Vietnam National University, Ho Chi Minh City
University of Technology
Faculty of Computer Science and Engineering

Computer Architecture (CO2008)

ASSIGNMENT

Instructor(s): Nguyễn Thiên Ân
Class: CC02
Student: Nguyễn Hoàng Quân - 2352998 (Group 7)
Nguyễn Đình Khôi - 2352611 (Group 7)
Nguyễn Quốc Huy - 2352397 (Group 7)

Ho Chi Minh City, April 2025

Contents

1 Introduction

Gomoku, also known as Caro, is a traditional two-player board game that emphasizes strategic thinking and careful planning. The game originated in East Asia, particularly in Japan and China, and has since gained widespread popularity in many countries.

In Gomoku, two players alternately place their marks, either **X** or **O**, onto a square grid. The standard board size is typically 15×15, although larger grids can be used depending on the rules agreed upon by the players. The objective is to be the first to form a continuous line of five marks horizontally, vertically, or diagonally. Despite its simple rules, Gomoku offers considerable strategic depth, making it a timeless and intellectually stimulating game.

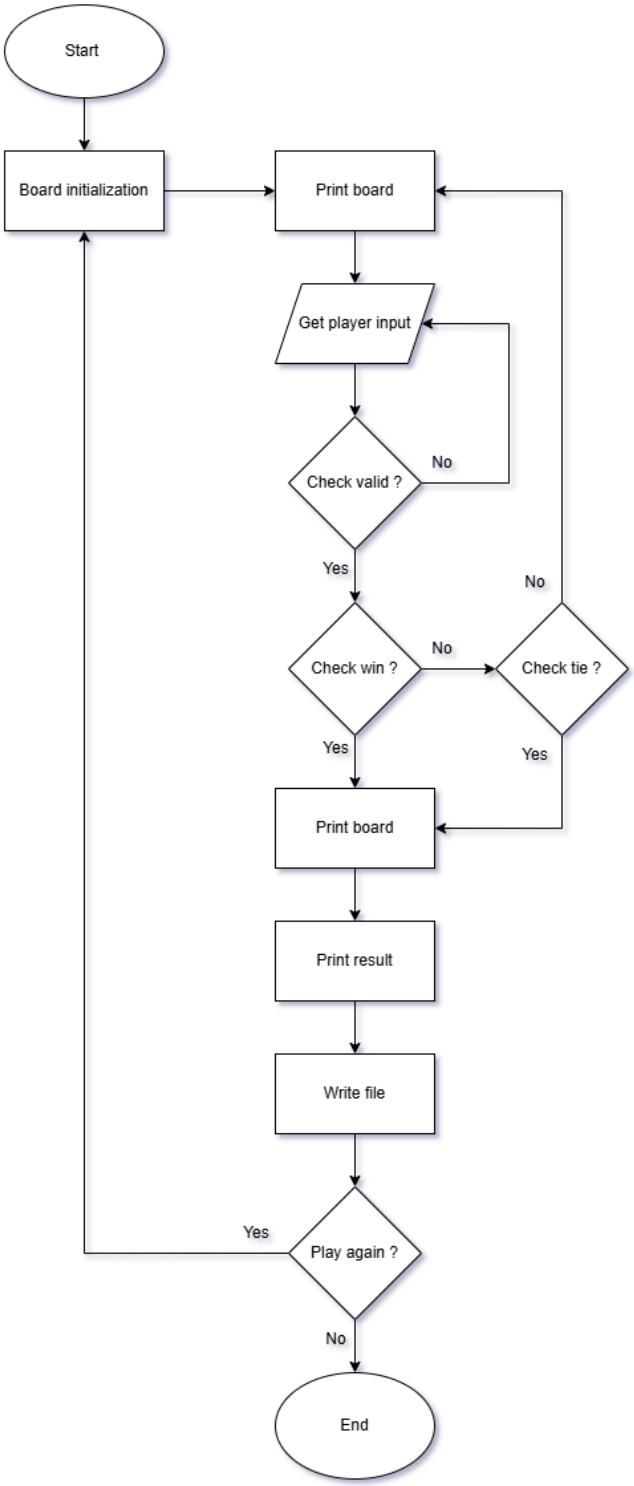
As part of this course, our project focuses on the development of a Caro game, which will be implemented on a 15×15 grid. The game will feature a user-friendly interface and allow two players to compete against each other. In addition, we will present and evaluate several test cases to thoroughly assess the game's functionality, ensuring that it works as intended under various conditions.

2 Workflow Design Overview

The main idea behind this assignment is to design a clear and systematic workflow that guides the process from initialization to final output. It begins with setting up the necessary environment and receiving user input, followed by careful processing of the data based on predefined logic and conditions. Each step is logically connected to the next, ensuring smooth and error-free operation. Special attention is paid to handling exceptions and validating inputs to enhance the reliability of the overall system. The design focuses on clarity, efficiency, and ease of understanding, making it easy to follow and implement.

The overall structure and sequence of operations are illustrated in the flow chart on the next page.

However, this is just a simple flow for the reader to understand how our program is run. In the next part, we will show you the way how we can check winning conditions, also handling exceptions in special cases, also clearly explaining some crucial parts in our code.


Start
Board initialization
Print board
Get player input
Check valid ?
No
Yes
No
Check win ?
No
Check tie ?
Yes
Yes
Print board
Print result
Write file
Yes
Play again ?
No
End

3 Board Initialization

Purpose

This function creates a game board for a new game. It technically sets all the values to a space character (' '), creating a blank, empty board state, ready for players to place their marks.

Explanation

The function uses a loop that iterates 225 times. For each iteration, it writes a space character (' ') to the current location, demonstrating a board cell, and moves to the next location. The process follows these steps:

- **Header Display:** The function first prints the column numbers as a header, helping the players visualize the positions on the board.
- **Row Iteration:** The function then iterates through each row (from 0 to 14). For each row:
  - It prints the row number with an appropriate spacing for alignment.
  - A nested loop iterates through each column (from 0 to 14) for that row.
  - For each column, the function fetches the character from the corresponding position in the board memory array and prints it, followed by a separator (such as a space or a pipe).
- **Newline and Footer:** After printing all columns for a row, the function prints a newline to move to the next row. Finally, after all rows are printed, a footer line is output, which completes the display of the board.

This step ensures that the game board is printed in a clear and structured format, with proper row and column markings, making it easy for players to interact with the game.

```
    0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
  +-------------------------------------------------------------+
 0  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
 2  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
 3  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
 4  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
 5  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
 6  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
 7  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
 8  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
 9  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
10  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
11  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
12  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
13  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
14  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
  +-------------------------------------------------------------+
```

4 Get Player Input and Display Values

4.1 Get Player Input

Purpose: To read the player's typed text input, expecting it in an `"x,y"` format, and convert this text into two distinct integer values representing the chosen row (x) and column (y) coordinates for their move.

Explanation:

The program starts by prompting the active player and then waits to read a line of text from the buffer (using `syscall 8`). The expectation is that the player types something like `"6,9"`. Once the text is received, the function parses it character by character:

- It first accumulates the digits before the comma, converting each ASCII digit ('0' through '9') into its corresponding integer value.
- It combines these digits by multiplying the current value by 10 and adding the new digit, allowing multi-digit numbers to be correctly constructed.
- After detecting the comma (ASCII code 44), it repeats the same digit-parsing process for the characters after the comma to obtain the column coordinate.
- Parsing stops when the newline character (ASCII code 10) is read, corresponding to the player pressing Enter.

An additional feature is implemented: if the player omits the number before or after the comma, the program automatically assigns a default value of 0 to the missing coordinate. For example:

- Input `",5"` is interpreted as $(0, 5)$,
- Input `"7,"` is interpreted as $(7, 0)$,
- Input `","` is interpreted as $(0, 0)$.

This ensures robustness against incomplete inputs and improves the flexibility of user interaction.

This approach guarantees that even two-digit inputs like "14,13" or incomplete formats are correctly processed into their intended numerical components.

4.2 Display Values

Purpose: To retrieve the character ('X', 'O', or space) representing the current state of a single specific cell from the game board's memory and print that character to the console, contributing to the visual representation of one part of a row during the full board display process.

Explanation:

This function is responsible for drawing the content of a single square onto the console grid. It does not display the entire board at once but rather displays one cell at a time and is repeatedly executed within nested loops (15 times for each row across 15 rows).

When `display_value` runs:

- It receives the current row ($\$t0$) and column ($\$t1$) values from the outer loop.

- It translates the 2D grid coordinates into a linear index suitable for accessing the 1D board array in memory using the formula:

$$\text{index} = (\text{row} \times 15) + \text{column} \tag{1}$$

- The function then adds this index to the starting memory address of the board to find the address of the specific cell.
- It uses a `Load Byte (lb)` operation to fetch the character ('X', 'O', or ' ') stored at that address into register $\$a0$.
- Using `syscall 11` (print character), the function outputs this character to the console.
- To maintain the visual grid structure, after printing the cell content, it prints a vertical separator `" | "` using `syscall 4`.

This process is repeated for each cell in a row, allowing the entire board to be printed row by row.

5 Algorithm for checking winner

As you know, in Gomoku (Caro) game, a player can get a win only when he or she get 5 in a row first, and Game will tie when no player win and the board is full.

Base on that idea we design our algorithm to use a counter to check if a player is win or not. Below is the concept of what we will implement.

- **(x,y)**
  This is the initial coordinate. We will traverse from this coordinate toward or backward after a player go their turn to check if they have won or not.
- **dx,dy**
  Because the program will have to check 8 directions to Know whether if Player has won or not. So we use dX, dY to move our coordinate and count the the number of char that each player has marked (check 5 or not) to turn on win flag.
- **Counter**
  Use to count 'X' and 'O'.
- **Save Player Char**
  use to compare to increase count

We will use loop to check after each player turn is recorded. At first, we just follow the direction of dX and dY represents, then we could get new coordinate $((x + dx), (y + dy))$, then we could use the formula as mentioned above to calculate index and take the value in the 1D-array that we have stored before with view to comparing.

$$\mathrm{x} \times 15 + \mathrm{y} \tag{2}$$

After then, if it reached boundaries or meet the sign that different from which current player is playing, our program will check in the opposite direction to examine winning conditions. We have also set up the function "check_direction" with intention to traverse forward and backward easily. The forward direction is as above, and for backward direction we just need to adjust $dx = dx \times (-1)$ and $dy = dy \times (-1)$. Then go through the loop again. If win_flag on, we would jump back where we have called jal then jump to print the winner, else go through the game loop again till we get the winner or tie result.

6 Exception Handling

Exception handling is implemented in the program to ensure stability and to guide users in providing valid input. The handled exceptions include:

- **Invalid Input Format** (`error_format`):
  If the user enters coordinates in an incorrect format (for example, missing a comma or entering invalid characters), the program displays the following message:

  ```
  Error: Invalid format. Please enter coordinates as 'x,y'.
  ```

  and prompts the user to re-enter the coordinates correctly.

  Detection mechanism: During input parsing, each character is read sequentially. The program verifies whether each character is a digit (ASCII codes from 48 to 57, corresponding to '0'–'9'). After reading the first number (representing the x-coordinate), it expects a comma (ASCII code 44). If the comma is missing or misplaced, subsequent reading of the input will fail to detect the newline character (ASCII code 10), triggering an invalid format error. This mechanism ensures that users strictly follow the 'x,y' format when entering coordinates.

- **Coordinates Out of Range** (`error_range`):
  If the entered coordinates are outside the valid range (0 to 14 for both axes), the program outputs:

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

  and asks the user to input valid coordinates within the allowed range.

  Detection mechanism: After successfully parsing the x and y coordinates from the input, the program compares each value individually against the permitted range [0, 14]. If either the x-coordinate or the y-coordinate is less than 0 or greater than 14, the error message is triggered. This range validation ensures that all moves stay within the playable board area and prevents illegal memory access.

- **Position Already Occupied** (`error_occupied`):
  If the selected position has already been occupied by a previous move, the program notifies the user with:

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

  and requests the user to choose a different empty position.

  Detection mechanism: The program calculates the memory address corresponding to the selected position using the formula $address = (x \times 15) + y$. Then it checks the content at that memory location. If the value stored there is not a space character (' '), it means that the position is already occupied, and the error message is triggered. This ensures that no player can overwrite an existing move.

These validation checks and error messages help the program avoid unexpected behavior and improve the user experience.

7 Write to File: result.txt

The `write_into_file` function is responsible for saving a complete record of the finished game to the `result.txt` file. It begins by opening this file in write mode, which creates the file if it does not exist or clears its contents if it already does. Upon opening, it obtains a unique file descriptor ID, which is stored in register `$s4`.

The function's core strategy is to first build the entire desired file content within a temporary memory area called the buffer. This process involves sequentially copying the standard board header, then iterating through all board rows and columns to append:

- Formatted row numbers,
- The actual character ('X', 'O', or space) retrieved from each cell in the main board memory,
- Visual separators such as `" | "` and newline characters.

After finishing the rows, it concludes with the board footer string.

Once the full board representation is constructed inside the buffer, the function proceeds to determine the game's final outcome. It cleverly checks whether it was invoked from the tie-handling code (by examining the saved return address) or determines the winner based on which player (`$s0`) made the last move. The appropriate result message ("Player 1 wins!", "Player 2 wins!", or "Tie!") is then located in memory and appended to the end of the buffer content.

Finally, the function calculates the total length of the constructed string in the buffer. It then uses a single system call to write the entire buffer content to the opened file using the saved file descriptor. Afterwards, it properly closes the file with another system call and restores the program's state (NULL) before returning control to the calling function, which asks the user if they would like to play another game.

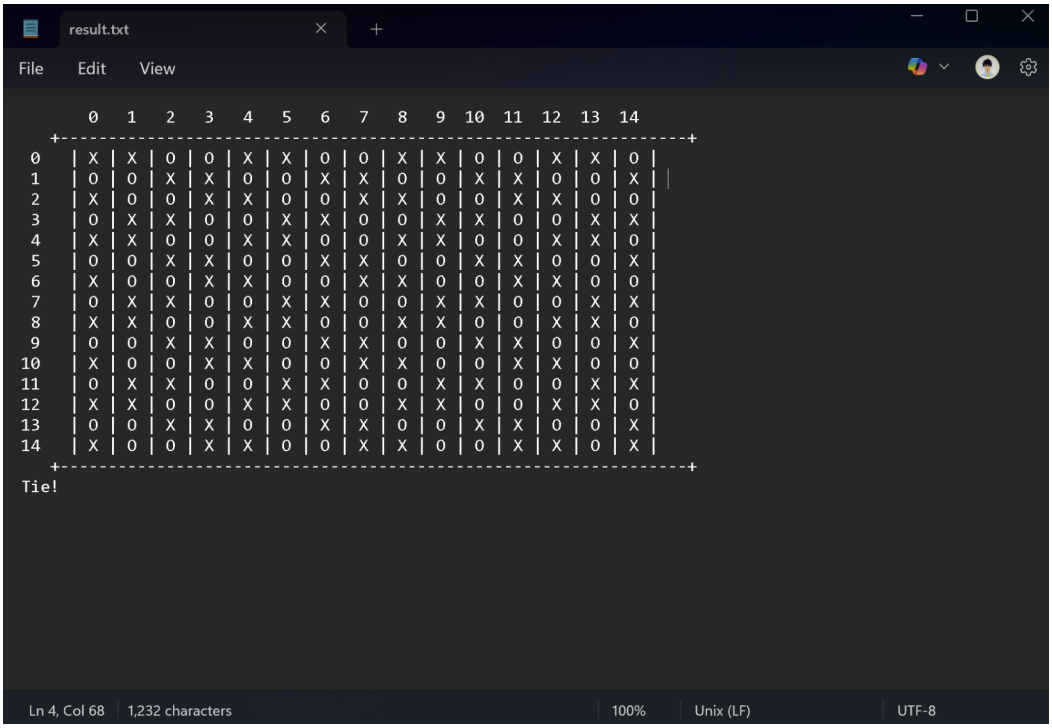
8 Testcase

- **Win_result**

```
Player 1, please input your coordinates: 4,0
      0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
   +-----------------------------------------------------------+
 0 | X | X |   |   | X |   |   |   |   |   |   |   |   |   |   |
 1 | O |   |   | X |   |   |   |   |   |   |   |   |   |   |   |
 2 | X |   | X |   |   |   | O |   |   |   |   |   |   |   |   |
 3 |   | X | X |   | O |   |   |   |   |   |   |   |   |   |   |
 4 | X |   | X |   |   | O | O |   |   |   |   |   |   |   |   |
 5 |   |   |   |   |   |   | O |   |   |   |   |   |   |   |   |
 6 |   |   |   |   | O |   |   |   |   | O |   |   |   |   |   |
 7 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
 8 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
 9 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
10 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
11 |   |   |   |   |   |   |   |   |   |   |   |   | O |   |   |
12 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
13 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
14 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
   +-----------------------------------------------------------+
Player 1 wins!
Play again? (1 for yes, 0 for no): 1
      0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
   +-----------------------------------------------------------+
 0 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
 1 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
 2 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
 3 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
 4 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
 5 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
 6 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
 7 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
 8 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
 9 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
10 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
11 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
12 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
13 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
14 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
   +-----------------------------------------------------------+
Player 1, please input your coordinates:
```

- **Tie_result**

```
Player 1, please input your coordinates: 14,14
      0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
  +-----------------------------------------------------------------+
0 | X | X | O | O | X | X | O | O | X | X | O | O | X | X | O |
1 | O | O | X | X | O | O | X | X | O | O | X | X | O | O | X |
2 | X | O | O | X | X | O | O | X | X | O | O | X | X | O | O |
3 | O | X | X | O | O | X | X | O | O | X | X | O | O | X | X |
4 | X | X | O | O | X | X | O | O | X | X | O | O | X | X | O |
5 | O | O | X | X | O | O | X | X | O | O | X | X | O | O | X |
6 | X | O | O | X | X | O | O | X | X | O | O | X | X | O | O |
7 | O | X | X | O | O | X | X | O | O | X | X | O | O | X | X |
8 | X | X | O | O | X | X | O | O | X | X | O | O | X | X | O |
9 | O | O | X | X | O | O | X | X | O | O | X | X | O | O | X |
10| X | O | O | X | X | O | O | X | X | O | O | X | X | O | O |
11| O | X | X | O | O | X | X | O | O | X | X | O | O | X | X |
12| X | X | O | O | X | X | O | O | X | X | O | O | X | X | O |
13| O | O | X | X | O | O | X | X | O | O | X | X | O | O | X |
14| X | O | O | X | X | O | O | X | X | O | O | X | X | O | X |
  +-----------------------------------------------------------------+
Tie!
Play again? (1 for yes, 0 for no):
```


result.txt
File
Edit
View
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
0 | X | X | O | O | X | X | O | O | X | X | O | O | X | X | O |
1 | O | O | X | X | O | O | X | X | O | O | X | X | O | O | X |
2 | X | O | O | X | X | O | O | X | X | O | O | X | X | O | O |
3 | O | X | X | O | O | X | X | O | O | X | X | O | O | X | X |
4 | X | X | O | O | X | X | O | O | X | X | O | O | X | X | O |
5 | O | O | X | X | O | O | X | X | O | O | X | X | O | O | X |
6 | X | O | O | X | X | O | O | X | X | O | O | X | X | O | O |
7 | O | X | X | O | O | X | X | O | O | X | X | O | O | X | X |
8 | X | X | O | O | X | X | O | O | X | X | O | O | X | X | O |
9 | O | O | X | X | O | O | X | X | O | O | X | X | O | O | X |
10 | X | O | O | X | X | O | O | X | X | O | O | X | X | O | O |
11 | O | X | X | O | O | X | X | O | O | X | X | O | O | X | X |
12 | X | X | O | O | X | X | O | O | X | X | O | O | X | X | O |
13 | O | O | X | X | O | O | X | X | O | O | X | X | O | O | X |
14 | X | O | O | X | X | O | O | X | X | O | O | X | X | O | X |
Tie!
Ln 4, Col 68
1,232 characters
100%
Unix (LF)
UTF-8

9 Conclusion

This project yielded several valuable lessons. First, it provided hands-on experience with MIPS assembly language, deepening understanding of low-level programming concepts, including memory management, input parsing, control flow, and system calls. Implementing player input handling, validation, and a simple game board through manual memory operations enhanced familiarity with how computers internally process and represent data.

Additionally, the project reinforced theoretical course concepts, offering a practical opportunity to apply core ideas in a real-world context. Developing a simple game in assembly language demanded meticulous attention to detail, structured thinking, and disciplined programming practices—skills critical for advanced topics in computer architecture and system-level programming.

Finally, the project illustrated fundamental principles of game development. It demonstrated how functional games can be built without high-level libraries or engines, relying solely on data structures, input/output operations, and control logic. This experience establishes a solid foundation for tackling more complex programming challenges in the future.

Overall, the project not only honed technical skills but also fostered a deeper appreciation for the elegance and intricacy of low-level programming.