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

COMPUTER ARCHITECTURE (CO2008)

Assignment

BATTLESHIP

Instructor: Phạm Quốc Cường
Student: Hồ Khánh Nam - 2252500

HO CHI MINH CITY, DECEMBER 2023

Table of contents

1. Introduction

In this MIPS assignment, my goal is to replicate the classic Battleship game with the constraint of limited, simple resource and assignment requirements. The game runs on the MARS MIPS simulator – a combined assembly language editor for assembling, simulating and debugging for MIPS processor.

1.1 Game overview

The Battleship game will involve a setup phase where 2 players can strategically set up their ships. Particularly, players will be prompted to insert ship coordinates. After the setup phase, the players moves to the combat phase where 2 players take turn targeting a cell from the other player's board map to attack blindly and after each attack turn, player will get announced in the screen whether the opponent's ship get hit or missed from the attack. When the player's ships in the board map is completely destroyed (which means that there are no ships occupied in the map), the other player would WIN the game.

1.2 Configuration

The setup phase requires players to arrange a fixed number of ships with predefined sizes. Each player have to set 3 2x1 ships, 2 3x1 ships and 1 4x1 ship. Importantly, ships cannot overlap with each other, ensuring a fair and challenging game experience.

1.3 Utilization

To complete the Battleship game implementation, I apply the arithmetic, data transfer, conditional and unconditional jump instructions. I also use the stack addressing register for procedure call, use syscall instruction to read/write integer, string, load, store address from random access memory and read/write to the separate file.

1.4 Game scope

The game involves a 7x7 matrix for each player to set up the ships. The number of 3 kinds of ships (2x1, 3x1 and 4x1 ships) is fixed. Ship coordinates would range from 0 to 6, with no allowance for entering more than one digit, wrong size of the ships, negative number or any character.

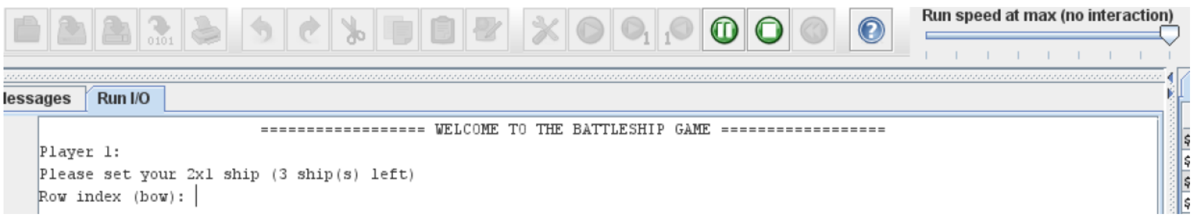
2. Design

2.1 Game display

In the game program, I would divide the program into 2 parts:

2.1.1 Set-up phase

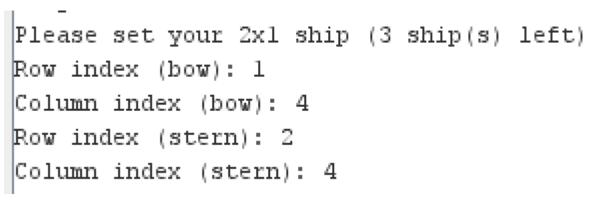
In this phase, I would implement for the introduction of the game, the UI of the game when getting started would show up similar to the **Figure 2.1**:



```
                  ================== WELCOME TO THE BATTLESHIP GAME ==================
Player 1:
Please set your 2x1 ship (3 ship(s) left)
Row index (bow):
```

Figure 2.1

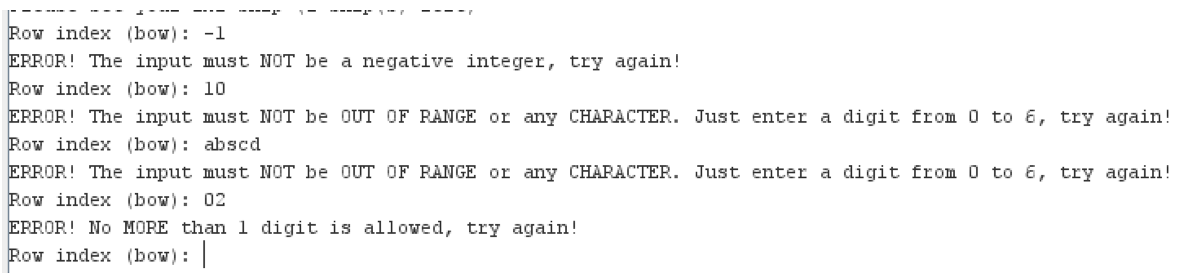
After the introduction, each of 2 players will get into the game by entering the coordinate of the bow and stern of each type of ships, the number of the ships that haven't set up yet will be shown on the screen, for example:



```
Please set your 2x1 ship (3 ship(s) left)
Row index (bow): 1
Column index (bow): 4
Row index (stern): 2
Column index (stern): 4
```

Figure 2.2

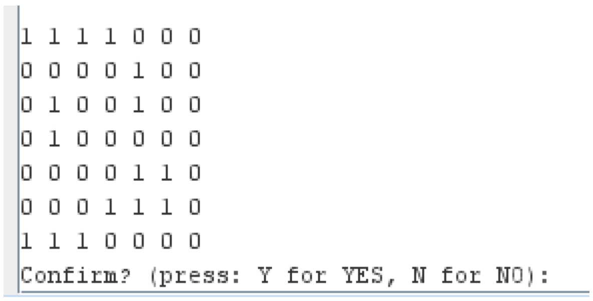
Note that the coordinate of the ship is in the range from 0 to 6 and only 1 digit number is allowed and the ship must be only set horizontally or vertically. Any exception would get the error announcement on the screen



```
Row index (bow): -1
ERROR! The input must NOT be a negative integer, try again!
Row index (bow): 10
ERROR! The input must NOT be OUT OF RANGE or any CHARACTER. Just enter a digit from 0 to 6, try again!
Row index (bow): abscd
ERROR! The input must NOT be OUT OF RANGE or any CHARACTER. Just enter a digit from 0 to 6, try again!
Row index (bow): 02
ERROR! No MORE than 1 digit is allowed, try again!
Row index (bow):
```

Figure 2.3

After showing the error announcement, the program will request the player to enter the coordinate of the ships again. Each of players take turns to enter their ships's coordinate strategically. When the first player finish the coordinate input, there will be the player's ships setup matrix showing up and a prompt message to check and confirm the ships coordinate again before letting the second player enter the coordinate

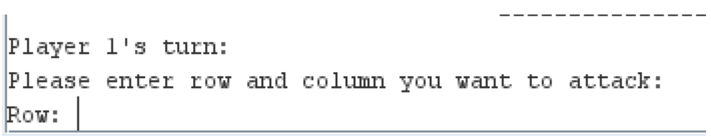


```
1 1 1 1 0 0 0
0 0 0 0 1 0 0
0 1 0 0 1 0 0
0 1 0 0 0 0 0
0 0 0 0 1 1 0
0 0 0 1 1 1 0
1 1 1 0 0 0 0
Confirm? (press: Y for YES, N for NO):
```

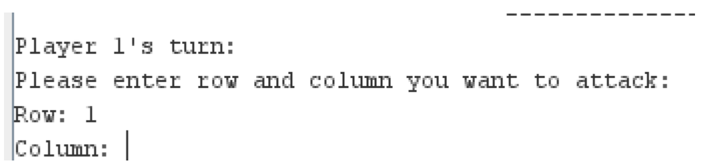
Figure 2.4

2.1.2 Combat phase

When 2 players have set up their ships into the matrix, the game officially get started. The rule for the game is the sam as the classic Battleship game: each player takes turn to target the opponent's ships by choosing 1 arbitary ceil on the opponent's board matrix to attack blindly. After attacking on the chosen ceil, there will be announcement for the player to inform that whether the attack hit the part of occupied ship on the targeted ship or not. In the program, after the setup phase, the game will ask the player to enter ceil's coordinate (row and column) on opponent's matrix to target:



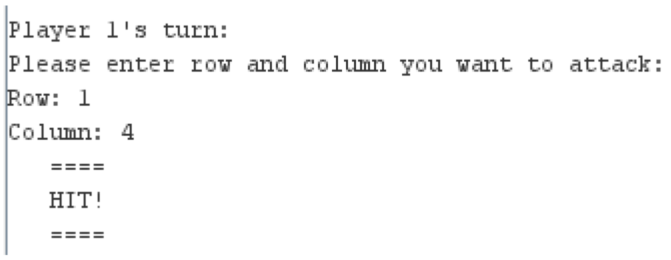
```
                                                ---------------
Player 1's turn:
Please enter row and column you want to attack:
Row:
```



```
                                                ---------------
Player 1's turn:
Please enter row and column you want to attack:
Row: 1
Column:
```

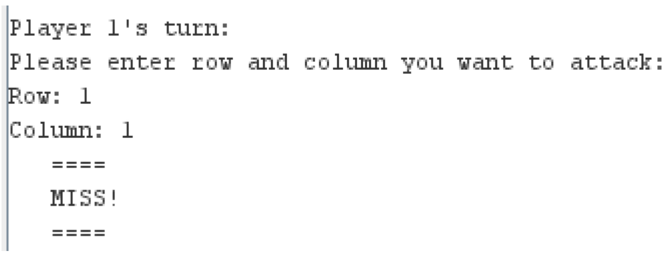
Figure 2.5

After entering row and column of the targeted ceil. The program wil announce for the player that the chosen ceil has been HIT or MISSED. For example in **Figure 2.6** and **Figure 2.7**:



```
Player 1's turn:
Please enter row and column you want to attack:
Row: 1
Column: 4
   ====
   HIT!
   ====
```

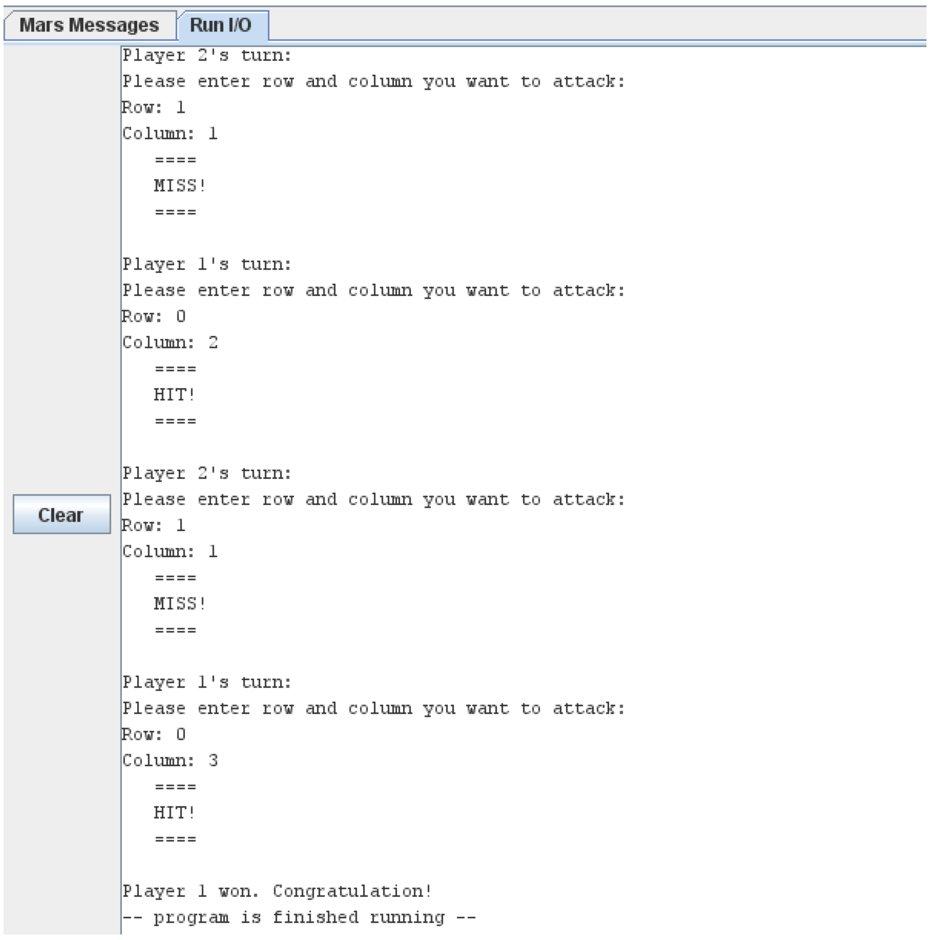
Figure 2.6 The attack HIT the occupied part of the ship



```
Player 1's turn:
Please enter row and column you want to attack:
Row: 1
Column: 1
   ====
   MISS!
   ====
```

Figure 2.7 The attack MISSED the occupied part of the ship

When first player finish attack the ships, the second player takes turn to attack the first player ships. Notice that each player only enter the coordinate for 1 ceil in each turn. 2 players alternatively attack their opponent until one of 2 players have their ships completely destroyed (which means that all the occuppied ships are not available on the matrix) first, then the other player will win the game. For instance in **Figure 2.8.**



```
Player 2's turn:
Please enter row and column you want to attack:
Row: 1
Column: 1
   ====
   MISS!
   ====

Player 1's turn:
Please enter row and column you want to attack:
Row: 0
Column: 2
   ====
   HIT!
   ====

Player 2's turn:
Please enter row and column you want to attack:
Row: 1
Column: 1
   ====
   MISS!
   ====

Player 1's turn:
Please enter row and column you want to attack:
Row: 0
Column: 3
   ====
   HIT!
   ====

Player 1 won. Congratulation!
-- program is finished running --
```

Figure 2.8 Player 1 won the game

2.2 Exception

During the gameplay, there might be some error occurred from the player's mistake of entering wrong format of the input or violate the game scope requirement, which make the program stops executing. To handle these mistakes, I would list some exception that may encounter in the game program:

- Entering the negative number (e.g. -1, -199, -48, -6,…)
- Entering the character that is not a digit number (e.g. @, #, a, B,…)
- Entering more than one digit even though the number is in the range from 0 to 6 (e.g 01, 02, 05, 06, …)
- Entering the number that is out of range from 0 to 6 (e.g. 10, 100, 21, 600)
- Setting the the ship diagonally (e.g rowbow = 1, columnbow = 4, rowstern = 2, columnstern = 5)
- Setting the ship that overlaps the ships already set on the matrix

- The player accidentally press enter without entering any input

3. Program implementation

3.1 Algorithm

The game procedures will be presented in the pseudo code here:

```
Begin program
    M ⟵ 1st player's matrix  // Load address of 1st player's matrix
    Input(M)
    N ⟵ 2nd player's matrix  //Load address of 2nd player's matrix
    Input(N)
    Combat:
    N1 ⟵ 16 //Number of occupied ceils of 1st  player
    N2 ⟵ 16 //Number of occupied ceils of 2st  player

    While N1 > 0 && N2 > 0 do:
            // first player attack
            Attack(N, N2)
            // second player attack
            Attack(M, N1)

    End while

    If N1 = 0 then player 2 Win
    Else player 1 Win

End program

Procedure Input(matrix M):
Begin:
        While not confirm do:
                Insert_2x1_ships(M)
                Insert_3x1_ships(M)
                Insert_4x1_ship(M)
                If confirm the matrix, then end while-loop
                Else reset the matrix, continue while-loop
        End while
End procedure

Procedure Insert_2x1_ships(matrix M):
    Begin:
    S ⟵ 3 //number of 2x1 ships
    While S > 0:
        startInput(M)
        S = S - 1
    End while
End procedure

Procedure Insert_3x1_ships(matrix M):
    Begin:
    S ⟵ 2 //number of 3x1 ships
    While S > 0:
        startInput(M)
        S = S - 1
    End while
End procedure

Procedure Insert_4x1_ship(matrix M):
Begin:
     S ⟵1 //number of 4x1 ship
     While S > 0:
        startInput(M)
        S = S - 1
     End while
End procedure

Procedure startInput(matrix M):
Begin:
        Input row_bow
        Input column_bow
        Input row_stern
        Input column_stern
        If row_bow == row_stern
                For i = column_bow to column_stern:
                      set all the ceils in M at the row_bow row and i column to value of 1
        Else if column_bow == column_stern
                For i = row_bow to row_stern:
                      set all the ceils in M at the column_bow column and i row to value of 1
End procedure


Procedure Attack(matrix M, int S):
Begin:
        Input row
        Input column
        T ⟵ value of ceil at row and bow
        If  T == 1, then print HIT && S = S – 1 // decrease the number of occupied ceil
        Else print MISSED
End procedure
```

3.2 Implemtation

3.2.1 Setup phase

In the MIPS program, I run the game following by the stage that I mentioned above, with the stages include the setup phase (the input procedure) and combat phase (processing procedure and output procedure). Before getting into the main program, I have set up some resource in the random access memory (RAM) for the gameplay such as the announcement for getting HIT, MISSED or errors, prompts for the input and most importantly, the matrices for 2 players. The matrix I use for the game is a 1-dimension array, with 49 elements represent for 49 ceils in the matrix. In every prompt of the game, I always use the syscall instruction li $v0, 4 to print the prompt.

To process the matrix in entire program, I use the **row-major order** to access the ceil of the matrix in right coordinate. Specifically, when the base address from the memory is in a register, I need to calculate the offset for the 1-dimension array by using the row-major order:

$$offset = rowIndex * colSize + colIndex$$

where rowIndex is the row index of the ceil we want to access, colIndex is the column index of the ceil and colSize is the size of the matrix's column (in this game, the column size is 7). After that, we can calculate the address of the targeted ceil by adding the base address of the matrix array with the multiplication of offset value and an integer value of 4, since the array needs **4 bytes** to leave enough between each offset's address for 1 integer value or a word. The formula of the chosen ceil's address:

$$address = base\_address + offset * 4$$

Figure 3.1 and **Figure 3.2** illustrate the way to access to a ceil's address to input the value of the ceil. When we want to access the coordinate of a ceil, which have the row with the value of 1 and the column with the value of 1 (in the range from 0 to 6).

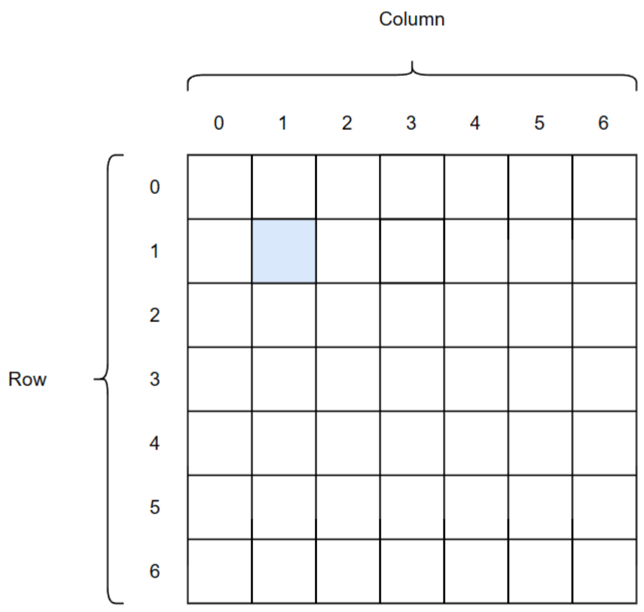


Figure 3.1 Player's Matrix board

Then we can calculate the offset value:

$$offset = 1 * 7 + 1 = 8$$

Which means that we can access to the ceil with coordinate (1,1) or the array's element with offset value of 8. After that, we obtain the selected ceil's address by multiplying the data size (= 4) with the offset and adding with base address of the array:

$$address = base\_address + 8 * 4 = base\_address + 32$$

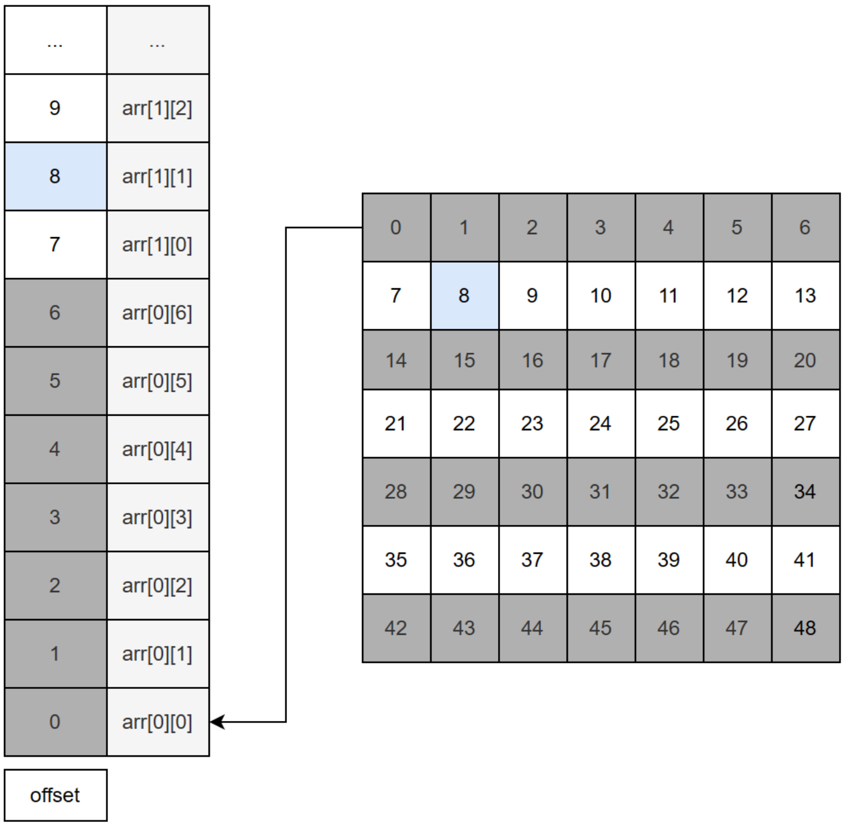
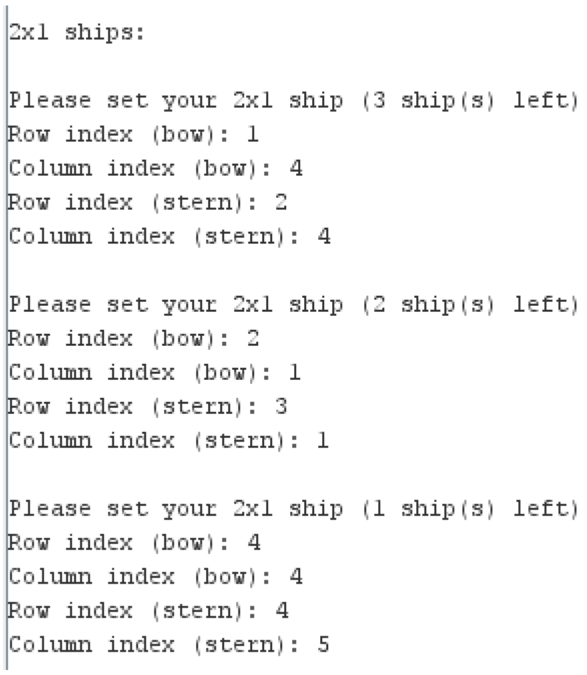


Figure 3.2 The ceil in coordinate (1,1) with offset value of 8

In the setup phase, the game require the player to enter the coordinate of each type of the ship, which includes the row and bow of the ship's bow and stern, and in each type of ships, there are particular numbers of ships for the input, therefore I use the loop to make the player enter all the number of ships in that type.



```
2x1 ships:

Please set your 2x1 ship (3 ship(s) left)
Row index (bow): 1
Column index (bow): 4
Row index (stern): 2
Column index (stern): 4

Please set your 2x1 ship (2 ship(s) left)
Row index (bow): 2
Column index (bow): 1
Row index (stern): 3
Column index (stern): 1

Please set your 2x1 ship (1 ship(s) left)
Row index (bow): 4
Column index (bow): 4
Row index (stern): 4
Column index (stern): 5
```

Figure 3.3

To implement the input, I use load address instruction to load the matrix of the first player into a temporary register, then I use the jal instruction to jump to Input Procedure. In the Input Procedure, each type of the ships is a procedure for the program to execute (there are 3 input procedure represent for 3 types of the ships).

Beside the prompt for input the ships, there is an update information next to each prompt for the player to know that how many ships left to enter the input. However, during the input, players can mistakely enter wrong input format like enter negative integers, enter wrong size of the ships, out of range, …. .Therefore, instead of using the read integer instructions, I use the string input to check the errors by using *beq* and *bne* instruction to compare each character of string with '\n' character (or line feed character in ASCII), which is illustrated in **Figure 3.4.**

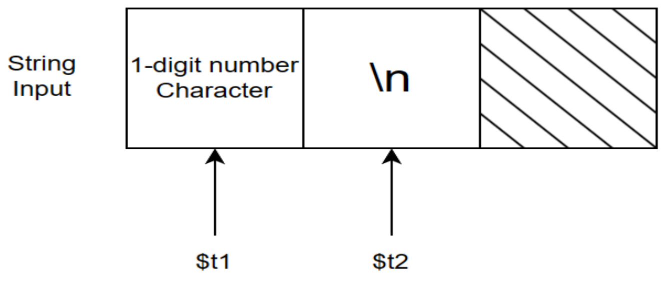
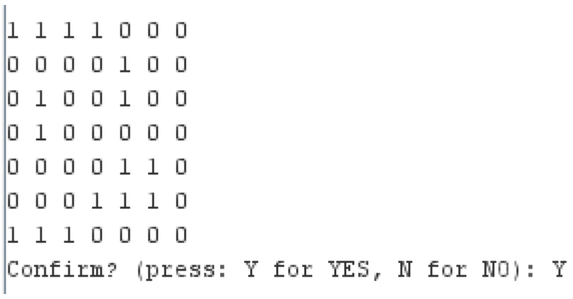


Figure 3.4 The input string with 1 number digit allowed

Each character of the string would be loaded in a register by using load byte instruction since a character takes only 1 byte (8 bits). In the checking character steps, I would check the first 3 characters in the input string to handle some cases of exception that will be explained in the Exception handling subsection.

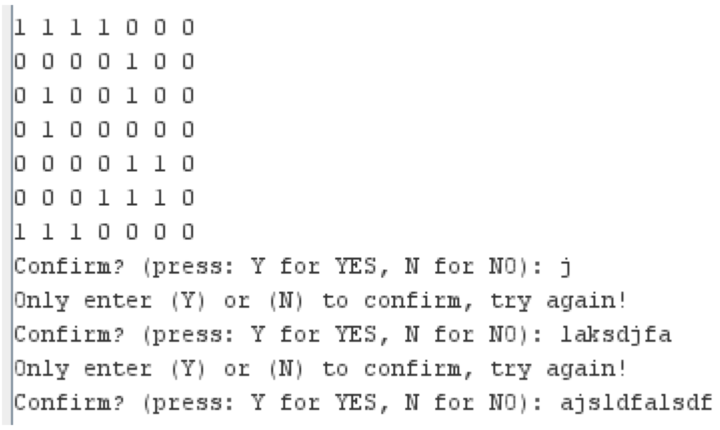
After the checking steps, each player needs to confirm the setup matrix in case that the player want to adjust the ships setup. The confirm prompt allows the player to enter only the characters 'Y' for Yes or 'N' for No, other exception will be considered as an error. If the player presses Y, then the program would get to the setup phase for the second player or the combat phase if the player 2 confirmed the matrix. When the player presses N, the the player's matrix will be reset all the ceil vallue to 0 by using the row-major order.



```
1 1 1 1 0 0 0
0 0 0 0 1 0 0
0 1 0 0 1 0 0
0 1 0 0 0 0 0
0 0 0 0 1 1 0
0 0 0 1 1 1 0
1 1 1 0 0 0 0
Confirm? (press: Y for YES, N for NO): Y
```

Figure 3.5

Other characters that is not character 'Y' or 'N' would be the exception



```
1 1 1 1 0 0 0
0 0 0 0 1 0 0
0 1 0 0 1 0 0
0 1 0 0 0 0 0
0 0 0 0 1 1 0
0 0 0 1 1 1 0
1 1 1 0 0 0 0
Confirm? (press: Y for YES, N for NO): j
Only enter (Y) or (N) to confirm, try again!
Confirm? (press: Y for YES, N for NO): laksdjfa
Only enter (Y) or (N) to confirm, try again!
Confirm? (press: Y for YES, N for NO): ajsldfalsdf
```

Figure 3.6

3.2.2 Combat phase

After the setup phase, the players get into the combat phase. As the pseudocode presented in the Algorithm section, first we set the number of all occupied ships's ceils for each player to the value of 16. In the program, I have set in 2 registers *$s3* and *$s5*

I also use the loop to makes the players to take turn attacking the ships until one of the players has the ships completely destroyed first. In the Attack procedure in each turn, the player is asked to type the row and column of the ceil to attack the opponent's ship, and then applying **row-major order** to find the address of the targeted ceil and check the value of the ceil. If the ceil has the value of 1, which means that the ceil is occupied by the opponent's ship, then the program will print **HIT!** and set the ceil's value back to 0. Otherwise the program will print **MISSED!** and not do anything with the matrix.

There will be some exception error when typing the input like negative number, more than 1 digit number, character,… and it can be handled by using the reading string instruction and the same instructions for handling exception that I used for the setup phase.

3.2.3 Writing file

To record all the gameplay of 2 players, I use the writing file instruction to write all the input and output of 2 players and the prompts of the game. In the **main** procedure, I first declare the name of the file to write in data type (In my program, the name of the file set in data type is "game_play.txt"). The **Figure 3.7** shows the writing file will be created at the folder which has the MARS MIPS application in the computer after the program termination.

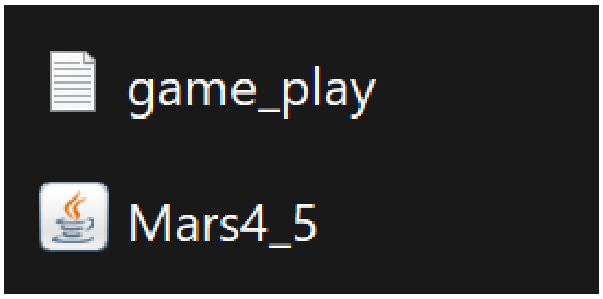
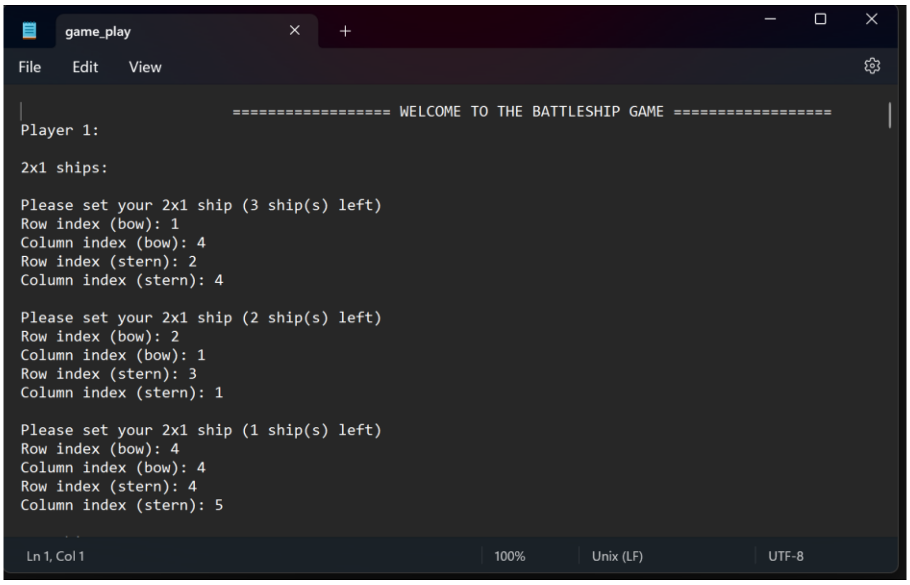


Figure 3.7

I use the open file instruction **li $v0, 13** open the writing file and close file instruction **li $v0, 16** at the beginning and the end of the program, respectively. In each printing instruction and input instruction, I use **li $v0, 15** to write the content of printing string into the file. With the printing instruction, there would be a fixed number of the buffer to hold all the characters of a string in the memory. However, in input instruction, the number of characters in a string is not fixed to determine the buffer for the string to write into the file. Therefore, I can use a procedure to count the number of characters of the string input of the player by using jump instruction to loop the counting instruction and the procedure can return the number of characters.



```
                        ================== WELCOME TO THE BATTLESHIP GAME ==================
Player 1:

2x1 ships:

Please set your 2x1 ship (3 ship(s) left)
Row index (bow): 1
Column index (bow): 4
Row index (stern): 2
Column index (stern): 4

Please set your 2x1 ship (2 ship(s) left)
Row index (bow): 2
Column index (bow): 1
Row index (stern): 3
Column index (stern): 1

Please set your 2x1 ship (1 ship(s) left)
Row index (bow): 4
Column index (bow): 4
Row index (stern): 4
Column index (stern): 5
```

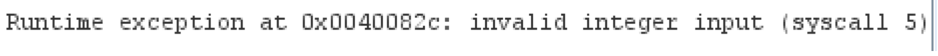
Figure 3.8 The result of writing file after the program terminated

3.2.4 Exception handling

To avoid program crash during the game play, I use instructions to check the character of the input string since using reading integer instrucion to enter the input might get into problem when the player mistakenly type wrong number format or the setup rules. These are some cases of exception I would consider to fix the problem:

- **Case 1 (No input entered):**

This is the case that when the player forget to enter the input, which may get the runtime error in MIPS shown in **Figure 3.9**.



```
Runtime exception at 0x0040082c: invalid integer input (syscall 5)
```

Figure 3.9

Since the reading integer instruction only accepts the integer value input and if the user enters only the enter key, the input would be a line feed character (‘\n’)

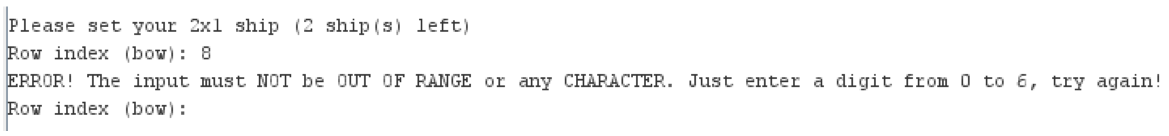
To solve this error, I would use the reading string instruction to to check if the first character of the input is a ‘\n’ character (10 in decimal in ASCII table) or not by using branch instruction (**bne $t1, 10, entered**). If the first character is not a ‘\n’, then the program will jump to the label **entered** to check for the next exception case.

- **Case 2 (The input is a negative number):**

If the first character is found to is a dash character (‘-’ or 45 in decimal ASCII) it would jump to the error announcement and jump back to the input to get the player to enter the input again. If it is not a negative number, we will get to the next checking steps

- **Case 3 (The input is out of range):**

In this case, I would consider the first character to check whether the number character is in the range from 7 to 9 (the range scope for the game is from 0 to 6) by using the instructions. Particularly, using the blt (branch if less than) instruction can check the character is less than the number 7 (or 55 in ASCII). if it is true, the program will jump to the label to go to the next checking steps, otherwise the program will announce the error to the player and ask the player to enter the input again:



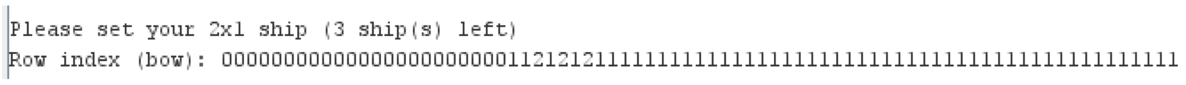
```
Please set your 2x1 ship (2 ship(s) left)
Row index (bow): 8
ERROR! The input must NOT be OUT OF RANGE or any CHARACTER. Just enter a digit from 0 to 6, try again!
Row index (bow):
```

Figure 3.10

Another out-of-range case is when the player input is the number value of coordinate more than 9, which can be solved be using beq instruction to check whether the second character is a newline character (10 in ASCII) or not.

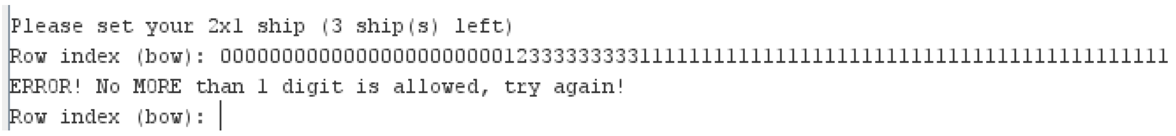
- **Case 4 (The input is more than 1 digit):**

In this case, all the input should be in 1 digit, even though the input might be in the range of game scope since the inputs are all in string **type?** and in some cases, the player can press for long character/number and if we use the read integer instruction for the input, we may get this program crash error shown in **Figure 3.9**:



```
Please set your 2x1 ship (3 ship(s) left)
Row index (bow): 00000000000000000000000112121211111111111111111111111111111111111111111111111111111111
```

Figure 3.11 The input of long number

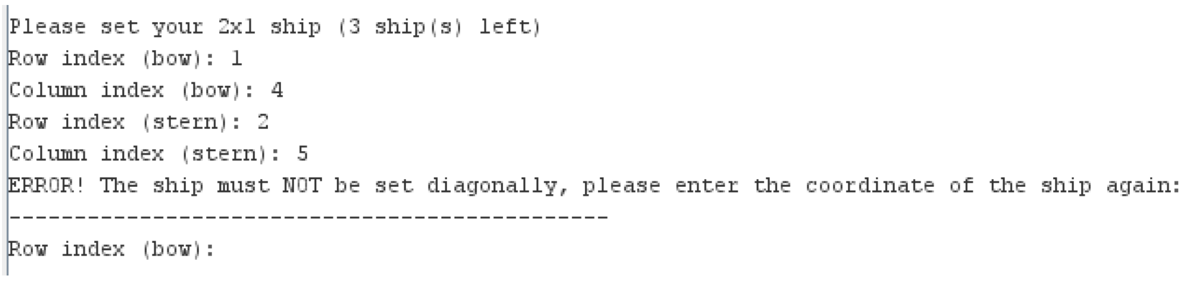


```
Please set your 2x1 ship (3 ship(s) left)
Row index (bow): 000000000000000000000001233333333311111111111111111111111111111111111111111111111111111
ERROR! No MORE than 1 digit is allowed, try again!
Row index (bow): |
```

Figure 3.12 Exception fixed

- **Case 5 (Diagonally ship setup):**

Figure 3.13 shows the case occurs when the ship is not set up in the same row and the same column



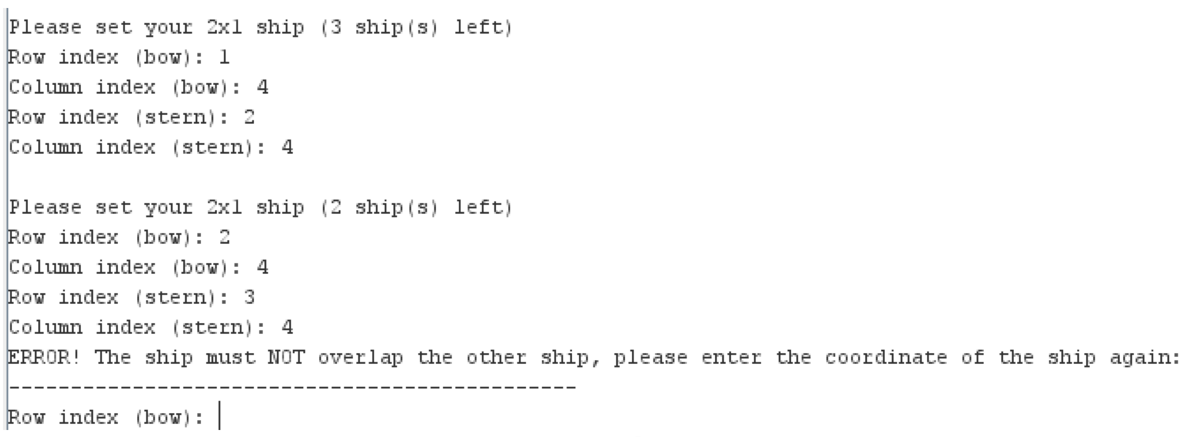
```
Please set your 2x1 ship (3 ship(s) left)
Row index (bow): 1
Column index (bow): 4
Row index (stern): 2
Column index (stern): 5
ERROR! The ship must NOT be set diagonally, please enter the coordinate of the ship again:
---------------------------------------------
Row index (bow):
```

Figure 3.13

- **Case 6 (Overlap and wrong ship's size):**

When the player finished the input of the coordinate of ship's bow and stern, the program will check if the ship overlaps the other ships or the ship is in the right size of ship type.

For the overlap case, I use procedure for overlap checking for the same-row-coordinate ships and same-column-coordinate ships, with the loop to access the ceils that the player wants to set up (using row-major order) and if one of ceils has the value of 1, then the program will print the Overlap error announcement and jump back to the input procedure and enter the input again



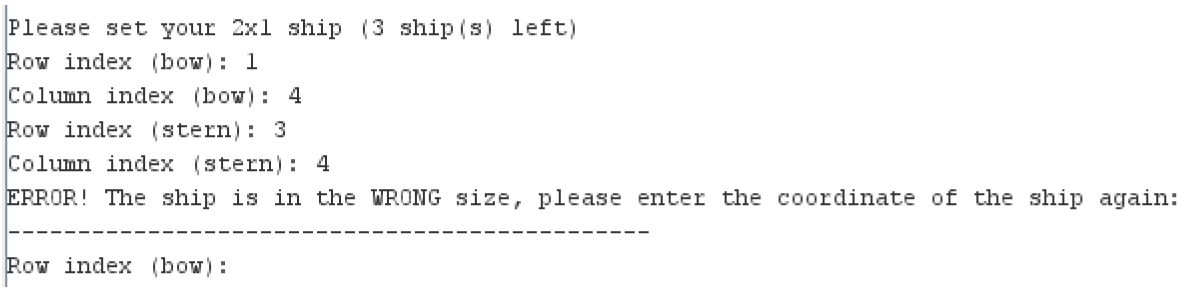
```
Please set your 2x1 ship (3 ship(s) left)
Row index (bow): 1
Column index (bow): 4
Row index (stern): 2
Column index (stern): 4

Please set your 2x1 ship (2 ship(s) left)
Row index (bow): 2
Column index (bow): 4
Row index (stern): 3
Column index (stern): 4
ERROR! The ship must NOT overlap the other ship, please enter the coordinate of the ship again:
----------------------------------------------
Row index (bow): |
```

Figure 3.14 Overlap error

In the checking procedure, I use a register as a return the result of overlap checking (In this part of code, I used register $s7) and beq instruction to check the result and decide for the next instruction. The $s7 can equal to 1 if the ship overlap and 0 otherwise.

For the wrong size case, I would check the distance from the bow's coordinate to the stern's coordinate, after the player entering the coordinate of the ship, then the program will calculate the distance between bow and stern of the column if the ship's rows of bow and stern are equal (row_{bow} == row_{stern}), or between row's bow and stern if the ship has the same column ($column_{bow}$ == $column_{stern}$). Each type of the ships has a different size, so in the insertion for each type would have a value of size (distance between bow and stern) for such type: The 2x1 ship's size is 1, 3x1 ship's size is 2 and 4x1 ship's size is 3. Before getting into the input procedure, I would assign the size value of each type into a register, so that in the Input procedure, the program can compare the distance of bow and stern of the player's input with the default size of the type that the player is entering

Figure 3.15 illustrates an example when the player enter the coordinate with row_{bow} = 1, $column_{bow}$ = 4, row_{stern} = 3, $column_{stern}$ = 4, then we will get the exception error announcement and ask the player to enter the coordinate of the ship again:



```
Please set your 2x1 ship (3 ship(s) left)
Row index (bow): 1
Column index (bow): 4
Row index (stern): 3
Column index (stern): 4
ERROR! The ship is in the WRONG size, please enter the coordinate of the ship again:
----------------------------------------------
Row index (bow):
```

Figure 3.15

Since the distance between rowbow and rowstern is 3 – 1 = 2, there will be 3 ceils on the row for a 2x1 ships, which is not an accurate size of the ship.