**

**VIETNAM NATIONAL UNIVERSITY**

**HOCHIMINH CITY UNIVERSITY OF TECHNOLOGY**

**FACULTY OF COMPUTER SCIENCE AND ENGINEERING**

**🙞···☼···🙜**

****

**REPORT**

**BATTLE SHIP**

………………………………………………………………………………………………………………………………………………………………………….

**Advisor:** Nguyen Thien An

|  |  |
| --- | --- |
| **Student** | **ID number** |
| Vo Truc Son | 2252720 |

*Ho Chi Minh city, December, 2023*

**Contents**

**1 Introduction 3**

**2 Implementation 4**

2.1 Overview 4

2.2 Data Structures 4

2.3 Pseudocodes 4,5

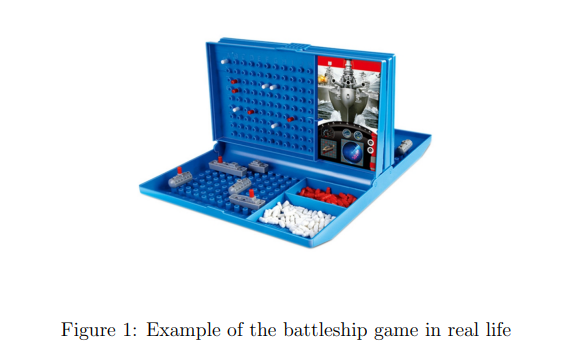
2.4 Codes 4.. 17

**3 Conclusion 18**

**4 References**18

**1 Introduction**

1. Battleship is a strategic board game between 2 players. It focuses on the process of planning deducting skill of both players. The game rules can be found in [1], in summary: 1. In a classic game, each player sets up a fleet of battleships on their map (a 10x10 grid). A fleet must contain a predefined set of battleships with different sizes. For example, a fleet of ships can consist of 5 2x1 ships, 3 3x1 ships, 1 5x1 ship.



1. 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.
2. If all of a player’s ships have been sunk, the game is over and their opponent wins

**2 Implementation**

2.1 Overview

Each player's board: Each player has a board of cells, usually divided into fixed-sized cells (e.g., 10x10 cells).

1. Place War: Each player places warriors on his board without letting the opponent know their location. The combatants come in different sizes and shapes, for example: boats, aircraft carriers, destroyers, storm ships, etc.
2. Turn: Players take turns choosing a cell on the opponent's board to attack. They announce the location of the cell they want to attack (e.g. "Column A, row 3"). The opponent will notify if they attack or miss (the shot doesn't hit).
3. Determining a war has been sunk: If the player attacks part of a war, the opposition will announce that the war has been sunk.
4. Win: The game continues until all of either player's warriors have been sunk. That player will be the winner

When developing this game in the MIPS board implementation, we need a way to place the ship, check the shoot and check whether there’s still valid ship

There needs to be a suitable data structure to store information about each player, the positions of the ships, and other information like played, last time, etc. We also need to handle user interaction so they can select cells to attack and can see the results.

Developing a Battleship game in the MIPS assembly would require the use of commands, control structures, and complex data structures.

2.2 Data Structures

- Data Structure type: no specific type, cause my idea is to store the input data in needed registers and do arithmetic and logic function.

- Purpose: cause input data is limit, so the way I solve problem with registers will give a better processing time

- Benefit: cause data is stored in separated register, it’s extremely easy to access data

- Complexity: in the worst case, the user input are wrong and too many loops are operated by the user. The complexity of the function is O(12n)

2.3 Pseudocodes

Input Player\_1\_name

Input Player\_2\_name

#*Input player 1 ship*

Input row/column of head/tail for a 4x1\_ship

Valid check and store data of 4x1\_ship

Input row/column of head/tail for 2 3x1\_ships

Valid check and store data of 3x1\_ships

Input row/column of head/tail for 3 2x1\_ships

Valid check and store data of 2x1\_ship

#*Input player 2 ship*

Input row/column of head/tail for a 4x1\_ship

Valid check and store data of 4x1\_ship

Input row/column of head/tail for 2 3x1\_ships

Valid check and store data of 3x1\_ships

Input row/column of head/tail for 3 2x1\_ships

Valid check and store data of 2x1\_ship

#*Game phase*

Input Player1 + Player2 loop

If input is correct, set data input to 49

Game over when all register of 1 player equal 49

2.4 Codes

.data

welcome: .asciiz "\*\*\*: Welcome to BATTLE SHIP! \n"

siri: .asciiz "\*\*\*: They call me Sirius, the supporter. \n"

again: .asciiz "Place your ship fail! \n "

again1: .asciiz "Invalid! \n"

tenP1: .asciiz "Sirius: Player 1, what can i call you (max 9 characters): \n"

p1: .asciiz "Player1: "

p2: .asciiz "Player2: "

sirius: .asciiz "Sirius: "

player1name: .space 10

player2name: .space 10

chaomung: .asciiz "Sirius: Nice to meet you, "

tenP2: .asciiz "Sirius: How about you, player 2. \n"

str: .asciiz "Player2: You can call me "

Intro: .asciiz "Now, let me tell you some information about BATTLE SHIP \n"

comma: .asciiz ", "

enter: .asciiz "\n"

Intro1: .asciiz "Both of you are competitors \n"

Intro2: .asciiz "You will have 6 ships and 49 blocks for each of you \n"

Intro3: .asciiz "Place the ship carefully, shoot each other and win the game \n"

Intro4: .asciiz "Have fun, both of you :> \n"

prepare: .asciiz "Now, let's prepare for the fight. \n"

prepare1: .asciiz "Place your 4x1 ship, "

prepare2: .asciiz "Place your first 3x1 ship, "

prepare3: .asciiz "Place your second 3x1 ship, "

prepare4: .asciiz "Place your first 2x1 ship, "

prepare5: .asciiz "Place your second 2x1 ship, "

prepare6: .asciiz "Place your third 2x1 ship, "

insert: .asciiz "Please insert row of head, collumn of head, row of tail and collumn of tail. \n"

begin: .asciiz "Now both of you are ready. For each turn, both of you get one shot. \n"

begin1: .asciiz "GOOD LUCK!!! \n"

fight: .asciiz "READYYYYYYYY!!!!!! SHOOTTTTTTTOOOOOO! \n"

fight1: .asciiz "Enter row and collumn of the square you want to shoot, "

miss: .asciiz "You missed. Opponent phase! \n"

ns: .asciiz "HIT !!! \n"

turn: .asciiz "Opponent phase \n "

victory: .asciiz "The victory belong to "

zero: .float 0.0

one: .float 1.0

two: .float 2.0

three: .float 3.0

six: .float 6.0

seven: .float 7.0

fn: .float 49.0

.text

lwc1 $f5, zero

lwc1 $f6, six

lwc1 $f8, three

lwc1 $f9, seven

lwc1 $f13, two

lwc1 $f14, one

# Get player name

li $v0, 4

la $a0, welcome

syscall

li $v0, 4

la $a0, siri

li $v0, 4

la $a0, tenP1

syscall

li $v0, 4

la $a0, p1

syscall

li $v0, 8

la $a0, player1name

li $a1, 10

syscall

li $v0, 4

la $a0, enter

syscall

li $v0, 4

la $a0, chaomung

syscall

li $v0, 4

la $a0, player1name

syscall

li $v0, 4

la $a0, tenP2

syscall

li $v0, 4

la $a0, str

syscall

li $v0, 8

la $a0, player2name

li $a1, 10

syscall

li $v0, 4

la $a0, enter

syscall

li $v0, 4

la $a0, chaomung

syscall

li $v0, 4

la $a0, player2name

syscall

#Introduce the game

li $v0, 4

la $a0, sirius

syscall

li $v0, 4

la $a0, Intro

syscall

li $v0, 4

la $a0, sirius

syscall

li $v0, 4

la $a0, Intro1

syscall

li $v0, 4

la $a0, sirius

syscall

li $v0, 4

la $a0, Intro2

syscall

li $v0, 4

la $a0, sirius

syscall

li $v0, 4

la $a0, Intro3

syscall

li $v0, 4

la $a0, sirius

syscall

li $v0, 4

la $a0, Intro4

syscall

li $v0, 4

la $a0, sirius

syscall

li $v0, 4

la $a0, prepare

syscall

j p1\_4x1

*\*\*\*Code represent for get ship data:*

#Get player 1 ship

# 4x1 ship

p1\_4x1:

li $v0, 4

la $a0, enter

syscall

li $v0, 4

la $a0, sirius

syscall

li $v0, 4

la $a0, prepare1

syscall

li $v0, 4

la $a0, player1name

syscall

li $v0, 4

la $a0, sirius

syscall

li $v0, 4

la $a0, insert

syscall

li $v0, 4

la $a0, p1

syscall

li $v0, 6

syscall

mov.s $f1, $f0

li $v0, 6

syscall

mov.s $f2, $f0

li $v0, 6

syscall

mov.s $f3, $f0

li $v0, 6

syscall

mov.s $f4, $f0

j check1\_4x1

check1\_4x1:

#check head row

c.lt.s $f1, $f5 # Compare if 0.0 < $f1

c.lt.s $f6, $f1 # Compare if $f1 < 6.0

bc1t wrong # Branch if $f1 is not within the range

c.lt.s $f2, $f5 # Compare if 0.0 < $f1

c.lt.s $f6, $f2 # Compare if $f1 < 6.0

bc1t wrong # Branch if $f1 is not within the range

c.lt.s $f3, $f5 # Compare if 0.0 < $f1

c.lt.s $f6, $f3 # Compare if $f1 < 6.0

bc1t wrong # Branch if $f1 is not within the range

c.lt.s $f4, $f5 # Compare if 0.0 < $f1

c.lt.s $f6, $f4 # Compare if $f1 < 6.0

bc1t wrong # Branch if $f1 is not within the range

j check2\_4x1

wrong:

li $v0, 4

la $a0, sirius

syscall

li $v0, 4

la $a0, again

syscall

j p1\_4x1

check2\_4x1:

c.eq.s $f1, $f3

bc1f check2\_2\_4x1

j check3\_1\_4x1

check2\_2\_4x1:

c.eq.s $f2, $f4

bc1f wrong

j check3\_2\_4x1

check3\_1\_4x1:

sub.s $f7, $f2, $f4

abs.s $f7, $f7

c.eq.s $f7, $f8

bc1f wrong

j convert4x1\_1

check3\_2\_4x1:

sub.s $f7, $f1, $f3

abs.s $f7, $f7

c.eq.s $f7, $f8

bc1f wrong

j convert4x1\_2

convert4x1\_1:

mul.s $f10, $f1, $f9

add.s $f10, $f10, $f2

mul.s $f11, $f1, $f9

add.s $f11, $f11, $f4

c.le.s $f10, $f11

bc1f convert4x1a

j convert4x1b

convert4x1a:

mov.s $f31, $f10

mov.s $f30, $f11

j p1\_3x1\_1

convert4x1b:

mov.s $f31, $f11

mov.s $f30, $f10

j p1\_3x1\_1

convert4x1\_2:

mul.s $f10, $f1, $f9

add.s $f10, $f10, $f2

mul.s $f11, $f3, $f9

add.s $f11, $f11, $f4

c.le.s $f10, $f11

bc1f convert4x1c

j convert4x1d

convert4x1c:

mov.s $f31, $f10

mov.s $f30, $f11

j p1\_3x1\_1

convert4x1d:

mov.s $f31, $f11

mov.s $f30, $f10

j p1\_3x1\_1

*\*\*\*Code represent for game phase*

main:

li $v0, 4

la $a0, enter

syscall

li $v0,4

la $a0, sirius

syscall

li $v0, 4

la $a0, begin

syscall

li $v0,4

la $a0, sirius

syscall

li $v0, 4

la $a0, begin1

syscall

j main1

main1:

lwc1 $f0, fn

li $v0, 4

la $a0, enter

syscall

li $v0, 4

la $a0, sirius

syscall

li $v0, 4

la $a0, fight

syscall

li $v0, 4

la $a0, sirius

syscall

li $v0, 4

la $a0, fight1

syscall

li $v0, 4

la $a0, player1name

syscall

li $v0, 4

la $a0, p1

syscall

li $v0, 6

syscall

mov.s $f1, $f0

li $v0, 6

syscall

mov.s $f2, $f0

j checkp1

checkp1:

lwc1 $f5, zero

lwc1 $f6, six

#check head row

c.lt.s $f1, $f5 # Compare if 0.0 < $f1

c.lt.s $f6, $f1 # Compare if $f1 < 6.0

bc1t wrong12 # Branch if $f1 is not within the range

c.lt.s $f2, $f5 # Compare if 0.0 < $f1

c.lt.s $f6, $f2 # Compare if $f1 < 6.0

bc1t wrong12 # Branch if $f1 is not within the range

j cvt1

wrong12:

li $v0, 4

la $a0, sirius

syscall

li $v0, 4

la $a0, again1

syscall

j main1

cvt1:

lwc1 $f0, seven

mul.s $f1, $f1, $f0

add.s $f1, $f1, $f2

j check1\_1

check1\_1:

c.eq.s $f1, $f31

bc1f c

j hit

c:

c.eq.s $f1, $f30

bc1f c1

j hit1

c1:

c.eq.s $f1, $f29

bc1f c2

j hit2

c2:

c.eq.s $f1, $f28

bc1f c3

j hit3

c3:

c.eq.s $f1, $f27

bc1f c4

j hit4

c4:

c.eq.s $f1, $f26

bc1f c5

j hit5

c5:

c.eq.s $f1, $f25

bc1f c6

j hit6

c6:

c.eq.s $f1, $f24

bc1f c7

j hit7

c7:

c.eq.s $f1, $f23

bc1f c8

j hit8

c8:

c.eq.s $f1, $f22

bc1f c9

j hit9

c9:

c.eq.s $f1, $f21

bc1f c10

j hit10

c10:

c.eq.s $f1, $f20

bc1f gg

j hit11

gg:

c.eq.s $f20, $f0

bc1f miss1

j totalcheck1

miss1:

li $v0, 4

la $a0, sirius

syscall

li $v0, 4

la $a0, miss

syscall

j totalcheck1

hit:

li $v0, 4

la $a0, ns

syscall

lwc1 $f31, fn

j main2

hit1:

li $v0, 4

la $a0, ns

syscall

lwc1 $f30, fn

j main2

hit2:

li $v0, 4

la $a0, ns

syscall

lwc1 $f29, fn

j main2

hit3:

li $v0, 4

la $a0, ns

syscall

lwc1 $f28, fn

j main2

hit4:

li $v0, 4

la $a0, ns

syscall

lwc1 $f27, fn

j main2

hit5:

li $v0, 4

la $a0, ns

syscall

lwc1 $f26, fn

j main2

hit6:

li $v0, 4

la $a0, ns

syscall

lwc1 $f25, fn

j main2

hit7:

li $v0, 4

la $a0, ns

syscall

lwc1 $f24, fn

j main2

hit8:

li $v0, 4

la $a0, ns

syscall

lwc1 $f23, fn

j main2

hit9:

li $v0, 4

la $a0, ns

syscall

lwc1 $f22, fn

j main2

hit10:

li $v0, 4

la $a0, ns

syscall

lwc1 $f21, fn

j main2

hit11:

li $v0, 4

la $a0, ns

syscall

lwc1 $f20, fn

j main2

totalcheck1:

c.eq.s $f31, $f30

bc1f main2

j e

e:

c.eq.s $f30, $f29

bc1f main2

j e1

e1:

c.eq.s $f28, $f29

bc1f main2

j e2

e2:

c.eq.s $f27, $f28

bc1f main2

j e3

e3:

c.eq.s $f26, $f27

bc1f main2

j e4

e4:

c.eq.s $f25, $f26

bc1f main2

j e5

e5:

c.eq.s $f24, $f25

bc1f main2

j e6

e6:

c.eq.s $f23, $f24

bc1f main2

j e7

e7:

c.eq.s $f22, $f23

bc1f main2

j e8

e8:

c.eq.s $f21, $f22

bc1f main2

j e9

e9:

c.eq.s $f20, $f21

bc1f main2

j end1

end1:

li $v0, 4

la $a0, sirius

syscall

li $v0, 4

la $a0, victory

syscall

li $v0, 4

la $a0, player1name

syscall

li $v0, 10

syscall

**3 Conclusion**

From a programmer's perspective, the game Battleship provides an interesting problem in logic implementation, data structure, and user interaction. Here is a conclusion from a programming perspective:

1. Logic and Algorithm: Battleship game requires a clear logic and effective algorithm to determine, manage and check the positions of battleships. Processing turns, checking shots, and determining sunk ships require an accurate and efficient algorithm.

2. Data Structure: Implementing a Battleship game requires the use of appropriate data structures to store each player's board, the positions of the battleships, and the game state. Using arrays, linked lists, or other data structures can make information management easier.

3. User interaction: The important part of the game is interaction with the player. The design of the user interface or the way users interact with the game needs to be carefully considered to make the gaming experience fun and easy to understand.

4. Testing and debugging: During game development, testing and debugging are important. It is necessary to examine test cases, identify and fix errors to ensure the game works as expected.

5. Development and Expansion: Battleship can be developed and expanded to include new features, such as multiplayer, artificial intelligence, or improved user interface . This requires flexibility in managing source code and data structures.

Overall, the Battleship game is not only an entertaining game but also an interesting challenge for programmers to implement logic, data structures and user interactions effectively.

**4 References**

[1] Wikipedia, “Battleship (game).” [Online]. Available: <https://en.wikipedia.org/wiki/Battleship> (game)