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

COMPUTER ARCHITECTURE (CO2007)

Assignment (Semester: 231)

BATTLESHIP

Advisor: Băng Ngọc Bảo Tâm, CSE-HCMUT
Student: Trần Đình Đăng Khoa - 2211649.

HO CHI MINH CITY, NOVEMBER 2023

Contents

1 Introduction

Battleship (also known as Battleships or Sea Battle) is a strategy type guessing game for two players. It is played on ruled grids (paper or board) on which each player's fleet of warships are marked. The locations of the fleets are concealed from the other player. Players alternate turns calling "shots" at the other player's ships, and the objective of the game is to destroy the opposing player's fleet.

Figure 1: Example of the classic battleship game

1.1 Description

The game is played on four grids, two for each player. The grids are typically square - usually 10×10 - and the individual squares in the grid are identified by letter and number. On one grid the player arranges ships and records the shots by the opponent. On the other grid, the player records their own shots.

Before play begins, each player secretly arranges their ships on their primary grid. Each ship occupies a number of consecutive squares on the grid, arranged either horizontally or vertically. The number of squares for each ship is determined by the type of ship. The ships cannot overlap (i.e., only one ship can occupy any given square in the grid). The types and numbers of ships

allowed are the same for each player. These may vary depending on the rules. The ships should be hidden from players sight and it's not allowed to see each other's pieces. The game is a discovery game which players need to discover their opponents ship positions.

After the ships have been positioned, the game proceeds in a series of rounds. In each round, each player takes a turn to announce a target square in the opponent's grid which is to be shot at. The opponent announces whether or not the square is occupied by a ship. If it is a "hit", the player who is hit marks this on their own or "ocean" grid (with a red peg in the pegboard version), and announces what ship was hit. The attacking player marks the hit or miss on their own "tracking" or "target" grid with a pencil marking in the paper version of the game, or the appropriate color peg in the pegboard version (red for "hit", white for "miss"), in order to build up a picture of the opponent's fleet.

If all of a player's ships have been sunk, the game is over and their opponent wins.

2 MIPS Assembly Implementation

2.1 Introduction

In this project, we aim to emulate the Battleship game using the *MIPS* assembly language. Due to resource constraints and to keep things simple, we will work with a smaller grid size which is **7**×**7**

A ship location is indicated by the coordinates of the bow and the stern of the ship (row_{bow}, $\text{column}_{\text{bow}}$, $\text{row}_{\text{stern}}$, $\text{column}_{\text{stern}}$).

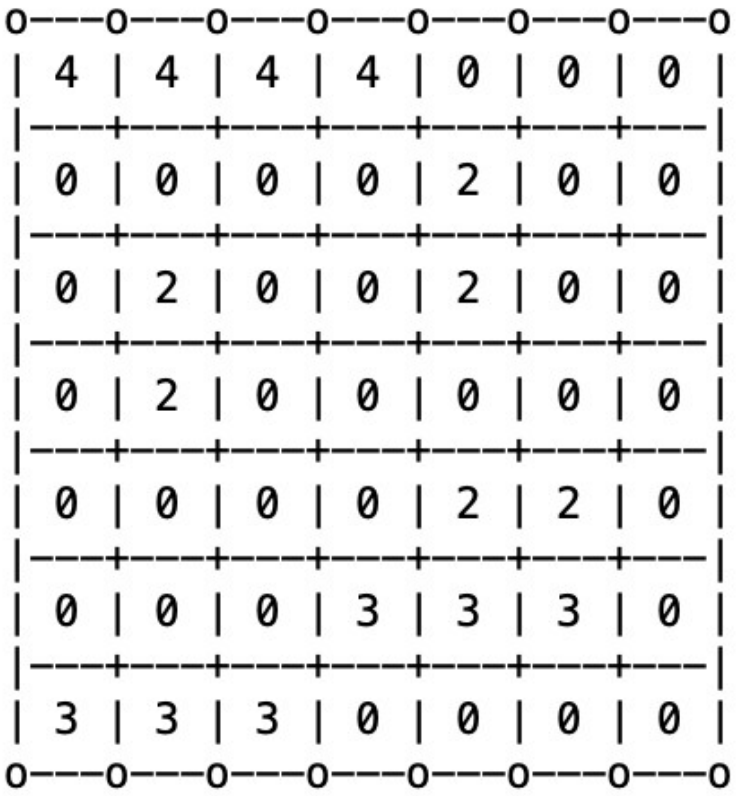

Figure 2: Map

In the above figure, the coordinates of the **4**×**1** ship is `0 0 0 3`; the green **3**×**1** ships are `5 3 5 5` and `6 0 6 2`; the **2**×**1** ships are `1 4 2 4`, `2 1 3 1`, and `4 4 4 5`.

The initial phase involves players arranging their ship positions. The program prompts players to input ship locations on a map where each cell holds either "the ship's size" or "0", indicating an empty/unoccupied space. Specifically, a cell contains the ship's size if occupied, and 0 if empty. During the setup phase, players must precisely position a predefined number of ships, each of the same size. In detail, each player will have 3 **2**×**1** ships, 2 **3**×**1** ships, and 1 **4**×**1** ship. Note that ships can not overlap with each other.

2.2 Overall Structure

1. **Data Section (.data):**
   The data section serves as the preamble to the program, laying the foundation for essential constants, character values, and game-related data. This section is crucial for establishing the groundwork necessary for the subsequent execution of the game.

   - Constants:
     Various system calls are defined to facilitate operations such as *printing* integers, strings, *reading* integers, strings, and exiting. Additionally, character values are assigned to digits, letters, and spaces, enhancing the readability and manageability of the code.
   - Game-related Constants:
     This subsection introduces constants that play a pivotal role in the game's mechanics. Values such as *player turns*, the *number of ships*, the *number of cells*, and *input lengths* are defined to ensure consistency and facilitate modifiability.
   - Data and Variables:
     Arrays for player A and player B maps are initialized, providing a structured representation of the game state. Strings containing the *game title*, *rules*, *prompts*, and *messages* are declared, contributing to a more comprehensible and user-friendly gaming experience.
     Numerous variables are introduced to store critical information, including the *grid size*, *ship coordinates*, and *shot coordinates*. These variables serve as dynamic entities, adapting to the evolving state of the game during execution.

2. **Text Section (.text):**
   The text section constitutes the heart of the program, containing the main program logic, function calls, and the structural components that orchestrate the flow of the game. It encapsulates the core functionalities responsible for the game's initiation, progression, and termination.

   - Main Section:
     The main section initializes registers, loads addresses for player maps, and orchestrates the initial setup of the game. It sets the stage for the subsequent execution of the game logic.
   - Function Calls:
     Function calls to `game_menu`, `rules_screen`, and `print_maps` contribute to the modular design of the code. These functions encapsulate specific functionalities, promoting code readability and maintainability.
   - Game Setup:
     The code calls the `read_ship` function twice, allowing players to input ship positions for player A and player B. After all the ships have been placed, the `print_maps` function is invoked to visualize the current game state.
   - Game Loop:
     The core game loop orchestrates the turn-based shooting mechanism, engaging players in strategic exchanges. Within the loop, players take turns calling the `read_shot` function to input coordinates for targeting their opponent's fleet. After each successful shot, the program evaluates the game state to determine if the current player has won. If a player emerges victorious, the game loop concludes, and the `print_maps` function

is invoked to showcase the final game board with updated ship positions and the winner's triumph. This meticulous updating of the visual representation ensures that players are presented with a clear and conclusive view of the game's outcome.

- Exit Section:
  The program concludes with an `exit` section, invoking a system call to terminate the execution gracefully. This ensures proper closure and resource release.

3. **Helper Functions**
   This subsection delves into the specifics of key functions such as `read_ship` and `read_shot`. These functions are responsible for *acquiring* user input, *validating* it, and *facilitating the corresponding game actions*.

2.3 Constants and Definitions

1. **System Calls**:
   - `SYS_PRINT_INT` : System call for printing an integer (value: 1).
   - `SYS_PRINT_STRING`: System call for printing a string (value: 4).
   - `SYS_READ_INT` : System call for reading an integer (value: 5).
   - `SYS_READ_STRING` : System call for reading a string (value: 8).
   - `SYS_EXIT` : System call for program exit (value: 10).
   - `SYS_PRINT_CHAR` : System call for printing a character (value: 11).
   - `SYS_READ_CHAR` : System call for reading a character (value: 12).
2. **Character Constants:**
   Constants representing ASCII values for characters:
   - `char_0` to `char_6`: ASCII values for the characters '0' to '6'.
   - `char_N`: ASCII value for the character 'N'.
   - `char_Q`: ASCII value for the character 'Q'.
   - `char_R`: ASCII value for the character 'R'.
   - `char_Y`: ASCII value for the character 'Y'.
   - `char_n`: ASCII value for the character 'n'.
   - `char_q`: ASCII value for the character 'q'.
   - `char_r`: ASCII value for the character 'r'.
   - `char_y`: ASCII value for the character 'y'.
   - `char_space`: ASCII value for the space character.
3. **Player Turn and Game Constants:**
   - `player_a_turn`: Value representing Player A's turn (value: 0).
   - `player_b_turn`: Value representing Player B's turn (value: 1).
   - `number_of_ships`: Total number of ships each player has (value: 6).
   - `number_of_cells`: Total number of cells on the grid (value: 49).
   - `input_coordinates_length_shot`: Length of input for shot coordinates (value: 4).

- `input_coordinates_length_ship`: Length of input for ship coordinates (value: 8).

4. **Player Maps and Grid Size:**
   - `player_a_map`: An array representing the map for Player A, initialized with zeros.
   - `player_b_map`: An array representing the map for Player B, also initialized with zeros.
   - `grid_size`: Word variable storing the size of the grid (value: 7).
5. **Coordinate Spaces:**
   - `ship_coordinates`: Space reserved for storing ship coordinates during input.
   - `shot_coordinates`: Space reserved for storing shot coordinates during input.

These directives and data declarations provide symbolic names for system calls, characters, game constants, and storage spaces, making the MIPS assembly code more readable and maintainable. The constants and data structures defined here are used throughout the program to enhance code clarity and facilitate updates.

2.4 User Interface

1. **Game Menu:**

   The game menu is displayed to the user with ASCII art, creating a visually appealing and thematic presentation. The menu includes a title and options for the user to proceed or start the game. The ASCII art in the title creates a distinctive and engaging visual experience, making the menu more enjoyable for the user. The options to proceed or start the game are presented clearly to guide the user through the initial steps.

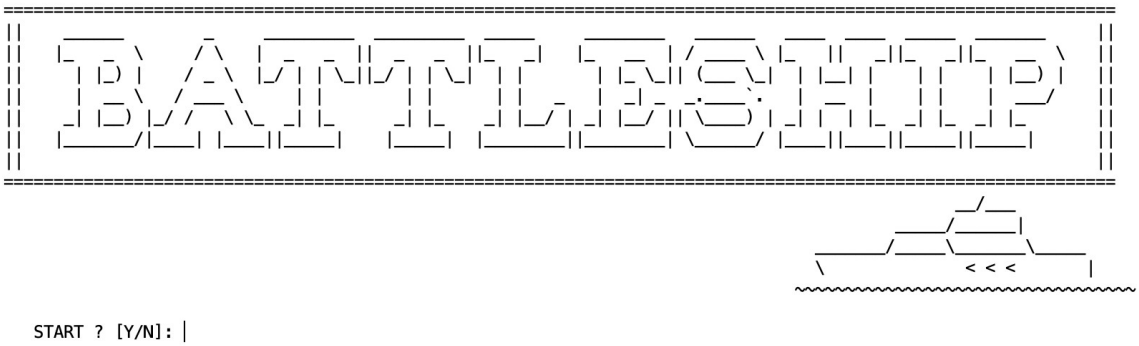

Figure 3: Game Menu Interface

2. **Rules Screen:**

   The rules screen is presented using ASCII art to provide an immersive and visually interesting display of the game rules. The ASCII art includes a title, setup phase instructions, gameplay details, update board information, winning conditions, and specific rules about ship placement and shooting. Each section is delineated with ASCII borders and design elements, making it easy for the user to navigate and comprehend the rules of the game. The use of ASCII characters enhances the aesthetic appeal of the rules screen, making it both informative and visually engaging for the player.

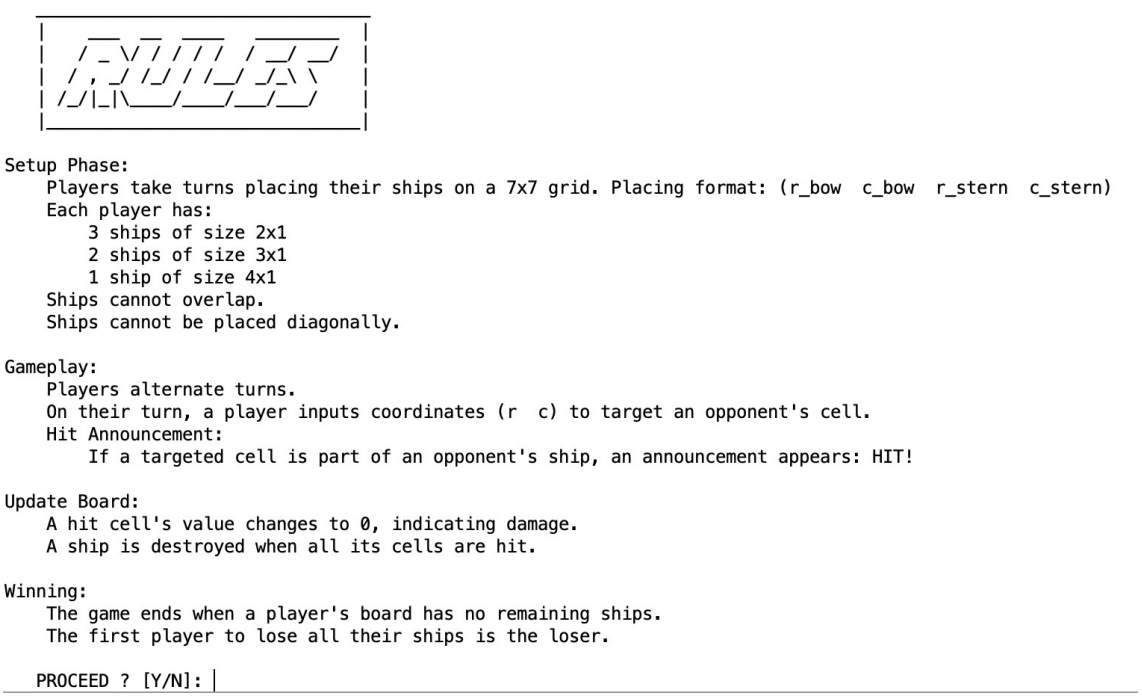

Figure 4: Rules Screen Interface

3. **Placing Ships Interface:**

```
! ENTER SHIPS LOCATIONS !~
Input format: [r_bow  c_bow  r_stern  c_stern]
[Q]: Quit to main menu
[R]: Restart
0 0 0 3
Ship position confirmed.
0 0 0 8
Invalid coordinates format. Try again.
0 0 3 0
Invalid ship size. Try again.
0 0 2 0
Ships cannot overlap. Try again.
1 1 2 2
Ships cannot be placed diagonally. Try again.
```

Figure 5: Placing Ships Interface

4. **Shooting and Endgame Interface:**

```
Enter shot location
Input format: [r  c]
[Q]: Quit to main menu
[R]: Restart

!!! PLAYER A TURN !!!
0 0
 O_O HIT! O_O

!!! PLAYER B TURN !!!
0 5
 T.T MISS! T.T

!!! PLAYER A TURN !!!
1 0
 O_O HIT! O_O

!!! PLAYER A WON THE GAME. CONGRATULATIONS !!!
    o---o---o---o---o---o---o---o        o---o---o---o---o---o---o---o
    |         Player A          |        |         Player B          |
    o---o---o---o---o---o---o---o        o---o---o---o---o---o---o---o
    | 2 | 0 | 0 | 0 | 0 | 0 | 0 |        | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
    |---+---+---+---+---+---+---|        |---+---+---+---+---+---+---|
    | 2 | 0 | 0 | 0 | 0 | 0 | 0 |        | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
    |---+---+---+---+---+---+---|        |---+---+---+---+---+---+---|
    | 0 | 0 | 0 | 0 | 0 | 0 | 0 |        | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
    |---+---+---+---+---+---+---|        |---+---+---+---+---+---+---|
    | 0 | 0 | 0 | 0 | 0 | 0 | 0 |        | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
    |---+---+---+---+---+---+---|        |---+---+---+---+---+---+---|
    | 0 | 0 | 0 | 0 | 0 | 0 | 0 |        | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
    |---+---+---+---+---+---+---|        |---+---+---+---+---+---+---|
    | 0 | 0 | 0 | 0 | 0 | 0 | 0 |        | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
    |---+---+---+---+---+---+---|        |---+---+---+---+---+---+---|
    | 0 | 0 | 0 | 0 | 0 | 0 | 0 |        | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
    o---o---o---o---o---o---o---o        o---o---o---o---o---o---o---o

-- program is finished running --
```

Figure 6: Shooting and Endgame Interface

5. **Player's Maps (For Grader):**

In addition to the player-facing interface, there is a dedicated section for the grader to inspect the initial maps. These maps, represented as matrices, serve as the starting configuration for each player's game board. The matrix values denote ship placements and empty cells. This section aids the grader in evaluating the correctness of the initial game state, ensuring that the ships are appropriately positioned and adhere to the specified rules. The clarity of the ASCII art in the initial maps facilitates a quick and accurate assessment, streamlining the grading process.

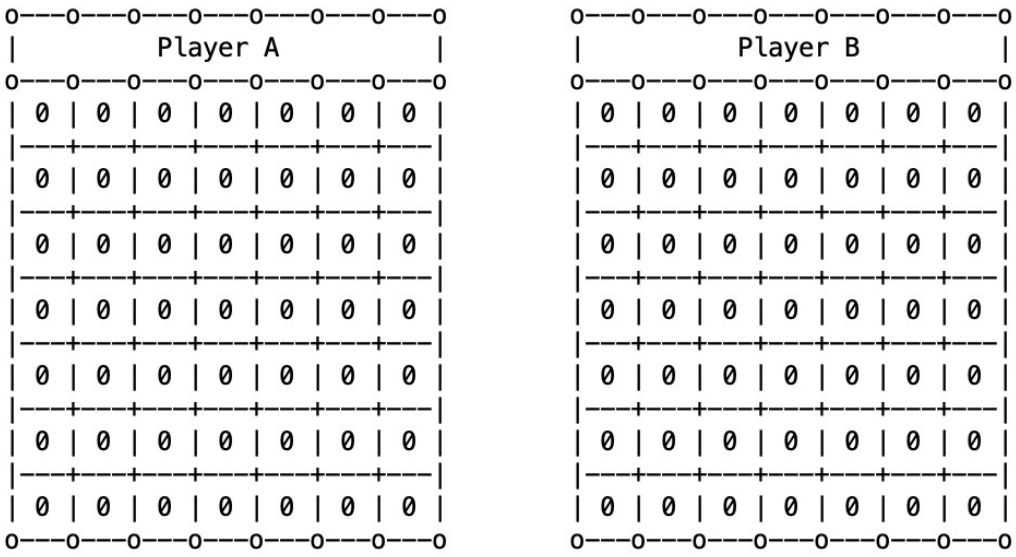

Figure 7: Initial Maps Interface

2.5 Game Logic

Let's break down the main functions, focusing on how players take turns placing ships and shooting, including input validation:

1. **Initialization:**

   `$s0`: Grid Size.
   `$s1`: **4×1** ships quantity (value: 1).
   `$s2`: **3×1** ships quantity (value: 2).
   `$s3`: **2×1** ships quantity (value: 3).
   `$s4`: Player A's map.
   `$s5`: Player B's map.
   `$s7`: Player A's turn.

   Calls `game_menu`, `rules_screen`, and `print_maps` to display the menu, rules, and initial maps.

2. **Player A Places Ships:**

   Read Ship Coordinates for Player A:

   - Calls `read_ship` function with the label `player_a_place_turn`.
   - Validates ship placement input using a series of checks (format, size, overlap, etc.).

3. **Player B Places Ships:**

   Read Ship Coordinates for Player B:

   - Calls `read_ship` function with the label `player_b_place_turn`.
   - Similar to Player A, validates ship placement input.

4. **Player A Takes a Shot:**

   Read Shot Coordinates for Player A:

   - Calls `read_shot` function and displays Player A's label (`player_a_place_turn`).
   - Validates shot input using similar checks as ship placement.

5. **Player B Takes a Shot:**

   Read Shot Coordinates for Player B:

   - Calls `read_shot` function and displays Player B's label (`player_b_place_turn`).
   - Validates shot input.

6. **Endgame Status Checking:**

   Check for Win/Loss:

   - After each shot, the game checks the status to determine if one of the players has won.
   - Calls `endgame_status_checking` to check if all ships of the opponent have been hit.

7. **Printing Maps:**

   Print Current State: Calls `print_maps` to display the updated maps after ship placements and shots.

8. **Exit:**

   Exiting the Game: The game can be exited at any time by choosing the option to quit (`[Q]` or `[q]`).

2.6 Helper Functions

1. `GAME_MENU:`

   The `game_menu` function serve the purpose of displaying a menu for the game and allowing the player to start or exit the game based on their input. Let's break down the key components:

   - Display Menu Titles:
     - The function uses a series of `la` (*load address*) instructions to load strings (`title_0` to `title_12`) representing different lines of the menu title.
     - It prints these titles to the console using the `SYS_PRINT_STRING` system call.
   - Display Start Prompt:
     It prints the string stored at the address `start` to prompt the player to start the game.
   - User Input:
     - It enters a loop (`start_loop`) that prompts the user to input a character.
     - The input character is stored in `$t0`, and an additional read is performed to consume the `newline (\n)` character.
   - Processing User Input:

     It checks the input character and proceeds accordingly:
     - If the character is `'y'` or `'Y'`, it jumps to the label `game_menu_proceed`, suggesting that the player wants to *proceed* with the game.
     - If the character is `'n'` or `'N'`, it jumps to the label `exit`, indicating that the player wants to *exit* the game.
     - If the input is anything else, it *continues* the loop (`start_loop`) to get a valid input.
   - End of the Function:

     The function ends by jumping to the `jr $ra` instruction, which returns control to the calling function (`main`).

2. `RULES_SCREEN:`

   The `rules_screen` function serve the purpose of displaying a set of rules to the player and allowing them to choose whether to proceed or return to the game menu based on their input. Let's break down the key components:

   - Display Rules:

  - The function loads a series of strings (`rule_0` to `rule_24`) representing different lines of the rules.
  - It prints these rules to the console using the `SYS_PRINT_STRING` system call.
- Display Proceed Prompt:

  It prints the string stored at the address `proceed` to prompt the player to either *proceed* or *return* to the game menu.
- User Input:
  - It enters a loop (`proceed_loop`) that prompts the user to input a character.
  - The input character is stored in *$t0*, and an additional read is performed to consume the `newline (\n)` character.
- Processing User Input:

  It checks the input character and proceeds accordingly:
  - If the character is `'y'` or `'Y'`, it jumps to the label `rules_screen_proceed`, suggesting that the player wants to *proceed* with the game.
  - If the character is `'n'` or `'N'`, it jumps to the label `main`, indicating that the player wants to *return* to the game menu.
  - If the input is anything else, it *continues* the loop (`proceed_loop`) to get a valid input.
- End of the Function:

  The function ends by jumping to the `jr $ra` instruction, which returns control to the calling function (`main`).

3. `PRINT_MAPS:`

   The `print_maps` function have the purpose of printing a game map to the console. It is a textual representation of a grid-based map. Let's break down the key components:

   - Print Header: The function starts by printing a header that includes visual separators (`grid_bar`) and a label for the player (`player_label`).
   - Initialization: It initializes variables (`$t2`, `$t3`, and `$t0`) that is to be related to map dimensions and loop control.
   - Print Map Rows: The core functionality involves a loop (`print_map`) for printing each row of the map.
   - Print Grid Elements: Within the row printing loop, the function prints various elements, including grid helpers (`grid_helper`), actual values from the map, and visual separators (`grid_vert`).

- Print Row Transitions: Transitions between different parts of the row are marked with the `grid_trans` string.
- Loop Control and Row End: The function carefully manages loop control variables (`$t1`, `$t4`) to control the number of rows printed. It ensures proper termination with the `print_row_loop_end` label.
- Print Footer: A footer is added with `newline` characters and another visual separator (`grid_bar`).

4. `READ_SHIP:`

   The `read_ship` function is responsible for obtaining and validating user input for placing ships on a game map. The input is expected in the format of ship coordinates, and the function performs various checks to ensure the input adheres to the game's rules.

   - Print Prompt: The function starts by printing a prompt (`prompt_ship`) to guide the user in entering ship coordinates.
   - Input Loop: The function enters a loop (`read_ship_loop`) to continuously read and validate ship coordinates until a valid input is provided.
   - Coordinate Validation:
     - It checks for specific user options (`[R], [r], [Q], [q]`) to either proceed with ship placement or return to the game menu.
     - Format checking to ensure correct input structure.
     - Value checking to verify that coordinates are within valid ranges.
     - Diagonal checking to restrict diagonal ship placement.
   - Overlap Checking: The function contains a section for checking whether the entered ship coordinates overlap with existing ships on the map.
   - Placing Ships: Once the input passes all validation checks, the function proceeds to place the ship on the game map.
   - Error Handling: The function includes various error messages (`invalid_format`, `invalid_size`, `invalid_placement`, `invalid_location`) to notify the user of input errors.

5. `READ_SHOT:`

   The `read_shot` function handles the process of obtaining, validating, and processing user input for shooting at the opponent's game map. It ensures that the entered shot coordinates are valid and performs the corresponding actions based on the game's rules.

- Print Prompt: The function starts by printing a prompt (`prompt_shot`) to guide the user in entering shot coordinates.
- Player Turn Determination: The function determines the current player's turn based on the value of `$s7`. It prompts the appropriate message for the player's turn (`player_a_place_turn` or `player_b_place_turn`).
- Input Loop: The function enters a loop (`read_shot_loop`) to continuously read, validate, and process shot coordinates until a valid input is provided.
- Coordinate Validation:
  - It checks for specific user options (`[R], [r], [Q], [q]`) to either proceed with shooting or return to the game menu. Further checks include format checking, ensuring only one space is present, and validating that coordinates are within valid ranges.
- Shooting Logic:
  - The function distinguishes between Player A's and Player B's turns.
  - It processes the shot coordinates, checks for hits or misses, updates the opponent's map accordingly, and proceeds to endgame status checking.
- Endgame Status Checking: The function checks whether the game has ended by verifying whether all cells on the opponent's map have been hit.
- Endgame Messages: It prints messages (`a_wins` or `b_wins`) based on the winner of the game.

3 Conclusion

Upon completion of this assignment, students demonstrate proficiency in utilizing the MARS MIPS simulator and applying essential MIPS assembly instructions. The implementation incorporates arithmetic and data transfer instructions, showcasing the students' command over low-level programming concepts. Moreover, the inclusion of conditional branch and unconditional jump instructions reflects a comprehensive understanding of control flow mechanisms in MIPS assembly.

Furthermore, the implementation effectively leverages procedures, emphasizing modular and structured programming practices. The students' ability to navigate the intricacies of the MIPS assembly language is evident throughout the project, culminating in a successful emulation of the Battleship game. This assignment not only serves as a practical application of theoretical knowledge but also showcases the students' aptitude in employing MIPS assembly for real-world programming tasks.

References

[1] Wikipedia. (n.d.). *Battleship (game)*. https://en.wikipedia.org/wiki/Battleship (game) Accessed 2023.