a one-dimensional array of 225 bytes in memory. The function accesses and updates each element sequentially. This method ensures that the entire board is cleared of any data from previous games, avoiding interference with the current session.
   - Although the dot character is used within memory, the actual visual representation may vary when the board is displayed to players, depending on the rendering function.
   - By the end of the loop, all 225 positions are filled with the empty symbol. At this point, the board is fully initialized and ready for players to begin the game.



(Flowchart of 15×15 board initialization)

2. **Game Loop Control (`game_loop`):**
   - Manage the turn alternation ($s0 register: 0 for Player 1, 1 for Player 2) and move counting by using $s1 register (does not exceed 225).
3. **Input Handling (`input_coords`):**
   - Read the player's input in the format `(x,y)` into the `buffer`.
   - Parse the `x` and `y` coordinates separately, ensuring each is a valid digit and within the allowed range `[0,14]`.
   - Handle invalid inputs gracefully with meaningful error messages to guide the user.

   Purpose: This section of the program is responsible for reading input from the player, expecting it in the format `"x,y"`, and converting the string into two separate integer values: one for the row (x) and one for the column (y), to process the move.

Explanation: After displaying a prompt to the player, the program uses syscall 8 to read up to 10 characters of input from the keyboard into a buffer. The expected syntax is in the form `"number,number"`, for example: `"6,9"`.

The program then parses each character in the buffer as follows:

(a) **Parsing the part before the comma (row coordinate – x):** Starting from the beginning of the buffer, each character is read and checked:

- If the character is a comma `','`, the program switches to parsing the y-coordinate.
- If the character is the null terminator `'\0'`, an invalid format error is raised.
- If the character is not a digit (not between `'0'` and `'9'`), it is considered invalid.
- If valid, the character is converted from ASCII to its numeric value by subtracting 48.
- The value is accumulated using the formula: `value = value * 10 + digit`, allowing for multi-digit inputs like `"14"`.

(b) **Parsing the part after the comma (column coordinate – y):** Starting right after the comma:

- The loop performs similarly to the x-coordinate parsing.
- Parsing stops when a null terminator or a newline character (ASCII 10) is encountered, which corresponds to the player pressing Enter.
- The same format validation rules apply as with the x-coordinate.

Input Validation and Error Handling:

To ensure correct and robust interaction, the program validates the input and handles the following cases:

- **Invalid Input Format:** If the user enters a malformed coordinate (e.g., missing comma or contains non-numeric characters), the program displays:

  ```
  Error: Invalid input. Please enter coordinates again (x,y).
  ```

  How the program checks for errors: Each character is parsed sequentially. The input is valid only if:

- It consists of digits (ASCII 48–57),
- Contains exactly one comma (ASCII 44),
- Ends with a newline character (ASCII 10).

If any of these are missing or invalid characters are found, an error is triggered.

- **Coordinates Out of Range:** If either coordinate is not within the range [0,14], the program displays:

  ```
  Error: Coordinates outside the range. Must be between
  0 and 14.
  ```

  How the program checks for errors: After parsing, the program compares each coordinate to the valid range. If either is out of bounds, the user is prompted to re-enter.

- **Position Already Occupied:** If the specified cell on the board is not empty, the user is warned:

  ```
  Error: That position is already occupied. Try again.
  ```

  How the program checks for errors: The board index is calculated as `index = x * 15 + y`, and the memory at this position is checked. If it does not contain the empty character `'.'`, the move is rejected.

These safeguards ensure stable gameplay and improve the overall user experience.

4. **Board Updating:**
   - Calculate the 1D index from 2D coordinates: `index = row * 15 + column`.
   - Update the board array at the calculated index with the corresponding player's mark.
5. **Display Board (`print_board`):**
   - Print the column headers (0→14).
   - For each row, print the row index and all cells with appropriate formatting and spacing.

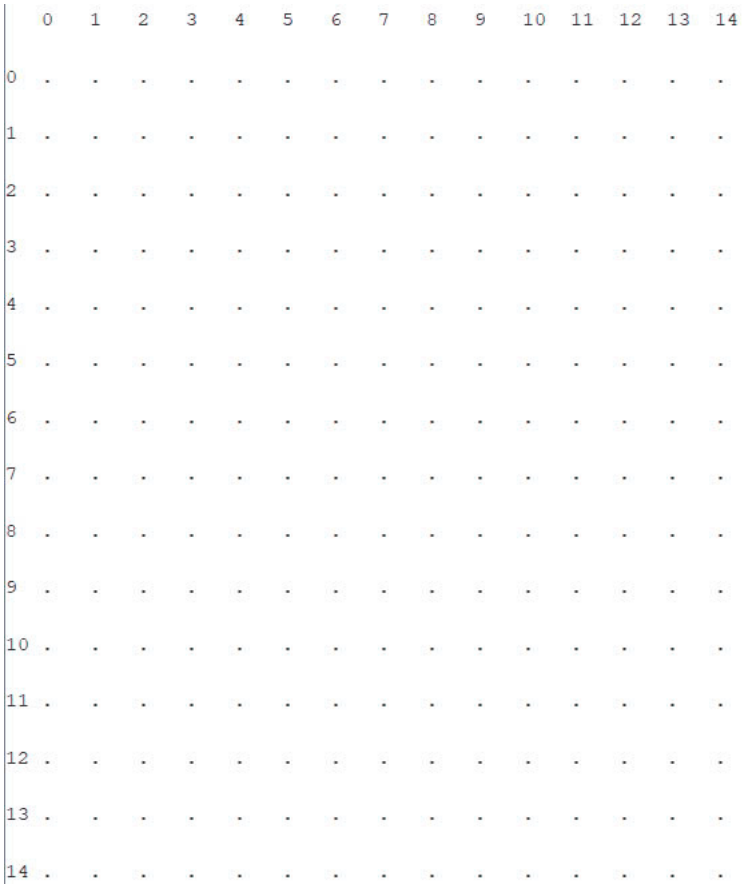
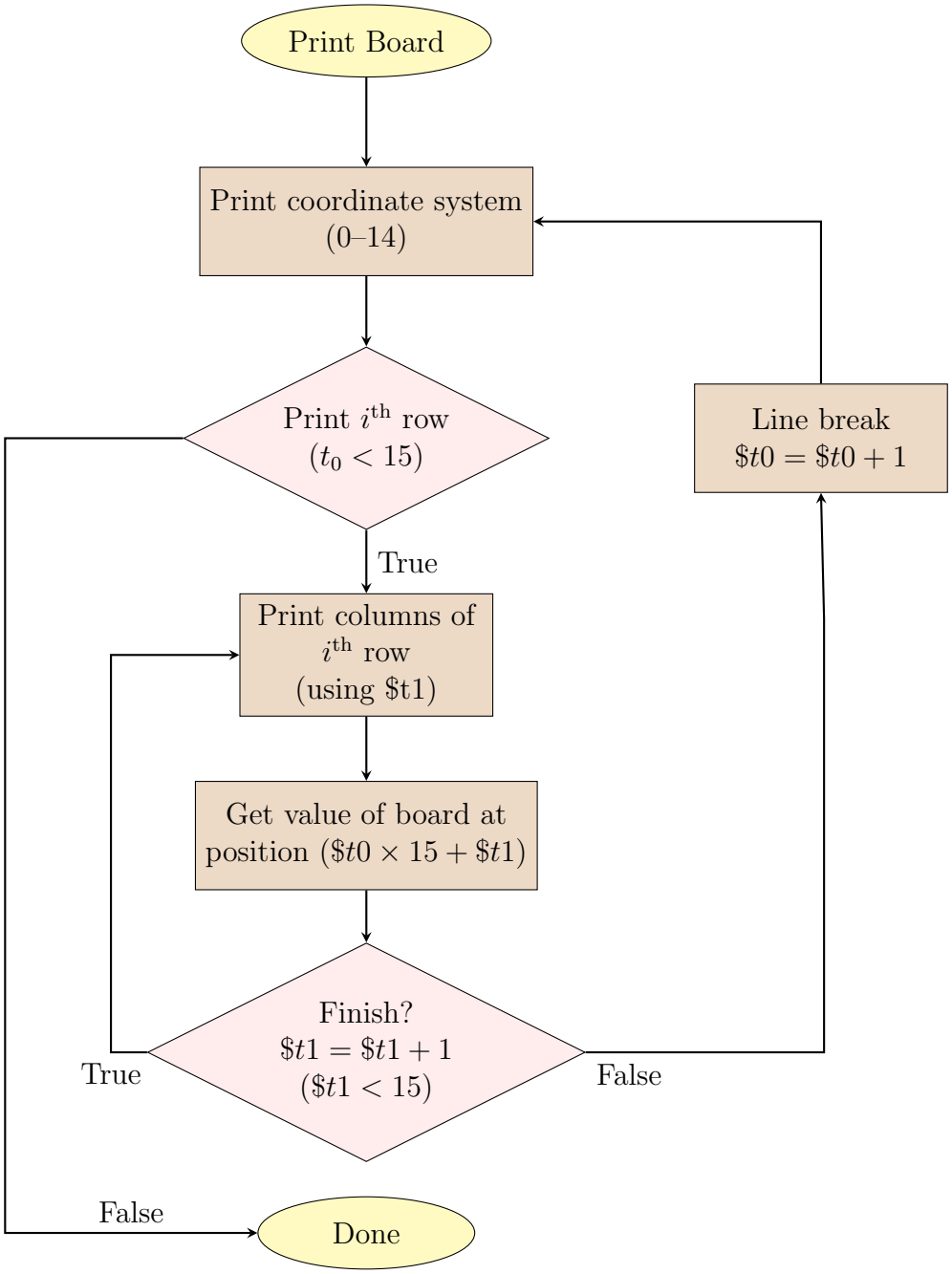


Figure 2: Board



(Flowchart of Board printing)

6. **Win Checking** (`check_win` + `check_line`):

Overview

- **check_win**: This is the main function that checks the win condition at a specific board position, given a row r, column c, and the player. The player is passed in register `$a2`; if `$a2 = 0`, it is Player 1, otherwise it's Player 2.
- **check_line**: A helper function called within `check_win` that checks in a specific direction (defined by delta row and delta column) whether there are five consecutive identical symbols (either all 'X' or all 'O') starting from the given position.
- **Tie Check**: In the main game loop, after each move, the program increments a counter that tracks how many cells have been filled. If no player has won and all 225 cells are filled, the game ends in a tie.

check_win Function Breakdown

The `check_win` function begins by saving its input arguments and the return address on the stack. It then determines which character the current player is using: 'X' for Player 1 or 'O' for Player 2.

After that, it checks four directions from the given position:

- Horizontally: $(0, 1)$
- Vertically: $(1, 0)$
- Diagonally (down-right): $(1, 1)$
- Diagonally (down-left): $(1, -1)$

For each direction, it calls the `check_line` function to check for five consecutive matching symbols. If any direction returns a win, the function returns 1. Otherwise, it returns 0.

check_line Function Breakdown

The `check_line` function checks in two directions from the current position:

- **Forward** in the direction given by the row and column deltas.
- **Backward** in the opposite direction.

Each time it finds a matching symbol, it increments a counter that keeps track of the number of consecutive matches. If the position goes out of bounds or the characters do not match, it exits the loop.

Forward Search Loop

In the forward loop, the function checks cells in the direction defined by the deltas $(\Delta r, \Delta c)$. For each step $k = 1, 2, 3, \ldots$, it computes the next row and column as:

$$r_{\text{new}} = r + k \cdot \Delta r, \quad c_{\text{new}} = c + k \cdot \Delta c$$

- If the new cell $(r_{\text{new}}, c_{\text{new}})$ is within bounds and contains the same symbol as the current player, the match counter is incremented.
- The loop continues until a mismatch is found or the edge of the board is reached.

Backward Search Loop

The backward loop is similar, but it searches in the opposite direction. For each step $k = 1, 2, 3, \ldots$, the coordinates are calculated as:

$$r_{\text{new}} = r - k \cdot \Delta r, \quad c_{\text{new}} = c - k \cdot \Delta c$$

- If the new position is valid and the symbol matches the current player's, the counter is incremented.
- The loop stops if an invalid cell is reached or the symbols do not match.

The goal is to count the total number of consecutive matching symbols from both directions, including the starting cell.

Final Check

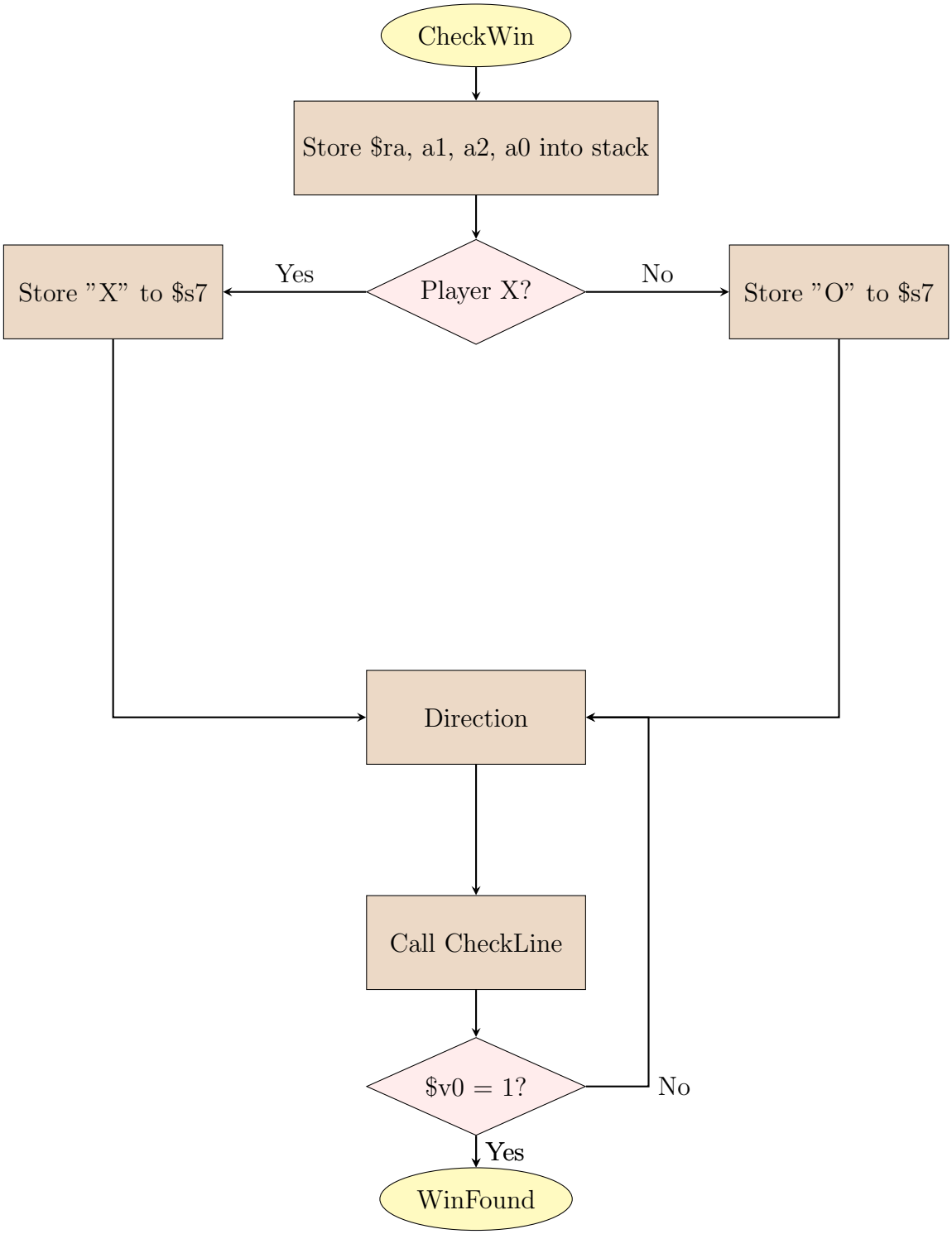
After scanning both directions, if the total number of consecutive matching characters (including the original cell) is greater than or equal to 5, the function returns 1 to indicate a win. Otherwise, it returns 0.

Tie Check Logic

Each time a player places a symbol, the program increments a move counter. If `$s1` (the move counter) reaches 225 and no player has won, the game is declared a tie. This ensures the game concludes in a draw if the board is completely filled with no winner.

Summary

- `check_win(r, c, player)` determines whether placing a symbol at position (r, c) results in a win for the given player.
- It does so by calling `check_line` in all four directions from the current cell.
- The `check_line` function checks forward and backward from that cell, counting the number of identical symbols. If five or more are found consecutively, it returns a win.
- If no win is detected and the total number of moves reaches 225, the game ends in a tie.


CheckWin
Store $ra, a1, a2, a0 into stack
Player X?
Yes
No
Store "X" to $s7
Store "O" to $s7
Direction
Call CheckLine
$v0 = 1?
No
Yes
WinFound

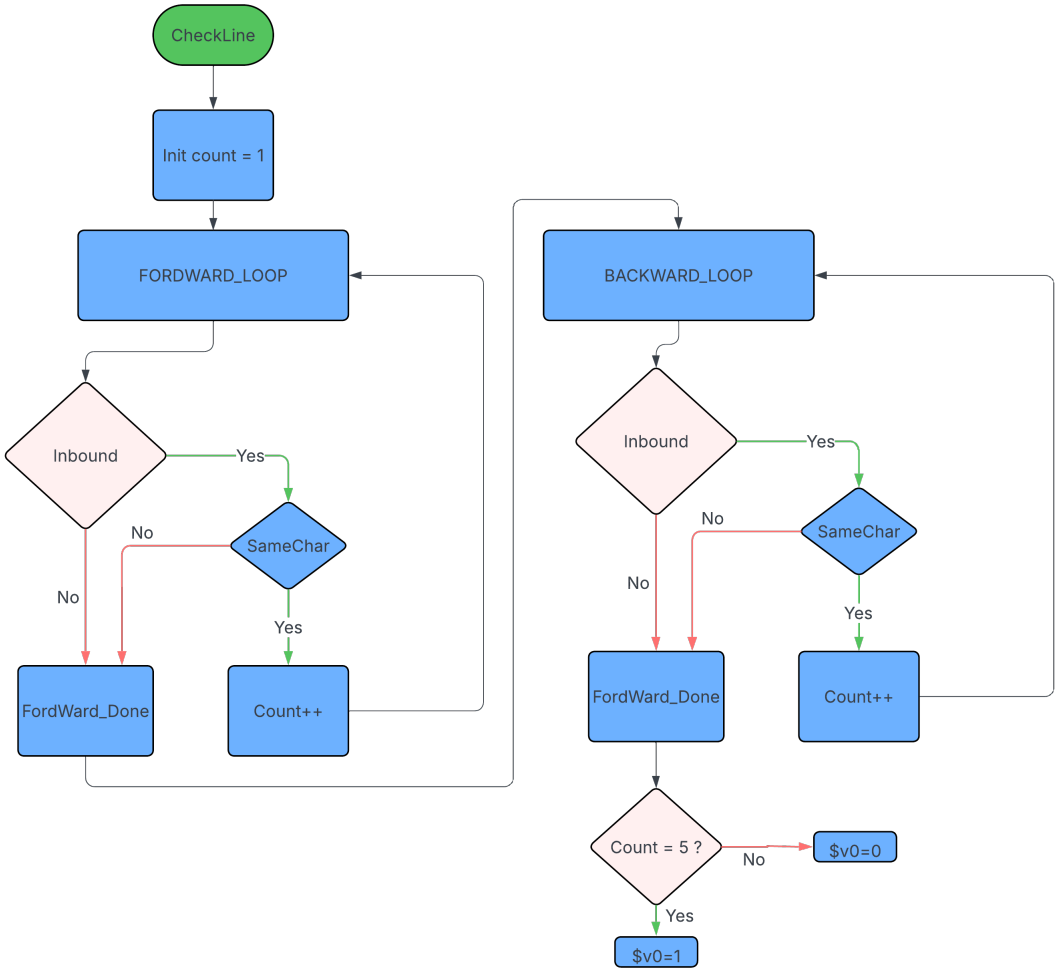


Figure 3: CheckLine

7. **Saving Result to File (`save_result`):**

   - The `save_result` procedure is used to write the result of the game to a text file named `result.txt` after the game ends.
   - The function works through the following steps:

   - **Saving Result to File (`save_result`):**
     - The `save_result` procedure is used to write the result of the game to a text file named `result.txt` after the game ends.
     - The function works through the following steps:
     - **Open the file:**
       The procedure opens the file `result.txt` in write mode. If the file does not exist, the operating system creates a new one. It opens the file in overwrite mode.
     - **Determine the game result:**
       Based on the value of the variable `turn`, the program determines the result:
       * If `turn = 0`: Player 1 wins (`"Player 1 wins"`).
       * If `turn = 1`: Player 2 wins (`"Player 2 wins"`).
       * If `turn = 3`: the game is a tie (`"Tie"`).
     - **Write the result to the file:**
       The corresponding result message is written to `result.txt`. This operation overwrites any existing content in the file, keeping only the latest result.
     - **Saving Result to File:**
       Write the board state row by row into the file. Append the result (Player 1 wins / Player 2 wins / Tie) at the end of the file.
     - **Close the file:**
       After writing, the file is closed to ensure data safety and to release system resources.
     - **Note:**

       * Each time the result is saved, the old content is **overwritten**, not appended.
       * The file `result.txt` is saved in the current working directory of the program.

8. **Restart Mechanism:**

   After the game ends—either due to a win or a tie—the program enters a restart phase, giving players the option to play another round.

   - Once a win is detected (by `check_win`) or the tie condition is met (when 225 cells are filled without a win), the program jumps to either the `winner` or `tie_game` label.
   - At this point, the game displays a message to indicate the result and prompts the user to choose whether to restart or quit. This is typically handled using system calls to read user input from the console.
   - The user's choice is evaluated:
     - If the player chooses to restart (e.g., enters 'Y'), the game resets key state variables:
       * The board array is cleared.
       * The move counter is set back to 0.
       * The starting player is reset (usually Player 1).

       The program then jumps back to the `game_loop` to start a new game.
     - If the player chooses to quit (e.g., enters 'N'), the program exits using a syscall.
   - The restart mechanism ensures that players can continuously enjoy the game without needing to re-run the program manually after each match.

4 Result

In this section, we present the results of several test cases for the Caro (Gomoku) game, including the outcomes of different end-game scenarios.

Test Cases

Test Case	Board State	Outcome	Result
Test Case 1	[Player 1 Wins]	Player 1 (X) Wins	Pass
Test Case 2	[Player 2 Wins]	Player 2 (O) Wins	Pass
Test Case 3	[Tie Game]	Tie (Board Full)	Pass

Table 1: Test Results for Various Game Scenarios

Individual Test Case Descriptions

Test Case 1: Player 1 Wins

- Board state:

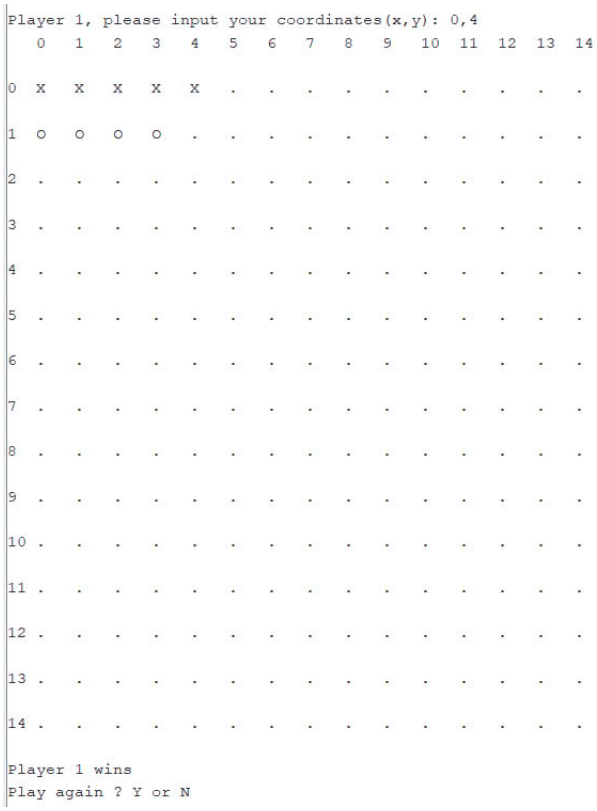


Figure 4: Sample board showing Player 1's winning move

- Expected Result: Player 1 (X) wins.
- Outcome: Pass

Test Case 2: Player 2 Wins

- Board state:

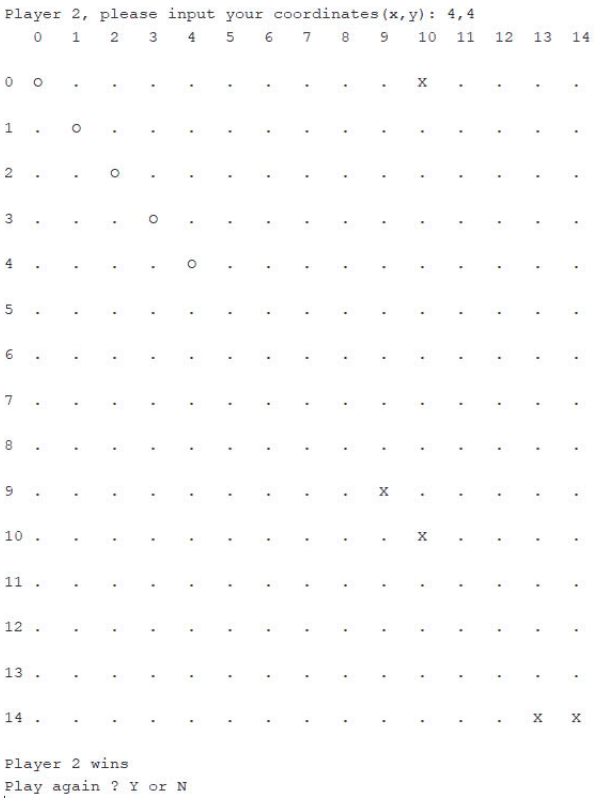


Figure 5: Sample board showing Player 2's winning move

- Expected Result: Player 2 (O) wins.
- Outcome: Pass

Test Case 3: Tie Game

- Board state:

```
Player 2, please input your coordinates: 14,12
   0  1  2  3  4  5  6  7  8  9  10 11 12 13 14

0  X  O  O  X  X  O  X  O  X  O  X  O  X  O  X

1  O  X  O  X  O  O  X  X  O  X  O  X  O  X  O

2  O  X  O  X  X  O  O  X  O  X  X  X  O  O  X

3  O  O  X  O  X  X  X  X  O  O  O  O  X  O  X

4  X  O  X  O  X  O  O  O  X  X  X  O  X  O  O

5  O  X  O  X  O  O  X  X  X  O  X  O  O  .  O

6  X  O  X  O  X  X  O  X  O  O  X  X  X  O  X

7  O  O  X  O  X  O  X  O  O  X  O  X  O  X  X

8  X  O  X  O  X  O  X  O  O  X  O  X  O  X  X

9  O  X  O  X  O  X  O  X  O  X  O  X  O  X  X

10 X  X  O  X  O  X  O  X  X  O  X  O  X  O  O

11 O  O  X  O  O  X  X  O  X  O  X  O  X  O  X

12 X  O  X  O  X  O  X  O  X  O  X  O  X  O  O

13 O  X  X  X  X  O  O  X  O  X  O  X  O  X  X

14 X  X  O  O  X  O  X  O  X  O  X  O  O  O  O

Player 1, please input your coordinates: 5,13
Tie
Play again ? Y or N
N
-- program is finished running --
```

Figure 6: Full board with no winner

- Expected Result: Tie.
- Outcome: Pass

5 Save Result

Test Case 1: Player 1 Wins

Figure 7: Sample board showing Player 1's winning move

Test Case 2: Player 2 Wins

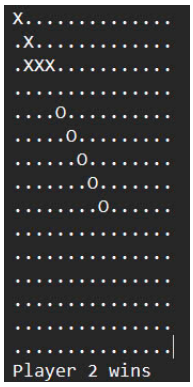


Figure 8: Sample board showing Player 2's winning move

Test Case 3: Tie

Figure 9: Sample board showing tie

6 Conclusion

This project provided several valuable lessons. Firstly, it offered practical experience with MIPS assembly language, enhancing my understanding of low-level programming principles such as memory management, input parsing, control flow, and system calls. By implementing player input handling, validation, and a basic game board through direct memory operations, I gained a better understanding of how computers process and represent data internally.

Moreover, the project reinforced theoretical knowledge from the course, allowing me to apply core concepts in a real-world setting. Creating a simple game in assembly language required careful attention to detail, logical thinking, and disciplined programming techniques—skills essential for more advanced topics in computer architecture and system-level programming.

Lastly, the project showcased fundamental game development principles, proving that functional games can be created without relying on high-level libraries or engines, using only data structures, input/output operations, and control logic. This experience laid a strong foundation for tackling more complex programming tasks in the future.

In summary, the project not only sharpened my technical abilities but also deepened my appreciation for the sophistication and beauty of low-level programming.