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**

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

Explain the purpose of constants defined using .eqv (e.g., system calls, characters, and game-related constants).

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.

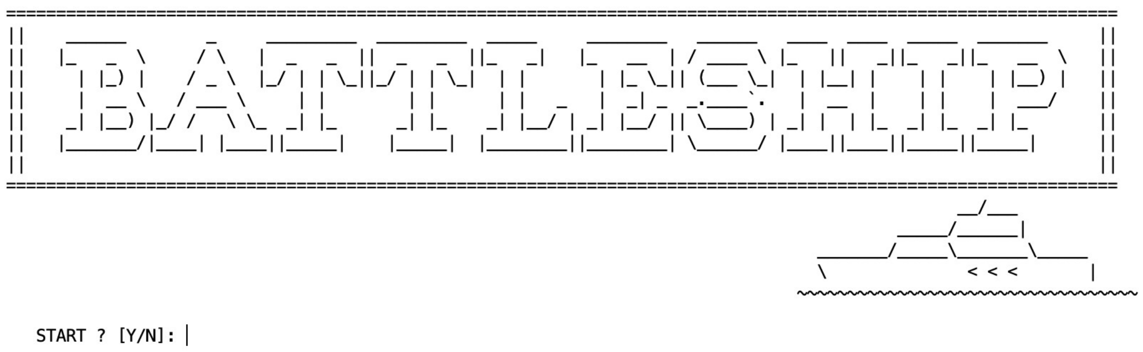
3. **Player Turn and Game Constants:**
   - `player_a_turn`: Value representing Player A's turn (value: 0).
   - `player_b_turn`: Value representing Player B's turn (value: 1).
   - `number_of_ships`: Total number of ships each player has (value: 6).
   - `number_of_cells`: Total number of cells on the 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.

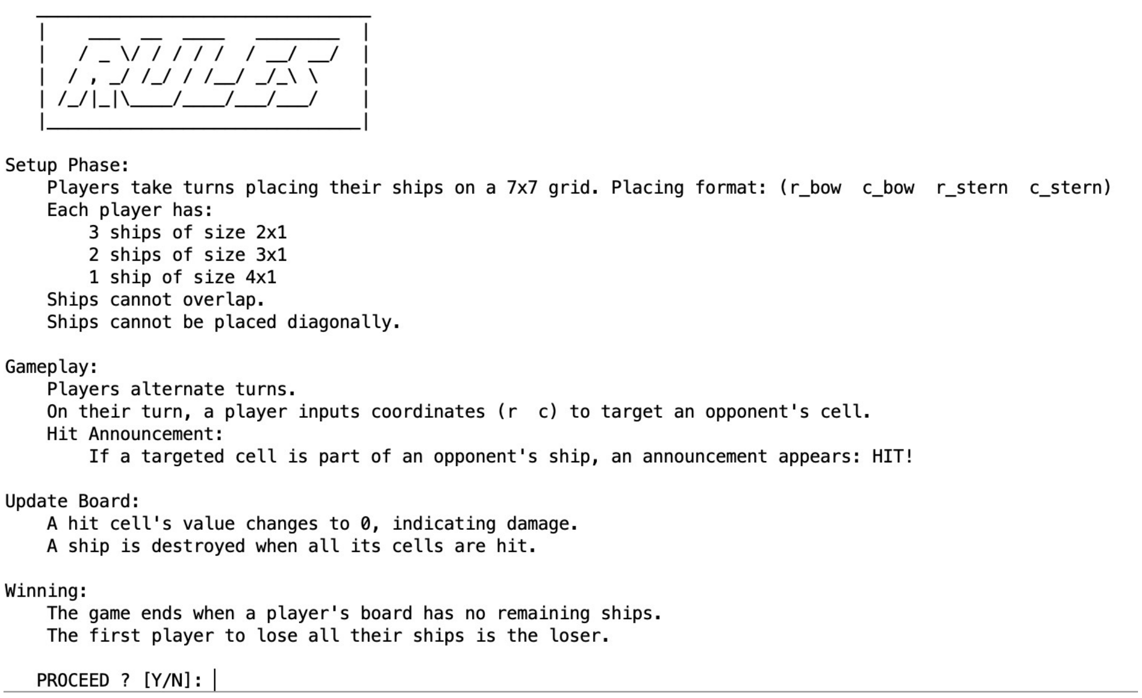


```
START ? [Y/N]:
```

Figure 2: 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.

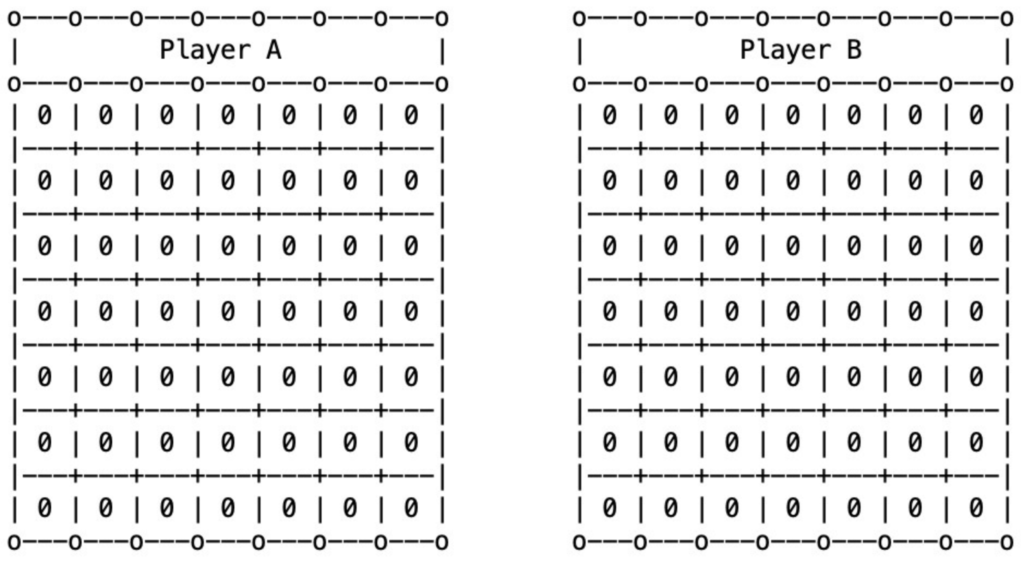


```
   ______________________________
  |    ___  __  ____   ________  |
  |   / _ \/ / / / /  / __/ __/  |
  |  / , _/ /_/ / /__/ _/_\ \    |
  | /_/|_|\____/____/___/___/    |
  |______________________________|

Setup Phase:
    Players take turns placing their ships on a 7x7 grid. Placing format: (r_bow  c_bow  r_stern  c_stern)
    Each player has:
        3 ships of size 2x1
        2 ships of size 3x1
        1 ship of size 4x1
    Ships cannot overlap.
    Ships cannot be placed diagonally.

Gameplay:
    Players alternate turns.
    On their turn, a player inputs coordinates (r  c) to target an opponent's cell.
    Hit Announcement:
        If a targeted cell is part of an opponent's ship, an announcement appears: HIT!

Update Board:
    A hit cell's value changes to 0, indicating damage.
    A ship is destroyed when all its cells are hit.

Winning:
    The game ends when a player's board has no remaining ships.
    The first player to lose all their ships is the loser.

   PROCEED ? [Y/N]: |
```

Figure 3: Rules Screen Interface

3. **Initial 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.



```
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
| 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 | 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
```

Figure 4: Initial Maps Interface

2.5 Game Logic

Analyze the game flow from the main function.

Discuss how players take turns placing ships and shooting, including input validation.

2.6 Helper Functions

Describe the purpose of each function $e.g., game_menu, rules_screen, read_ship, read_shot.$

Highlight the functionality of key functions and their interaction.

2.7 Input Validation - Error Handling

Discuss how the code validates user input for ship and shot coordinates.

Analyze the error messages and how they guide the user.

2.8 Gameplay Logic

Explain the sequence of events during ship placement and shooting.

Discuss how ships are drawn on the maps and the conditions for hitting or missing.

3 Conclusion

References

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