To address the query, I'll provide a Finite State Machine (FSM) outline for the Gomoku game based on the requirements in the "Assignment_sem_242.pdf" document, followed by a breakdown into reasonably small procedures or macros for implementation in MIPS assembly language using the MARS simulator. The game is a two-player Gomoku on a 15x15 board, where players alternate placing 'X' (Player 1) or 'O' (Player 2) to achieve five in a row horizontally, vertically, or diagonally, with specific input and output requirements.

Finite State Machine Outline

The FSM organizes the game into distinct states that handle initialization, player turns, and game termination, ensuring all requirements—such as displaying the board, prompting for moves, validating input, checking win/tie conditions, and writing results—are met. Here's the FSM structure:

1. **State 0: Initialization**
    - **Purpose**: Set up the game environment.
    - **Actions**:
        - Initialize a 15x15 board (225 positions) with all positions set to '.' (empty).
        - Set the current player to Player 1.
        - Set the move count to 0.
    - **Transition**: Move to State 1 (Player Turn).
    - **Requirement Met**: "An empty board is displayed in the terminal at the beginning."
2. **State 1: Player Turn**
    - **Purpose**: Handle a single turn for the current player (Player 1 or 2).
    - **Actions**:
        - Display the current board state with row and column labels (0-14).
        - Prompt the current player (e.g., "Player 1, please input your coordinates").
        - Read input in the format "x,y" (e.g., "4,5" for row 4, column 5).
        - Validate input:
            - Ensure format is correct (two integers separated by a comma).
            - Check coordinates are within 0-14.
            - Verify the position is empty ('.').
            - If invalid, display an error message and re-prompt.
        - If valid:
            - Update the board with 'X' (Player 1) or 'O' (Player 2) at the position.
            - Increment move count.
            - Check for win: Five consecutive 'X's or 'O's horizontally, vertically, or diagonally from the last move.
            - If win detected, transition to State 2 (Game Over - Win).
            - If no win, check for tie: Move count equals 225 (board full).
            - If tie detected, transition to State 3 (Game Over - Tie).
            - If neither win nor tie, switch the current player (1 to 2, or 2 to 1) and remain in State 1.
    - **Requirements Met**:
        - "A prompt for each player is needed to begin a move."
        - "The program must then receive a coordinate input in the form of x,y."

        - "The program automatically checks for any form of incorrect input."
        - "The board updated with the new move must be shown right after the player input."

3. **State 2: Game Over (Win)**
    - **Purpose**: Handle a winning condition.
    - **Actions**:
        - Display the final board.
        - Print "Player 1 wins" or "Player 2 wins" based on the current player.
        - Write the final board and result to "result.txt".
        - Exit the program.
    - **Requirements Met**:
        - "In the case of a win, a final statement Player 1 wins or Player 2 wins is shown."
        - "The program must also write the final version of the board and the final statement into a text file name 'result.txt'."

4. **State 3: Game Over (Tie)**
    - **Purpose**: Handle a tie condition.
    - **Actions**:
        - Display the final board.
        - Print "Tie".
        - Write the final board and result to "result.txt".
        - Exit the program.
    - **Requirements Met**:
        - "A tie would instead show Tie."
        - "The program must also write the final version of the board and the final statement into a text file name 'result.txt'."

This FSM loops in the Player Turn state until a win or tie occurs, then transitions to the appropriate Game Over state, ensuring a clear flow that meets all specified requirements.

Procedure/Macro Breakdown

To implement this FSM in MIPS using the MARS simulator, the program is broken into modular procedures. Each procedure handles a specific task, making the code manageable and reusable. Below is the breakdown:

1. `init_board`

- **Purpose**: Initialize the 15x15 board.
- **Input**: None.
- **Output**: None (modifies global board array).
- **Tasks**:
    - Allocate a 225-byte array in the data segment (e.g., `board: .space 225`).
    - Fill all 225 positions with '.' (ASCII 46) using a loop.
- **MIPS Notes**: Use a loop with `sb` (store byte) to set each position.

2. `display_board`

- **Purpose**: Print the current board state to the terminal.
- **Input**: None (accesses global board array).
- **Output**: Board displayed in terminal.
- **Tasks**:
  - Print column labels: "Columns: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14".
  - For each row (0 to 14):
    - Print "Row X: " where X is the row number (adjust spacing for 0-9 vs. 10-14).
    - Print 15 board positions (e.g., ". ", "X ", "O ") separated by spaces.
    - Print a newline.
- **MIPS Notes**: Use syscall 4 (print string) and syscall 1 (print integer) for labels, syscall 11 (print character) for board cells.

3. `get_move`

- **Purpose**: Prompt for and validate player input.
- **Input**: Current player number (1 or 2).
- **Output**: Valid x, y coordinates (in registers, e.g., $v0, $v1).
- **Tasks**:
  - Loop until valid input:
    - Print prompt (e.g., "Player 1, please input your coordinates").
    - Read input string (e.g., "4,5") using syscall 8.
    - Call `parse_input` to extract and validate x, y.
    - If invalid, print error message (e.g., "Invalid input, try again") and repeat.
    - If valid, return x, y.
- **MIPS Notes**: Manage stack for temporary storage, use registers for return values.

4. `parse_input`

- **Purpose**: Parse and validate the input string.
- **Input**: Input string buffer.
- **Output**: x, y coordinates if valid, or error code (e.g., -1 in $v0).
- **Tasks**:
  - Find the comma separator.
  - Extract substrings before and after the comma.
  - Convert to integers (check for digits only).
  - Validate:
    - x and y are between 0 and 14.
    - Board position at y*15 + x is '.' (empty).
  - Return x, y or error code.
- **MIPS Notes**: Use loops to process characters, perform ASCII-to-integer conversion.

5. `update_board`

- **Purpose**: Place a piece on the board.
- **Input**: x, y coordinates, current player (1 or 2).
- **Output**: None (modifies board array).
- **Tasks**:

    - Calculate index = y * 15 + x.
    - Store 'X' (ASCII 88) if player 1, 'O' (ASCII 79) if player 2 at board[index].
- **MIPS Notes**: Use `sb` to update the byte at the calculated address.

6. `check_win`

- **Purpose**: Check if the last move resulted in a win.
- **Input**: x, y of last move, current player.
- **Output**: 1 if win, 0 if no win (e.g., in $v0).
- **Tasks**:
    - Define directions: horizontal (1,0), vertical (0,1), diagonal (1,1), diagonal (1,-1).
    - For each direction:
        - Count consecutive pieces in positive direction (e.g., right, down).
        - Count in negative direction (e.g., left, up).
        - Total = positive + negative + 1 (current position).
        - If total >= 5, return 1.
    - Return 0 if no direction has 5 or more.
- **MIPS Notes**: Use loops with boundary checks (0-14), compare bytes with 'X' or 'O'.

7. `check_tie`

- **Purpose**: Check if the game is a tie.
- **Input**: Move count.
- **Output**: 1 if tie, 0 if not (e.g., in $v0).
- **Tasks**:
    - Return 1 if move count == 225, else 0.
- **MIPS Notes**: Simple comparison with constant 225.

8. `switch_player`

- **Purpose**: Toggle between players.
- **Input**: Current player (1 or 2).
- **Output**: Updated current player.
- **Tasks**:
    - If current player is 1, set to 2; if 2, set to 1.
- **MIPS Notes**: Use conditional branch or XOR for toggle.

9. `write_to_file`

- **Purpose**: Write the final board and result to "result.txt".
- **Input**: Result string (e.g., "Player 1 wins" or "Tie").
- **Output**: None (creates/updates file).
- **Tasks**:
    - Open "result.txt" (syscall 13, write mode).
    - Write column labels.
    - Write each row as in `display_board`.
    - Write result string.
    - Close file (syscall 16).

- **MIPS Notes**: Use syscall 15 for writing, manage file descriptor.

Main Program Flow

In MIPS, the main program ties these procedures together:

```
main:
    # Initialization
    jal init_board
    li $s0, 1          # current_player = 1
    li $s1, 0          # move_count = 0

game_loop:
    jal display_board
    move $a0, $s0      # Pass current player
    jal get_move       # Returns x in $v0, y in $v1
    move $a0, $v0      # x
    move $a1, $v1      # y
    move $a2, $s0      # current_player
    jal update_board
    addi $s1, $s1, 1   # move_count++

    move $a0, $v0      # x
    move $a1, $v1      # y
    move $a2, $s0      # current_player
    jal check_win
    beq $v0, 1, win    # If win, go to win handling

    move $a0, $s1      # move_count
    jal check_tie
    beq $v0, 1, tie    # If tie, go to tie handling

    jal switch_player  # Update $s0
    j game_loop

win:
    jal display_board
    # Print "Player X wins" based on $s0
    la $a0, win_msg1   # "Player 1 wins"
    beq $s0, 1, print_win
    la $a0, win_msg2   # "Player 2 wins"
print_win:
    li $v0, 4
    syscall
    jal write_to_file
    j exit

tie:
    jal display_board
    la $a0, tie_msg    # "Tie"
    li $v0, 4
    syscall
```

```
    jal write_to_file

exit:
    li $v0, 10
    syscall
```

Notes

- **Data Segment**: Define `board: .space 225`, `current_player: .word 1`, `move_count: .word 0`, and strings for prompts and messages.
- **Registers**: Use saved registers ($s0, $s1) for persistent variables, manage stack in procedures.
- **Requirements**: All specified prompts, input formats, validations, and outputs are handled as per the assignment.

This structure provides a clear, modular implementation of Gomoku in MIPS, fulfilling the assignment's requirements while maintaining readability and maintainability.