VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
Faculty of Computer Science and Engineering

Course:
COMPUTER ARCHITECTURE - CO2008

Report: IMPLEMENTATION OF FIVE IN A ROW USING ASSEMBLY CODE

Supervisor: Prof. Nguyen Thien An

Students: Tran Gia Linh – 2252434
Nguyen Viet Hung - 2252273
Nguyen Thi Quoc Nguyen – 2252549

HO CHI MINH CITY, APRIL 2025

Assessment

No.	Fullname	Student ID	Tasks	Completion
1	Nguyen Viet Hung	2252273	Implementation, Report	100%
2	Nguyen Thi Quoc Nguyen	2252549	Implementation, Report	100%
3	Tran Gia Linh	2252434	Implementation, Report	100%

Table 1: Table of Workload Distribution for the project

Contents

1 Introduction

In this report, we discussed the implementation of the program:

- Section 1 declares the objectives of the project and introduce **Five in a row** and its rules.
- Section 2 discusses the flow of the program and provide pseudo-code of key procedures of the program.
- Section 3 shows an example game-play from the start to the end, including error handling; and also provide link to source code.

1.1 Objective

The goal of this project is to create a Five in a row game, an advanced version of Tic-Tac-Toe, on a 15x15 grid using MIPS assembly language.

The program allows two players to alternately place their markers 'X' and 'O' on the board, aiming to form a continuous line of five markers horizontally, vertically, or diagonally to secure victory. Key functionalities include a clear board display, robust input handling, and logic to detect winning conditions or a draw.

Developed in MIPS, this project showcases skills in low-level programming, efficient algorithm design, and game mechanics implementation in a resource-constrained environment.

1.2 Five In A Row

Five in a Row is a two-player strategy game played on a square grid, commonly 15x15, though other sizes are also acceptable. The goal is simple: be the first player to place five or more of your marks in a continuous line—horizontally, vertically, or diagonally.

Players take turns placing their marks on empty intersections of the board. Player 1 always goes first, followed by Player 2. Once placed, marks cannot be moved or removed. The game ends immediately when a player completes a line of five marks. In this project, standard rule is applied, so six or more marks in a row is still counted as a win. If the board fills with no winner, the game is declared a draw.

Five in a Row is known for being easy to learn but hard to master, requiring players to balance attack and defense while thinking several steps ahead. This project implements the game using standard rules, ensuring proper turn-taking, move validation, handling input errors, and win detection, while offering a smooth and fair gameplay experience.

2 Implementation details

2.1 Program flow

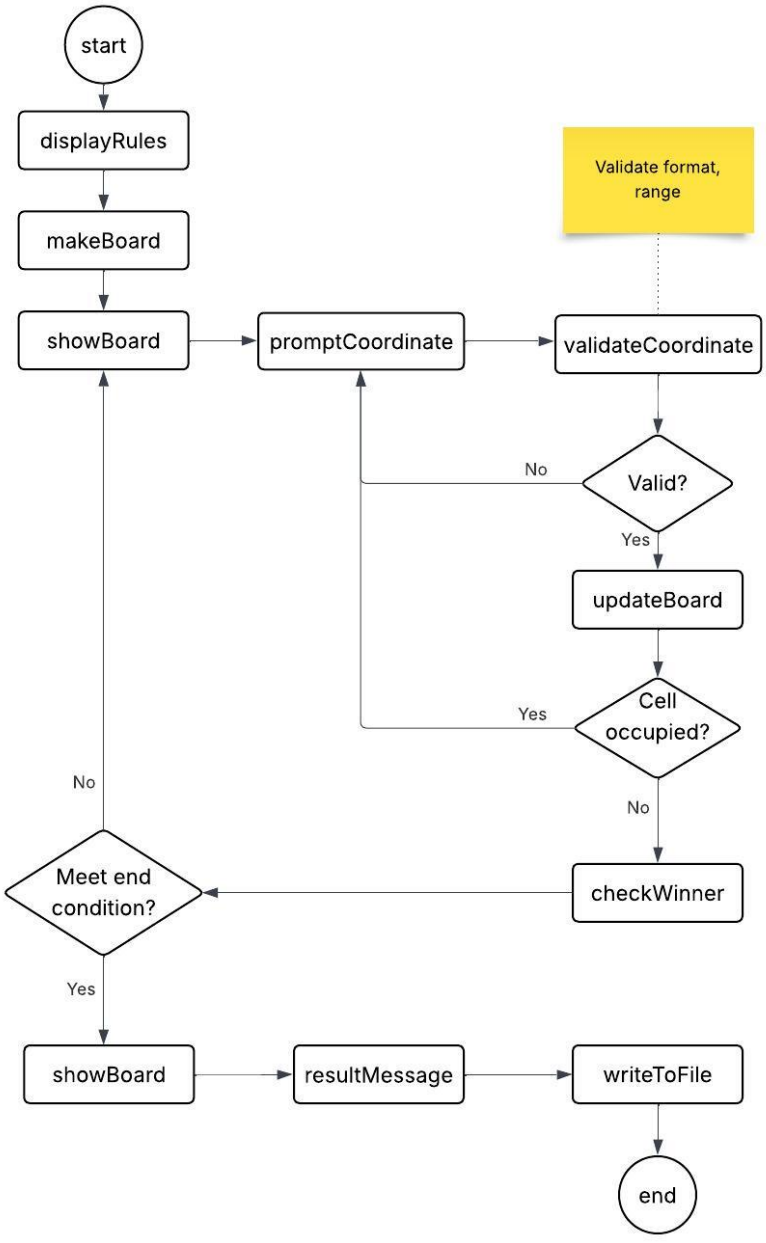

Figure 2.1: Activity Diagram of Program

Figure 2.1 describes the flow of the program. In details, the program will go through these steps:

Step 1. Display rules of the games, including rules for inputting, stop conditions.

Step 2. Initialize array `board` - which is used to stored information about the cell (whether it is not occupied - value 0, or occupied by either of the players - value 1 or 2); and also initialize the `displayBoard` - which is the formatted board used for displaying. Two different boards were used to check for end conditions and to display game visualization more efficiently.

Step 3. Show `displayBoard` by simply printing it.

Step 4. Read input coordinates from players, then check if the format and range is appropriate. If invalid, perform step 4 from the start. If valid, move to step 5.

Step 5. Access memory address of input coordinates (`board`), then check whether that position is occupied by either player. If it is occupied, go to step 4. If it is not occupied by any player, change the values of the memory location corresponding to input coordinates in both `board` and `displayBoard` to appropriate values.

Step 6. Check for stop conditions: if any player occupied five in a row (vertically, horizontally, diagonally) or board is full. If stop conditions are not met, go to step 3. Otherwise, go to step 7.

Step 7. Show `displayBoard` by simply printing it.

Step 8. Write latest `displayBoard` and result to `result.txt`. Program stops.

In the next section, pseudo-code of key procedures will be provided in depth.

2.2 Procedure details

2.2.1 makeBoard

Algorithm 1: `makeBoard` Procedure

Purpose: Initialize the `board` (15×15 grid) and the display string `displayBoard`
Modifies: `board`, `displayBoard`

1: Save return address and call `stackIn`
{— Initialize `board` entries to 0 —}
2: **for** `i = 0` to 224 **do**
3: `board[i]` ← `0`
4: **end for**
{— Build header of `displayBoard` —}
5: Write 5 spaces
6: **for** `col = 0` to 14 **do**
7: Write _D_ for column number with padding (3 spaces)
8: **end for**
9: Write newline
{— Build 15 board rows —}
10: **for** `row = 0` to 14 **do**
11: Write row index with correct spacing (1-digit: 4 spaces, 2-digit: 3 spaces)
12: **for** `col = 0` to 14 **do**
13: Write ___ and 3 spaces
14: **end for**
15: Write newline
16: **end for**
17: Append null terminator to `displayBoard`
18: Call `stackOut`, restore return address, and return

The `makeBoard` procedure is responsible for preparing the data structures needed to represent a 15×15 game board.

- It first initializes the `board` array by setting all 225 entries to `0`.
- Then, it builds a human-readable version of the board in the `displayBoard` character array:
  - A header row includes column numbers 0 to 14 in a padded `'_D_ '` (3 spaces) format.
  - Each board row is prefixed with its row number, properly padded (5 characters per prefix).
  - Each cell is shown as ___ followed by 3 spaces for alignment.
  - Rows are terminated with newline characters, and the board ends with a null terminator.

Upon return, `board` and `displayBoard` are fully initialized and ready for use in the game.

2.2.2 promptCoord

Algorithm 2: `promptCoord` Procedure

Require: `$s0`: Player flag (0 = Player 1, 1 = Player 2)
Return: `$s2, $s3`: Row and column of valid move

```
1: Save return address and call stackIn
2: loop
3:    if $s0 == 0 then
4:       Prompt Player 1 for coordinates
5:    else
6:       Prompt Player 2 for coordinates
7:    end if
8:    Call validateCoord
9:    if $s5 == 0 then
10:      continue
11:   end if
12:   Call updateBoard
13:   if $s5 == 0 then
14:      continue
15:   else
16:      break
17:   end if
18: end loop
19: Call stackOut and return
```

The `promptCoord` procedure handles user input for a turn-based game involving two players. It prompts the active player (based on the value in `$s0`) to enter their move coordinates, validates the input, and updates the game board if the move is legal.
The function operates within a loop, repeating the input-validation-update process until a valid move is successfully applied:

- If `$s0 == 0`, it prompts Player 1; otherwise, it prompts Player 2.
- It calls `validateCoord` to check if the input coordinates are valid.
- If validation fails (`$s5 == 0`), it restarts the loop from the prompt step.
- If validation succeeds, it proceeds to call `updateBoard`.
- If the board update also succeeds (`$s5 == 1`), the loop exits.
- Otherwise, the loop restarts from the prompt step.

Upon completing a successful move, the procedure prints a newline, restores the return address, and exits. On return, the output registers contain:

- `$s2`, `$s3`: the row and column of the valid move.

2.2.3 validateCoord

Algorithm 3: `validateCoord` Procedure

Return: `$s5`: 1 if input is valid, 0 otherwise; `$s2, $s3`: row and column values

1: Save return address and call `stackIn`
2: Read a string from input into buffer `coord` (max 127 chars)
3: Scan the buffer:

- Count characters
- Locate the comma (only one allowed)

4: **if** length < 3 or > 5 **or** comma not found **or** more than one comma **then**
5: Display invalid format message
6: Set `$s5` $= 0$ and **return**
7: **end if**
8: Extract and parse substring before comma via `checkEachCoord`, store in `$s2` (row)
9: **if** `checkEachCoord` fails **then**
10: Set `$s5` $= 0$ and **return**
11: **end if**
12: Extract and parse substring after comma via `checkEachCoord`, store in `$s3` (column)
13: Set `$s5` $= 1$
14: Call `stackOut` and return

The `validateCoord` procedure is responsible for reading a coordinate input string from the user and validating its format. It expects an input in the form of `row,col`, where both parts are numeric and separated by a single comma. Accepted formats include inputs like `1,2`, `12,3`, or `4,10`.
The procedure works as follows:

- It reads a line of input into a buffer.
- It counts the total number of characters and checks for exactly one comma.
- If the format is incorrect (e.g., too short, too long, or has no comma/multiple commas), it prints an error message and returns `$s5 = 0`.
- If the format is structurally valid, it splits the input into two parts:
  - The part before the comma is parsed and validated using `checkEachCoord`, and stored in `$s2`.
  - The part after the comma is similarly validated and stored in `$s3`.
- If both parts are valid numbers, the function sets `$s5 = 1`.

On return:

- `$s5` $= 1$ indicates the input was valid; 0 otherwise.
- `$s2` and `$s3` contain the parsed row and column values if the input was valid.

2.2.4 checkEachCoord

Algorithm 4: `checkEachCoord` – Coordinate Validation

Require: `$t0`: Start index, `$t2`: End index + 1
Return: `$s4`: Parsed coordinate (0–14), `$s5`: Valid flag (1 if valid, 0 otherwise)

```
1: Save return address and call stackIn
2: Compute length: $t2 ← $t2 - $t0
3: if $t2 == 0 or $t2 > 2 then
4:    Print "Out of Range", set $s5 ← 0, return
5: end if
6: $t0 ← coord + $t0
7: if $t2 == 1 then
8:    Read char, convert to digit
9:    if not a digit then
10:      Print "Not a Digit", $s5 ← 0
11:   else
12:      $s4 ← digit, $s5 ← 1
13:   end if
14: else
15:   Read first char, reject if '0'
16:   Check if it is '1'; else, out of range
17:   Read second char, convert to digit
18:   if digit in [0, 4] then
19:      $s4 ← digit + 10, $s5 ← 1
20:   else
21:      Print "Out of Range", $s5 ← 0
22:   end if
23: end if
24: Call stackOut, restore return address, return
```

The `checkEachCoord` procedure validates the coordinates entered by the user for a move on the game board. It checks that the input is within the allowed range and that it consists of valid digits.
The procedure works as follows:

- It first calculates the length of the input coordinates by subtracting the start index (`$t0`) from the end index (`$t2`).
- If the length of the input is invalid (i.e., it's either zero or greater than two characters), it prints an error message and sets `$s5` to 0 to indicate an invalid coordinate.
- If the input consists of a single character:
  - It checks if the character is a valid digit (between 0 and 9).
  - If valid, the corresponding coordinate value is stored in `$s4`, and `$s5` is set to 1.
  - If invalid, an error message is printed, and `$s5` is set to 0.

- If the input consists of two characters:
  - It checks if the value is a valid number between 10 and 14 (inclusive).
  - If valid, the coordinate value is stored in `$s4`, and `$s5` is set to 1.
  - If invalid (either out of range or non-numeric), an error message is printed, and `$s5` is set to 0.
- After the validation, if the coordinate is valid, `$s5` will be 1, indicating success. If the coordinate is invalid, `$s5` will be 0.

2.2.5 updateBoard

Algorithm 5: `updateBoard` Pseudo-Algorithm

Arguments: `$s0`: Player flag, `$s2`: Row index, `$s3`: Column index
Return: `$s5`: 1 if successful, 0 if failed

```
1: index = row * 15 + col
2: if board[index] != 0 then
3:    Print "Cell occupied", $s5 = 0
4: else
5:    if $s0 == 0 then
6:       mark = 1                                   (Player 1: 'X')
7:       display_char = 'X'
8:    else
9:       mark = 2                                   (Player 2: 'O')
10:      display_char = 'O'
11:   end if
12:   board[index] = mark
13:   displayBoard_index = (row + 1) * 96 + 5 + col * 6 + 1
14:   displayBoard[displayBoard_index] = display_char
15:   $s5 = 1
16: end if
```

`updateBoard` updates both the internal game board and the display board with the current player's move. The procedure performs the following tasks:

- Calculates the index in the `board` array based on the given row and column.
- If the selected cell is already occupied (i.e., `board[$index] != 0`), it prints a message indicating the cell is preoccupied and returns with `$s5 = 0`.
- If the cell is empty:
  - It assigns the appropriate mark for the current player:
    - Player 1 (`$s0 = 0`) is assigned 'X' (represented by 1).
    - Player 2 (`$s0 = 1`) is assigned 'O' (represented by 2).
  - The mark is stored in the `board` array.

  - The corresponding display character ('X' or 'O') is placed in the `displayBoard` at the correct position.

- Finally, the procedure returns with `$s5 = 1`, indicating the successful update of both boards.

2.2.6 checkWinner

Algorithm 6: `checkWinner` Pseudo-Algorithm (Part 1: Horizontal Checks)

Return: `$s5`: 1 if game ends, 0 otherwise; `$s4`: 0 if Player 1 wins, 1 if Player 2 wins, 2 if Tie

```
1:  // Check for horizontal wins
2:  for each row r in the board do
3:    for each column c where c ≤ 10 do
4:      index = row * 15 + col
5:      cell_value = board[index]
6:      if cell_value != 0 then
7:        consecutive = 1
8:        player_mark = cell_value
9:        while consecutive < 5 do
10:         next_cell_value = board[index + consecutive]
11:         if next_cell_value != player_mark then
12:           break
13:         end if
14:         consecutive++
15:         if consecutive == 5 then
16:           goto winner_found
17:         end if
18:       end while
19:     end if
20:   end for
21: end for
```

The `checkWinner` procedure determines the result of the current board state in a two-player game. It checks whether either player has achieved five consecutive marks in any valid direction or if the board is full (resulting in a tie).
The procedure works as follows:

- It initializes `$s5` to 0 (game not ended) and `$s4` to 2 (default: tie).
- It checks for a horizontal win:
  - It iterates through each row and, within each row, scans columns 0 to 10 to ensure enough space for a 5-in-a-row.
  - If five consecutive identical non-zero values are found horizontally, the procedure jumps to the winner-handling section.
- It checks for a vertical win:

Algorithm 7: `checkWinner` Pseudo-Algorithm (Part 2: Vertical Checks)

```
// Check for vertical wins
for each row r where r ≤ 10 do
  for each column c in the board do
    index = row * 15 + col
    cell_value = board[index]
    if cell_value != 0 then
      consecutive = 1
      player_mark = cell_value
      while consecutive < 5 do
        next_index = index + consecutive * 15
        next_cell_value = board[next_index]
        if next_cell_value != player_mark then
          break
        end if
        consecutive++
        if consecutive == 5 then
          goto winner_found
        end if
      end while
    end if
  end for
end for
```

  - It iterates through each column and, within each column, scans rows 0 to 10.
  - If five consecutive identical non-zero values are found vertically, the procedure jumps to the winner-handling section.

- It checks for diagonal wins in two directions:
  - For down-right diagonals (\), it checks from rows 0 to 10 and columns 0 to 10 by moving diagonally with an index offset of +16.
  - For down-left diagonals (/), it checks from rows 0 to 10 and columns 4 to 14 by moving diagonally with an index offset of +14.
  - In both cases, if five consecutive identical non-zero values are found, the procedure jumps to the winner-handling section.

- If no winning sequence is found, it checks for a tie:
  - It scans all 225 board cells to see if any are empty (value = 0).
  - If all cells are filled and no winner is detected, the game is declared a tie by setting `$s4` to 2 and `$s5` to 1.
  - If at least one cell is empty, the game continues and `$s5` remains 0.

- If a winner is found:
  - The procedure checks the player's mark (`1` or `2`) and sets `$s4` accordingly (0 for Player 1, 1 for Player 2).

Algorithm 8: `checkWinner` Pseudo-Algorithm (Part 3: Diagonal Checks)

```
1: // Check for diagonal wins (down-right: \)
2: for each row r where r ≤ 10 do
3:   for each column c where c ≤ 10 do
4:     index = r * 15 + c
5:     cell_value = board[index]
6:     if cell_value != 0 then
7:       consecutive = 1
8:       player_mark = cell_value
9:       while consecutive < 5 do
10:        next_index = index + consecutive * 16 {Move diagonally down-right}
11:        next_cell_value = board[next_index]
12:        if next_cell_value != player_mark then
13:          break
14:        end if
15:        consecutive++
16:        if consecutive == 5 then
17:          goto winner_found
18:        end if
19:      end while
20:    end if
21:  end for
22: end for
23: // Check for diagonal wins (down-left: /)
24: for each row r where r ≤ 10 do
25:   for each column c where c ≥ 4 do
26:     index = r * 15 + c
27:     cell_value = board[index]
28:     if cell_value != 0 then
29:       consecutive = 1
30:       player_mark = cell_value
31:       while consecutive < 5 do
32:         next_index = index + consecutive * 14 {Move diagonally down-left}
33:         next_cell_value = board[next_index]
34:         if next_cell_value != player_mark then
35:           break
36:         end if
37:         consecutive++
38:         if consecutive == 5 then
39:           goto winner_found
40:         end if
41:       end while
42:     end if
43:   end for
44: end for
```

Algorithm 9: `checkWinner` Pseudo-Algorithm (Part 4: Tie Check and Results)

```
for each cell in the board do
    if cell is empty (value = 0) then
        $s5 = 0 {Game continues}
        goto end_checkWinner
    end if
end for
$s4 = 2, $s5 = 1 {Tie, game ended}
goto end_checkWinner
winner_found:
if player_mark == 1 then
    $s4 = 0 {Player 1 wins}
else
    $s4 = 1 {Player 2 wins}
end if
$s5 = 1 {Game ended}
end_checkWinner:
Return to caller
```

  - It then sets `$s5` to 1 to indicate that the game has ended.

- After completing the checks, the procedure restores saved registers and returns to the caller.

2.2.7 writeToFile

Algorithm 10: `writeToFile` Procedure

Arguments: `$s4` – 0 if Player 1 wins, 1 if Player 2 wins, 2 if tie

```
Save return address and call stackIn
Move to the end of displayBoard (1536 bytes)
if $s4 == 0 then
    Set message to "Player 1 wins" (player1win)
else if $s4 == 1 then
    Set message to "Player 2 wins" (player2win)
else
    Set message to "Tie" (playertie)
end if
Append the result message to the end of displayBoard
Null-terminate the buffer
Open file result.txt for writing
Write the content of displayBoard including the appended result
Close the file
Call stackOut, restore return address, and return
```

The `writeToFile` procedure is used to write the current game board display along with the game result to a text file named `result.txt`. It assumes that the board display has

already been rendered in the `displayBoard` buffer, which is 1536 bytes in length and null-terminated. The procedure works as follows:

- It appends a short result message ("Player 1 wins", "Player 2 wins", or "Tie") to the end of the display buffer.
- The message to append is selected based on the value of `$s4`:
  - `$s4 = 0` ⇒ Player 1 wins
  - `$s4 = 1` ⇒ Player 2 wins
  - `$s4 = 2` ⇒ Tie
- Add null-terminate character.
- The file `result.txt` is then opened in write mode, and the content of `displayBoard` (including the appended result message) is written to it.
- Finally, the file is closed and the procedure returns.

3 Result

3.1 Gameplay

The Gomoku game project, developed on a 15x15 grid using MIPS assembly language, was successfully completed, delivering a robust and interactive gaming experience while meeting all specified objectives. The program begins by clearly displaying the game rules, providing players with concise instructions on how to place markers ('X' or 'O') and the goal of aligning five consecutive markers to win, either horizontally, vertically, or diagonally.

```
------------------------------------------------------------
=================== FIVE IN A ROW RULES ====================
------------------------------------------------------------
* GAME OBJECTIVE *
-Be the first player to place 5 of your marks in a row
(horizontally, vertically, or diagonally).
- The game is played on a 15x15 grid (225 cells).
------------------------------------------------------------
* PLAYERS AND SYMBOLS *
Two players take turns:
- Player 1 uses symbol: X
- Player 2 uses symbol: O
------------------------------------------------------------
* INPUT FORMAT *
Every coordinates input must be in format: [   X   ,   Y   ]
Where:
- X: horizontal coordinate, 0 <= X <= 14
- Y: vertical coordinate, 0 <= Y <= 14
Note:
- No leading zeros.
- No trailing spaces.
- If there is a format mistake, we will let you input again.
------------------------------------------------------------
* WINNING CONDITION & TIE CONDITION *
A player wins if they have 5 consecutive symbols in one of
these ways:
- Horizontally
- Vertically
- Diagonally (\ or / direction)
All 225 cells are filled with no winner - the game announces:
"Tie"
------------------------------------------------------------
```

Figure 3.1: Display of game rules on terminal

This rule display is formatted for readability, ensuring players understand the mechanics before starting (see Figure 3.1).

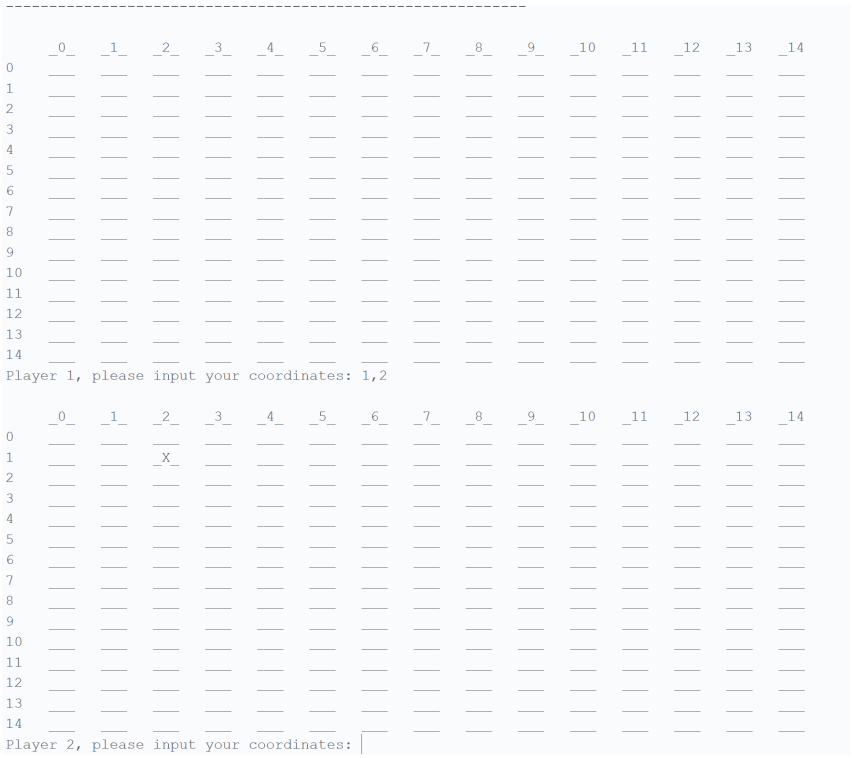

Figure 3.2: Prompting for coordinates input from players

Users input handling was implemented with precision, allowing players to specify grid coordinates for marker placement through a structured interface. The user will input in the format of "row number,column number" (see figure 3.2).

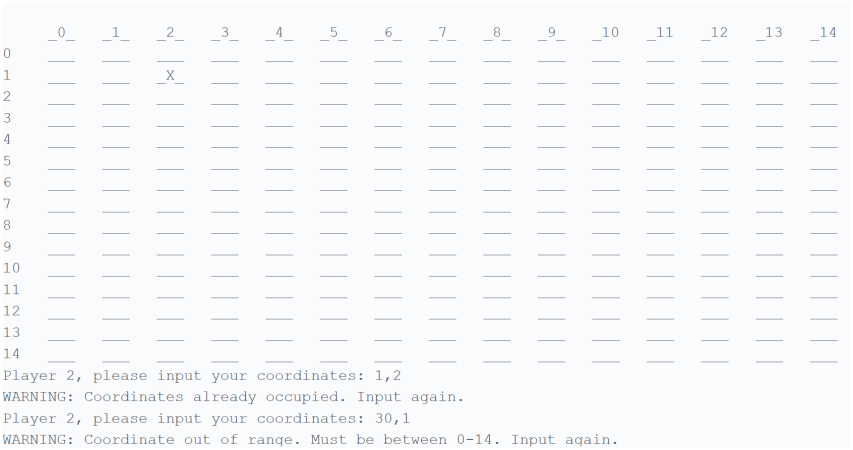

Figure 3.3: Handling error to allow players to continue the game whenever small mistakes were made.

The program validates each input to ensure it has correct format, falls within the 15x15 board boundaries and targets an unoccupied cell. If the input is invalid, the program warns user about the error and prompt the user to retry without crashing or resetting the game state (see figure 3.3).

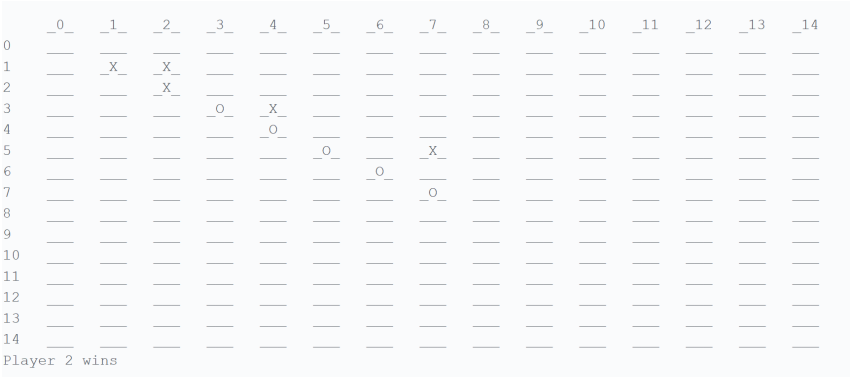

Figure 3.4: Gameplay stops when stop conditions are met (either player won or tie)

The gameplay stops when either player has five marks in a row (horizontally, vertically or diagonally) or all the cells in the board are occupied. The program also announces the result of the gameplay (see figure 3.4).

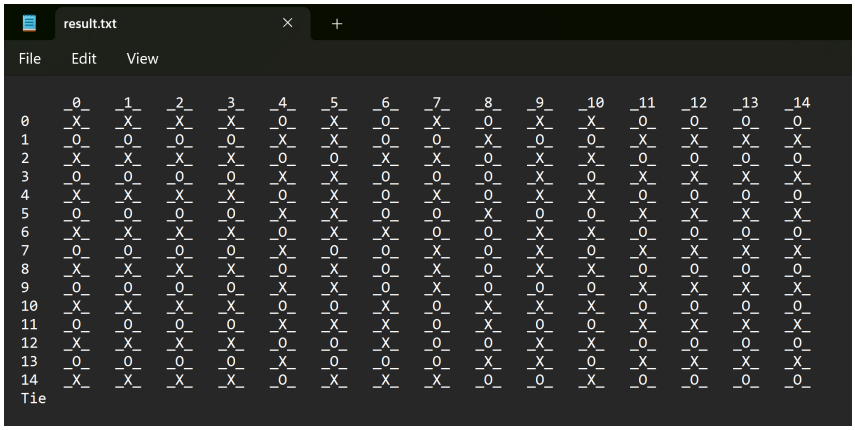

Figure 3.5: File result of a match ending with tie.

Furthermore, the project incorporates a file-writing feature that successfully saves the final game state, including the board configuration and outcome, to a text file (see figure 3.5).

3.2 Source code

The repository for the program can be found at: code

4 References

[1] Wikipedia, “Gomoku.” [Online]. Available: https://en.wikipedia.org/wiki/Gomoku