

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



COMPUTER ARCHITECTURE (C02007)

Assignment

BATTLESHIP

Advisor: Assoc. Prof. Pham Quoc Cuong
Student: Vo Thanh Tam - Student ID: 2213046

HO CHI MINH CITY, NOVEMBER 2023



Contents

1	Introduction	2
1.1	About the project	2
1.2	Objectives	2
2	Development process	3
2.1	Start and Input	3
2.2	Game mechanics	8
2.3	Storing player's input:	11
3	Results and in-game captures	12
4	Conclusion	19
5	References	19

1 Introduction

1.1 About the project

This project comes along with Computer Architecture (CO2007) assignment. It is a MIPS program presents childhood 2-player game "Battleship" with a 7×7 map and 3 types of ships to place on:

- 1 ship at 4×1 size.
- 2 ships at 3×1 size.
- 3 ships at 2×1 size.

Each player will be given a map to secretly place their ships on. After that, the game begins:

- Each player take turn and try to destroy all the opponent's ships by guessing their place with a coordinate on the map, one square at once, called "target square".
- If there is a opponent's ship at the coordinate given by the player, then the opponent will announce "HIT!", else "MISS!".
- Player who destroy all opponent's ships first is the winner.



Figure 1: The Battleship game in real life

1.2 Objectives

In this submission, my goal is delivering a functional Battleship game on MIPS with text-based interface:

- Two players mode.
- Receive ship initialize input from each player.

- Start the game with receiving target from player's input like the original one.

All of this will be made on MARS [1] - a MIPS simulator and its instruction [2].

But in this report, I will show my idea and development of each part of the game with no code involved.

2 Development process

2.1 Start and Input

When open the game, a Start screen appears. A map for each player will be prepared, with some new things are introduced as Instruction because this is made on MIPS:

- As the map is a 7×7 2D array in real life, but since MIPS doesn't support 2D array directly, a 1D array with 49 elements will be used to present the map. In this report we will mainly see the map at 2D array for easier understanding.
- In the beginning, all elements is set to 0 to show this is a blank map.

Here is the initial map, presented in 7×7 2D array but bear in mind that the map we use in the program is actually a 49-element 1D array:

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0

Table 1: Initial map for each player

Then the program will ask each player to place their ships:

- 1 ship at 4×1 size.
- 2 ships at 3×1 size.
- 3 ships at 2×1 size.

The program then require each player to enter all ships placement, first is Player 1 and then Player 2.

Input rule:

- **Input format:** The player's input must follow exactly the format:

$$row_{bow} \text{ [space] } column_{bow} \text{ [space] } row_{stern} \text{ [space] } column_{stern}$$

in which:

- row_{bow} , $column_{bow}$, row_{stern} , $column_{stern}$ are the coordinate of the ship's bow and stern respectively.

- After type one number, the next number is separated with the current one by a space, there is exact one space between two numbers, no space at the begin or in the end.

For example: if the bow of the ship lies on (0,0) and the stern lies on (3,0) in the map (or the grid for easier to follow), then the input would be: 0 0 3 0.

• **Input rule and checking:**

- First, all four numbers represent the coordinates of bow and stern of the ship must be from 0 to 6, as the size of the map is 7×7 so the row and column are labeled from 0 to 6.
- After that, the bow and stern of a ship must be on the same row or column, which means there must be $row_{bow} = row_{stern}$ or $column_{bow} = column_{stern}$. This can be written as:

$$\begin{cases} row_{bow} - row_{stern} = 0 \\ column_{bow} - column_{stern} = 0 \end{cases}$$

- When the above criteria is met, the length of the input ship must equal to the required length l , as the bow and stern must on the same row or column, the criteria will be valid if the following statement is true:

$$|(row_{bow} - row_{stern}) + (column_{bow} - column_{stern})| = l - 1 \quad (1)$$

Because the ship could be place

- * Rightward: $(row_{bow} < row_{stern})$
- * Leftward: $(row_{bow} > row_{stern})$
- * Upward: $(column_{bow} < column_{stern})$
- * Downward: $(column_{bow} > column_{stern})$

we must take the absolute value of the left-hand side of (1) to compare with $l - 1$, which is a positive value.

The right-hand side of (1) is $l - 1$ can be explained by below example:

1	1	1	1
---	---	---	---

Table 2: Ship with length 4

Let say this ship is placed leftward with the red cell is the bow, and blue cell is the stern, and if the bow is (3,2) then the stern is (3,5), so $5 - 2 = 3 = 4 - 1 = l - 1$.

- Overlapping - placing ship on an area that has been occupied by previous ship, is not allowed.
- If the input doesn't meet any criteria above, player must type in the new input until it is valid.

After finished length checking - that means the input has passed format and position checking, use l as the required length, starting mark the place of the ship in player's map by turning 0 into 1 in corresponding area. Let see the map is a 7×7 2D array, here are the steps:

1. Determine if the ship is placed vertically ($column_{bow} = column_{stern}$) or horizontally ($row_{bow} = row_{stern}$).

2. If the ship is vertically placed:

- (a) In a 1D array with $m.n$ elements, the distance between two elements $a(x_1, y_1), b(x_2, y_2)$ that are seem to be on the same column in 2D array $m \times n$ is always $k.n$, with $k = |x_1 - x_2|$.
For example:

	0	1	2	3	4	5	6
0	0	x_0	0	0	0	0	0
1	0	x_1	0	0	0	0	0
2	0	x_2	0	0	0	0	0
3	0	x_3	0	0	0	0	0
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0

Table 3: Same column

Let assume the above map is an array with name "M". In term of 2D array, there are elements x_0, x_1, x_2, x_3 on the same column 1, their locations are $M[0][1], M[1][1], M[2][1], M[3][1]$ respectively. When see the map as 1D array like what its truly is, with the size of the 2D array is 7×7 , then $n = 7$, their locations are: $M[1], M[8], M[15], M[22]$, in that order.

- (b) Check if it is placed upward ($row_{bow} < row_{stern}$) or downward ($row_{bow} > row_{stern}$).
If upward, store the address of the bow and set it to current address and store the column the ship is placed on, then do a loop:
- Change the value of current address (current position in the map) from 0 to 1.
 - Update the current address to the next position. To do so, increment the current address - for example x_0 at $M[1]$ by n , in this case is 7 so $1 + 7 = 8 \Rightarrow M[8] : x_1$.
 - Keep doing the loop for l times. After stop iterating, the ship is placed correctly on the map.
- (c) If the ship is placed downward, doing the same steps as above but instead of increment the current address by n , we subtract it by n , in this case is 7.

3. If the ship is horizontally placed:

- (a) In a 1D array with $m.n$ elements, the distance between two elements $a(x_1, y_1), b(x_2, y_2)$ that are seem to be on the same row in 2D array $m \times n$ is always k , with $k = |y_1 - y_2|$.
For example:

	0	1	2	3	4	5	6
0	0	x_0	x_1	x_2	0	0	0
1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0

Table 4: Same row

Let assume the above map is an array with name "M". In term of 2D array, there are elements x_0, x_1, x_2 on the same row 0, their locations are $M[0][1], M[0][2], M[0][3]$ respectively. When see the map as 1D array like what its truly is their locations are: $M[1], M[2], M[3]$ in that order.

- (b) Check if it is placed leftward ($column_{bow} < column_{stern}$) or rightward ($column_{bow} > column_{stern}$).

If leftward, store the address of the bow and set it to current address, then do a loop:

- Change the value of current address (current position on the map) from 0 to 1.
- Update the current address to the next position. To do so, simply increment the current address by 1. For example: from Table 4, take x_0 : $M[1]$ as current address then the next address will be $1 + 1 = 2 \Rightarrow M[2] : x_1$.
- Keep doing the loop for l times. After stop iterating, the ship is placed correctly on the map.

- (c) If the ship is placed rightward, doing the same steps as above but instead of increment the current address by 1, we subtract it by 1.

A valid input example: Player type in "0 0 3 0" with the required length is 4 for 4×1 ship, then the player's map will be like this:

	0	1	2	3	4	5	6
0	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0
2	1	0	0	0	0	0	0
3	1	0	0	0	0	0	0
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0

Table 5: Valid input

But if in the process replacing 0 to 1 in the new ship area, the value of one cell isn't 0, means there is already a ship there, this leads to overlapping problem. For example: using the example above, add new ship with length 3 (ship 3×1) with player input "2 2 2 0":

	0	1	2	3	4	5	6
0	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0
2	1	1	1	0	0	0	0
3	1	0	0	0	0	0	0
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0

Table 6: Overlapping

The red square shows that the new 3×1 ship is overlap the old 4×1 ship which was placed before, therefore the player need to enter a new place for the ship as it is invalid input. But before allowing player enter new ship, we need to undo the replacing 0 with 1 process because the ship is now invalid, means the 1 we marked before has no meaning and thus must be turned back to 0 for later ship placing.

To do so, as we detect the overlapping problem in the red cell in above table, and its value is 1 which belongs to the previous valid ship, we start tracing back the road:

- If the ship is placed horizontally:
 - If the ship is placed leftward:

As the initialization take from left to right, increment by 1 each step. In the removing, we go to the left from the overlap position: store its address and subtract by 1 (to move to the previous position), then:

 - * Change the value of current address (current position on the map) from 1 to 0.
 - * Update current address to go backward, in this case is subtract itself by 1.
 - * Keep doing the loops for k times, which k is the number of initializing loop has been done until overlapping detected.
 - If the ship is placed rightward, do exact above steps but now the current address is incremented by 1 after a step as the undo runs from left to right now.
- If the ship is placed vertically:
 - If the ship is place upward:

As the initialization take downward, increment by n (the amount of column in 2D map) each step. In the removing, we go upward from the overlap position: store its address and subtract by n (to move to the previous position), then.

 - * Change the value of current address (current position on the map) from 1 to 0.
 - * Update current address to go upward, in this case is subtract itself by n .
 - * Keep doing the loops for k times, which k is the number of initializing loop has been done until overlapping detected.
 - If the ship is placed downward, do exact above steps but now the current address is incremented by n after a step as the undo runs upward now.

Since overlapping is invalid input, player need to enter a new place for the ship until it is valid. After a player finished placing ships, the program will ask if they satisfy with their current map, or want to reset the map to place ships in other places, or just in case they get bored, there will be an "EXIT" option appears so they can exit the game.

To sum up, all the input process for each player is described below:

Player start with placing 1 4×1 ship, then 2 3×1 ships and finally 3 2×1 ships, in that order.

Step 1: Player type in their choice for ship's position.

Step 2: Check if the input is valid, if not player need to type in a new input until it is valid.

Step 3: Start marking/placing the ship on player's map, the program will show player their current map so they can track them easily.

Step 4: Check if player has placed enough the amount of ship of required type, if not goes to Step 1, else stop and begin placing next type of ship.

Step 5: If player has placed all the ships, a complete map shows up and ask if player feel good with their choice or not.

Here is an example of two valid maps after players have placed all require ships:

• Player 1:

- 4×1 ship: "0 0 3 0"
- 3×1 ships: "6 0 6 2", "5 3 5 5"
- 2×1 ships: "1 4 2 4", "2 1 3 1", "4 4 4 5"

• Player 2:

- 4×1 ship: "6 1 6 4"
- 3×1 ships: "4 1 4 3", "3 3 1 3"
- 2×1 ships: "0 0 0 1", "0 5 1 5", "4 0 5 0"

	0	1	2	3	4	5	6
0	1	0	0	0	0	0	0
1	1	0	0	0	1	0	0
2	1	1	0	0	1	0	0
3	1	1	0	0	0	0	0
4	0	0	0	0	1	1	0
5	0	0	0	1	1	1	0
6	1	1	1	0	0	0	0

Table 7: Player 1 map

	0	1	2	3	4	5	6
0	1	1	0	0	0	1	0
1	0	0	0	1	0	1	0
2	0	0	0	1	0	0	0
3	0	0	0	1	0	0	0
4	1	1	1	1	0	0	0
5	1	0	0	0	0	0	0
6	0	1	1	1	1	0	0

Table 8: Player 2 map

2.2 Game mechanics

The rule of the game is simple and got introduced at [here](#). In this part we dig into how does this work in the program:

1. There are two counters c_1 and c_2 for tracking how many times each player "hit" opponent's ships. As each player has 1 4×1 ship, 2 3×1 ships, 3 2×1 ships, the sum of all 1 in a player's map is: $4.1 + 2.3.1 + 3.2.1 = 16$, means there are 16 parts of ships to be hit. Therefore if a counter reaches 16, then that Player win. For example: $c_1 = 16$, then Player 1 win.

2. Each player can see a map that is made from opponent's one, but it is hidden with all value is $*$ - basically it is the copy of opponent's map but with a layer of all $*$ cover on top, and will be revealed (modify) after each turn, call $C_{i=\overline{1,2}}$. When it is Player i turn, it will show map C_j , with $i \neq j; i, j = \overline{1,2}$. And as this is used in MIPS, $C_{i=\overline{1,2}}$ must be 1D array but for convenient, we will see it as 2D array in this report, so its size is 7×7 .

3. Input:

- Player enter a string input with format: row_{target} [space] $column_{target}$:
 - row_{target} and $column_{target}$ must be a number value from 0 to 6 represent the row and column index of the target position.
 - There is **only one** [space] between them and no other spaces appear in the string input.
- If the input is invalid, program requires player to enter the coordinate again until it is valid.

4. Process:

- As we all work in 1D array, then keep assuming the player's map is M , and the position player typed in is $M[x][y]$ in 2D 7×7 array. To access that position - let call it p in 1D array, $p = 7.x + y$.
- If the value of $M[p] = 1$, announce player "HIT!" to tell them there is a ship's part in that position, and change $M[p]$'s value to 0, their counter is incremented by 1, so if player accidentally enter the same position, the counter won't be incremented.
For example: It's Player 2 turn and they type "0 0" as their target, then use the Table 7 above for Player 1's map:

	0	1	2	3	4	5	6
0	1	0	0	0	0	0	0
1	1	0	0	0	1	0	0
2	1	1	0	0	1	0	0
3	1	1	0	0	0	0	0
4	0	0	0	0	1	1	0
5	0	0	0	1	1	1	0
6	1	1	1	0	0	0	0

Table 9: HIT

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	1	0	0	0	1	0	0
2	1	1	0	0	1	0	0
3	1	1	0	0	0	0	0
4	0	0	0	0	1	1	0
5	0	0	0	1	1	1	0
6	1	1	1	0	0	0	0

Table 10: Changing value

- If the value of $M[p] = 0$, announce player "MISS!" to tell them there isn't any ship's part or there was a ship's part there but they "hit" it before, the value and counter stays remain.
For example: It's Player 2 turn and they type "3 3" as their target, then use the Table 7 above for Player 1's map:
- The map C_j corresponding with current player will be modified (revealed) at $C_j[p]$ if $C_j[p] \neq 0$:

$$C_j[p] = \begin{cases} X & \text{if there is a ship's part.} \\ \sim & \text{if there isn't anything.} \end{cases}$$

For example, using two examples above with Player 2 choose "0 0" and "3 3" as their targets, the modification in C_1 will be like this:

	0	1	2	3	4	5	6
0	1	0	0	0	0	0	0
1	1	0	0	0	1	0	0
2	1	1	0	0	1	0	0
3	1	1	0	0	0	0	0
4	0	0	0	0	1	1	0
5	0	0	0	1	1	1	0
6	1	1	1	0	0	0	0

Table 11: MISS

- With "0 0" first, value of $C_1[0][0]$ will be changed to X because it hasn't been revealed yet and there is a ship's part.

	0	1	2	3	4	5	6
0	*	*	*	*	*	*	*
1	*	*	*	*	*	*	*
2	*	*	*	*	*	*	*
3	*	*	*	*	*	*	*
4	*	*	*	*	*	*	*
5	*	*	*	*	*	*	*
6	*	*	*	*	*	*	*

Table 12: Before

	0	1	2	3	4	5	6
0	X	*	*	*	*	*	*
1	*	*	*	*	*	*	*
2	*	*	*	*	*	*	*
3	*	*	*	*	*	*	*
4	*	*	*	*	*	*	*
5	*	*	*	*	*	*	*
6	*	*	*	*	*	*	*

Table 13: After

- Then with "3 3", value of $C_1[3][3]$ will be changed to \sim because it hasn't been revealed yet and there isn't anything.

	0	1	2	3	4	5	6
0	X	*	*	*	*	*	*
1	*	*	*	*	*	*	*
2	*	*	*	*	*	*	*
3	*	*	*	*	*	*	*
4	*	*	*	*	*	*	*
5	*	*	*	*	*	*	*
6	*	*	*	*	*	*	*

Table 14: Before

	0	1	2	3	4	5	6
0	X	*	*	*	*	*	*
1	*	*	*	*	*	*	*
2	*	*	*	*	*	*	*
3	*	*	*	\sim	*	*	*
4	*	*	*	*	*	*	*
5	*	*	*	*	*	*	*
6	*	*	*	*	*	*	*

Table 15: After

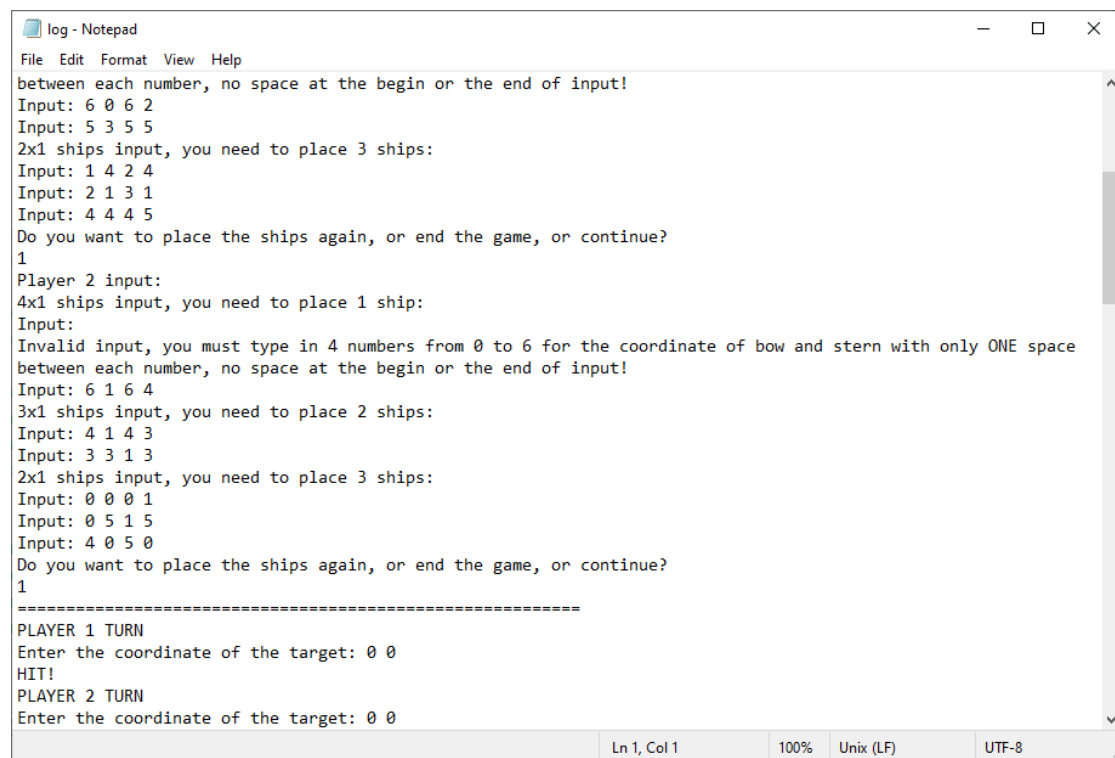
- And if Player choose "0 0" or "3 3" again, the value of $C_1[0][0]$ or $C_1[3][3]$ won't be changed as they are already revealed.

- Repeat from step 2 until a counter reaches 16, announce the winner.
- Ask players if they want to play again, or exit the game. If play again, two maps will be reset to all $*$ and start over again.

2.3 Storing player's input:

All of two players moves (inputs) are stored in a log file for further checking or reviewing purpose. Doing that is pretty easy because all inputs in the program so far is entered as string, therefore we just need to write string into the log file. MIPS supports write string to file by giving it the string and the length of it, and we just need to count the length of string by keep incrementing the counter until it meets `\0` or register `$zero` - the end of the string.

Here is how it looks like:



```
log - Notepad
File Edit Format View Help
between each number, no space at the begin or the end of input!
Input: 6 0 6 2
Input: 5 3 5 5
2x1 ships input, you need to place 3 ships:
Input: 1 4 2 4
Input: 2 1 3 1
Input: 4 4 4 5
Do you want to place the ships again, or end the game, or continue?
1
Player 2 input:
4x1 ships input, you need to place 1 ship:
Input:
Invalid input, you must type in 4 numbers from 0 to 6 for the coordinate of bow and stern with only ONE space
between each number, no space at the begin or the end of input!
Input: 6 1 6 4
3x1 ships input, you need to place 2 ships:
Input: 4 1 4 3
Input: 3 3 1 3
2x1 ships input, you need to place 3 ships:
Input: 0 0 0 1
Input: 0 5 1 5
Input: 4 0 5 0
Do you want to place the ships again, or end the game, or continue?
1
=====
PLAYER 1 TURN
Enter the coordinate of the target: 0 0
HIT!
PLAYER 2 TURN
Enter the coordinate of the target: 0 0
Ln 1, Col 1    100%    Unix (LF)    UTF-8
```

Figure 2: Log file

3 Results and in-game captures

Here are captures from real program writing in MIPS:

- **START:**

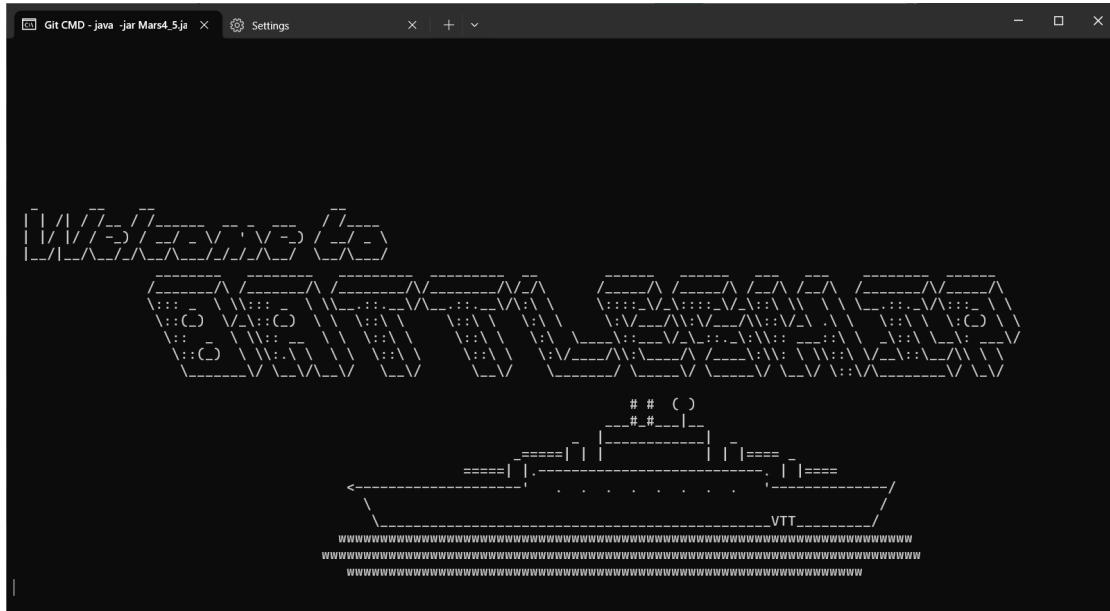


Figure 3: Start screen

- **INSTRUCTION:**

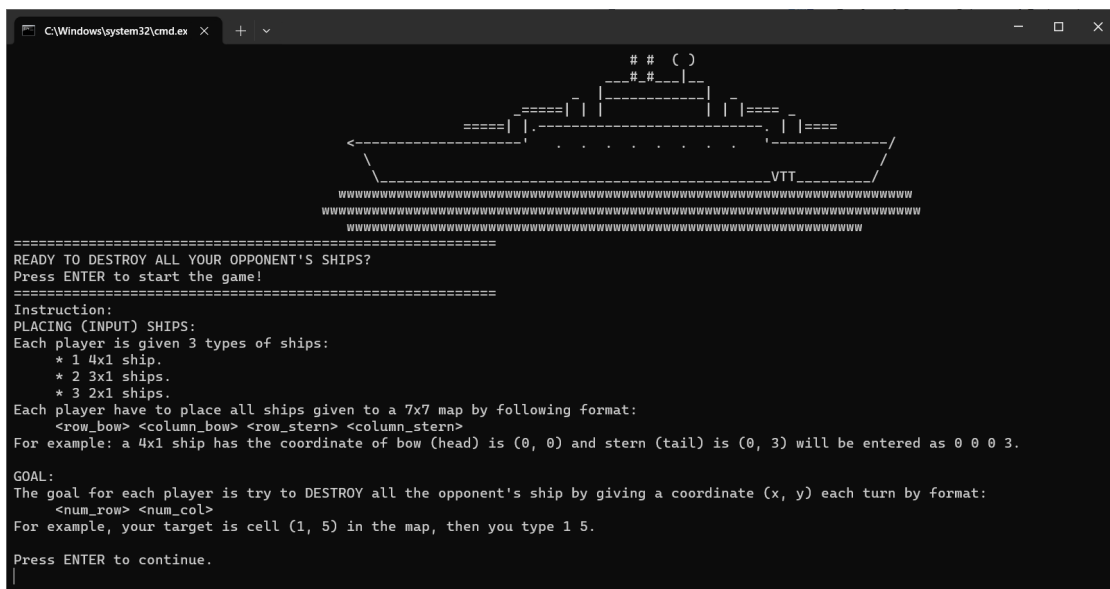


Figure 4: Instruction

- **INPUT:**

```
Git CMD - java -jar Mars4.5ja × + ~  
Press ENTER to continue.  
  
=====
```

```
[ _\N/_ ] [ _\D/_ ] [ _\I/_ ]  
[ _\N/_ ] [ _\D/_ ] [ _\I/_ ]  
[ _\N/_ ] [ _\D/_ ] [ _\I/_ ]  
=====
```

```
Player 1 input:  
Current map:
```

r\c	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0

```
4x1 ships input, you need to place 1 ship:  
Ship 1  
|
```

Figure 5: Player 1 input

- After player entered a valid input (valid ship), their map will be updated:

```

Git CMD
x + v
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|
4x1 ships input, you need to place 1 ship:
Ship 1
0 0 0 3
Current map:
r\c |---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|
| 2 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|
| 3 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|
| 4 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|
| 5 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|
| 6 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|

```

Figure 6: Player 1 input

- Invalid input:

* Wrong format:

```
4x1 ships input, you need to place 1 ship:
Ship 1
abcd
Invalid input, you must type in 4 numbers from 0 to 6 for the coordinate of bow and stern with only ONE space between each number,
no space at the begin or the end of input!
```

Figure 7: A wrong format example

* Bow and stern are not on the same row or column:

```
Ship 1
1 2 3 4
Invalid input, bow and stern must be on the same row or same column!
```

Figure 8: Wrong position

* The length of input isn't equal to the required length:

```
Ship 1
0 0 0 2
Invalid input, required length is 4
```

Figure 9: Wrong length

* Overlap:

```
Current map:
|---|---|---|---|---|---|---|
|r\c| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|
3x1 ships input, you need to place 2 ships:
Ship 1
0 0 2 0
There is already a ship here, please place elsewhere!
```

Figure 10: Overlapping

- After a player has placed every ships, program will show the complete map and ask if they already satisfy with their choice or not, or if they get bored they can exit the game right now.

Here is the complete map of Player 1 based on Table 7:

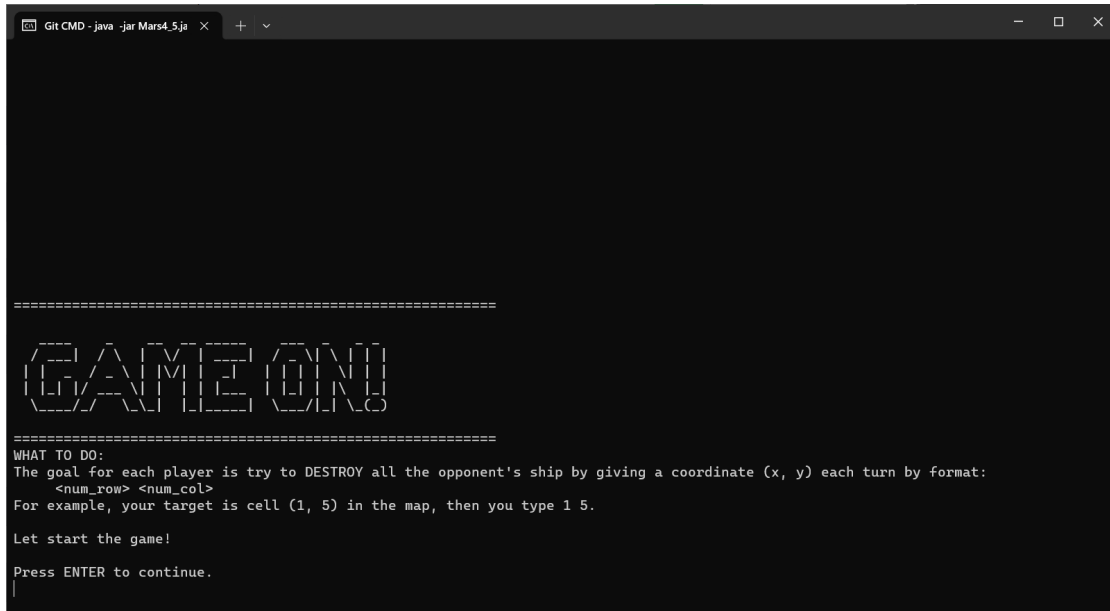
```
Git CMD - java -jar Mars4_5.jar x + v

=====
Map of Player 1:
r\c 0 1 2 3 4 5 6
0 1 1 1 1 0 0 0
1 0 0 0 0 1 0 0
2 0 1 0 0 1 0 0
3 0 1 0 0 0 0 0
4 0 0 0 0 1 1 0
5 0 0 0 1 1 1 0
6 1 1 1 0 0 0 0
=====
Do you want to place the ships again, or end the game, or continue?
Plese type 1 to continue, 2 to place the ship again and 3 for exit the game.
```

Figure 11: Complete P1 map and choices

- **THE GAME BEGINS:**

- The instruction (reminder) appears to help players know what to do:



```
Git CMD - java -jar Mars4_5.jar x + v

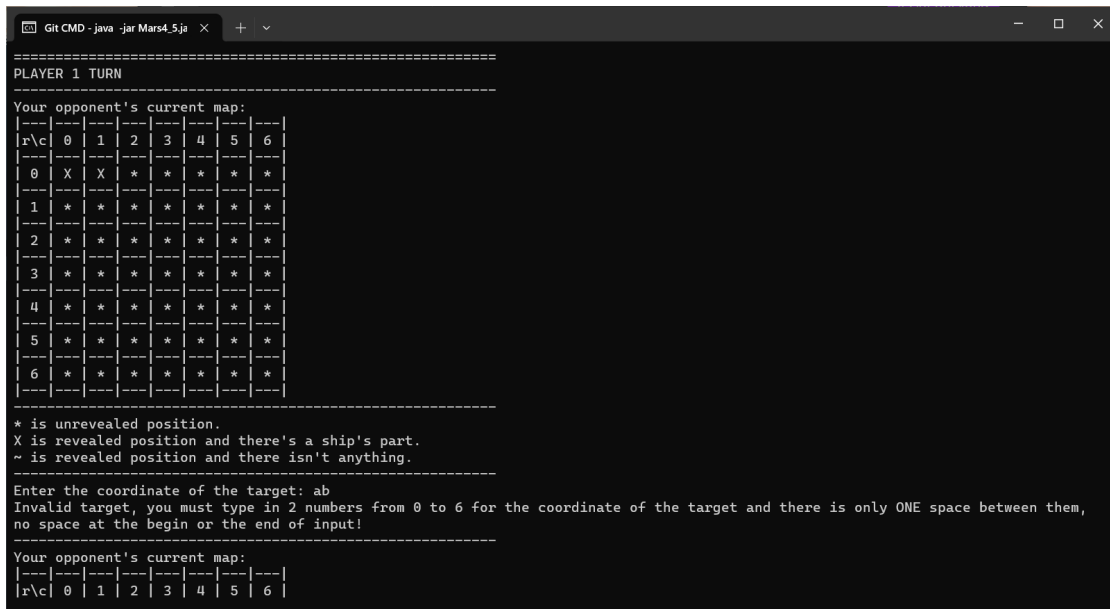
=====
  GAME ON!
=====
WHAT TO DO:
The goal for each player is try to DESTROY all the opponent's ship by giving a coordinate (x, y) each turn by format:
  <num_row> <num_col>
For example, your target is cell (1, 5) in the map, then you type 1 5.

Let start the game!

Press ENTER to continue.
|
```

Figure 12: Attack instruction

- Invalid target:



```
Git CMD - java -jar Mars4_5.jar x + v

=====
PLAYER 1 TURN
=====
Your opponent's current map:
|---|---|---|---|---|---|---|
|r\c| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 0 | X | X | * | * | * | * |
| 1 | * | * | * | * | * | * |
| 2 | * | * | * | * | * | * |
| 3 | * | * | * | * | * | * |
| 4 | * | * | * | * | * | * |
| 5 | * | * | * | * | * | * |
| 6 | * | * | * | * | * | * |
|---|---|---|---|---|---|---|

* is unrevealed position.
X is revealed position and there's a ship's part.
~ is revealed position and there isn't anything.

Enter the coordinate of the target: ab
Invalid target, you must type in 2 numbers from 0 to 6 for the coordinate of the target and there is only ONE space between them,
no space at the begin or the end of input!

Your opponent's current map:
|---|---|---|---|---|---|---|
|r\c| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
```

Figure 13: Wrong target (format)

- If the input is valid, the program will show if player "HIT" opponent's ship at that position:

```

Git CMD - java -jar Mars4_5ja  X + v
PLAYER 1 TURN
-----
Your opponent's current map:
-----


|     |   |   |   |   |   |   |   |
|-----|---|---|---|---|---|---|---|
| r\c | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 0   | X | * | * | * | * | * | * |
| 1   | * | * | * | * | * | * | * |
| 2   | * | * | * | * | * | * | * |
| 3   | * | * | * | * | * | * | * |
| 4   | * | * | * | * | * | * | * |
| 5   | * | * | * | * | * | * | * |
| 6   | * | * | * | * | * | * | * |


-----
* is unrevealed position.
X is revealed position and there's a ship's part.
~ is revealed position and there isn't anything.
-----
Enter the coordinate of the target: 0 1

```

Figure 14: HIT!

- Else, it shows "MISS":

[illegible]

Figure 15: MISS!

- Once a player has destroyed all their opponent's ship, the game announces the winner. For example, this is when Player 1 is the winner:

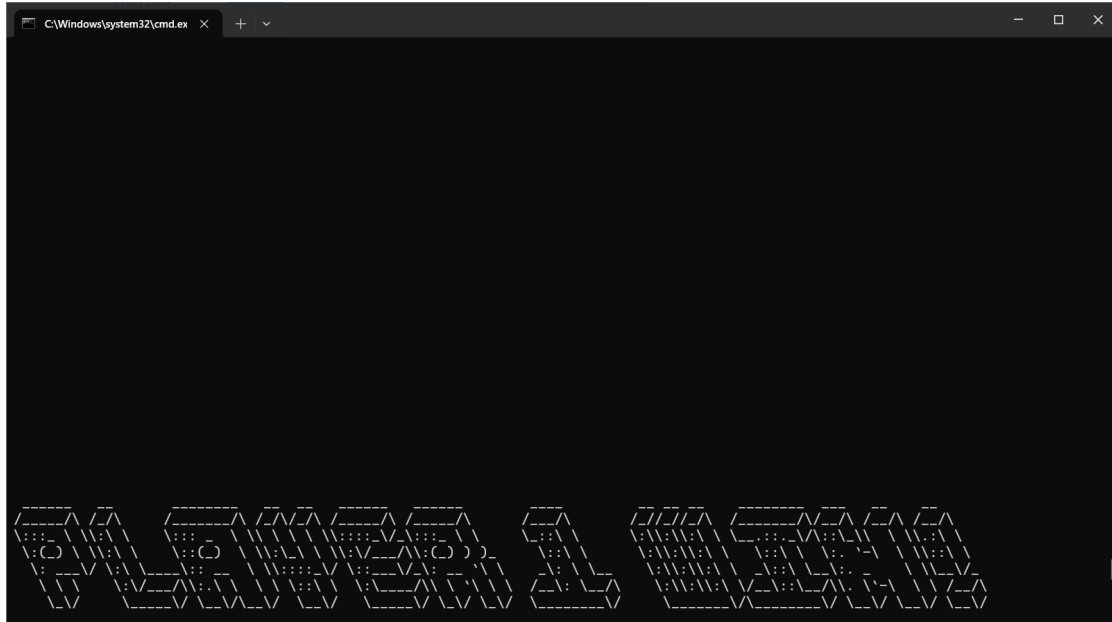


Figure 16: Player 1 WIN!

- After announcing the winner, the game asks if players want to play again:

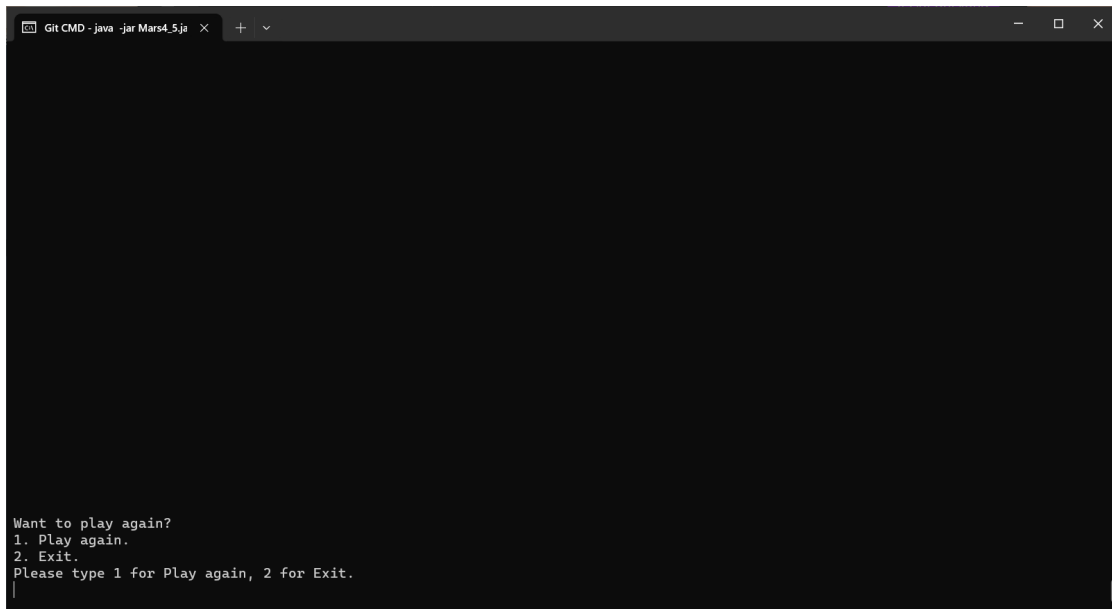


Figure 17: Choices

- If players want to exit the game, a thankful prompt appear:

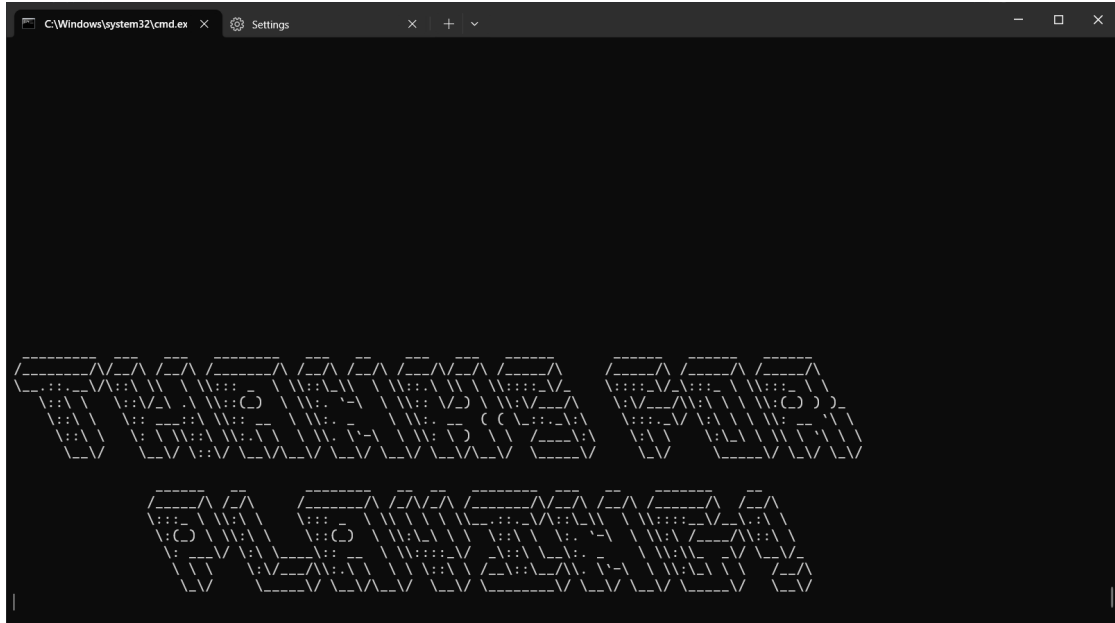


Figure 18: Thanks for playing!

4 Conclusion

In conclusion, this "BATTLESHIP" program made on MIPS has been an interesting and challenging experience. The program works smoothly with all errors are handled to help players having the best experience. However, there are still some difficulties that I cannot resolve like the boring black & white text-based interface, no sound effects can be inserted,.. these things will make players get bored after a few times playing.

From doing this, I now have gained a better understanding of MIPS in term of mechanism and coding.

5 References

- [1] Pete Sanderson Ken Vollmar. *MARS MIPS simulator - Missouri State University*. URL: <https://courses.missouristate.edu/kenvollmar/mars/> (visited on 12/01/2023).
- [2] Missouri State University. *MIPS syscall functions available in MARS*. URL: <https://courses.missouristate.edu/kenvollmar/mars/help/syscallhelp.html> (visited on 12/01/2023).