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

COMPUTER ARCHITECTURE - SEMESTER 231

ASSIGNMENT REPORT

Lecturer: Pham Quoc Cuong
Class: CC08
Students: Nong Thuc Khue - 2252385

HO CHI MINH CITY, DECEMBER 2023

Contents

1 Introduction

1.1 Battleship game

Battleship is a strategic board game between 2 players. In this assignment, each player must define their 3 2x1 ships, 2 3x1 ships, and 1 4x1 ship on their maps of a 7x7 grid. After the ships have been positioned, the game proceeds in a series of attacks from 2 players. In each round, each player takes a turn to attack a target square by announcing the coordinates of that square. If that square is part of an opponent's ship, that opponent marks "hit" on their own and announces what ship was hit. The winner is the player who hits all the ships of the opponent. The game is over when the winner is found.

1.2 MIPS assembly language

MIPS stands for Microprocessor without Interlocked Pipeline Stages. The assembly language of the MIPS processor is referred to as MIPS assembly language. Learning how to code in this language allows for a more in-depth grasp of how these systems work at a lower level. In this assignment, we are required to implement the Battleship game by using MIPS assembly language.

2 Algorithm explanation

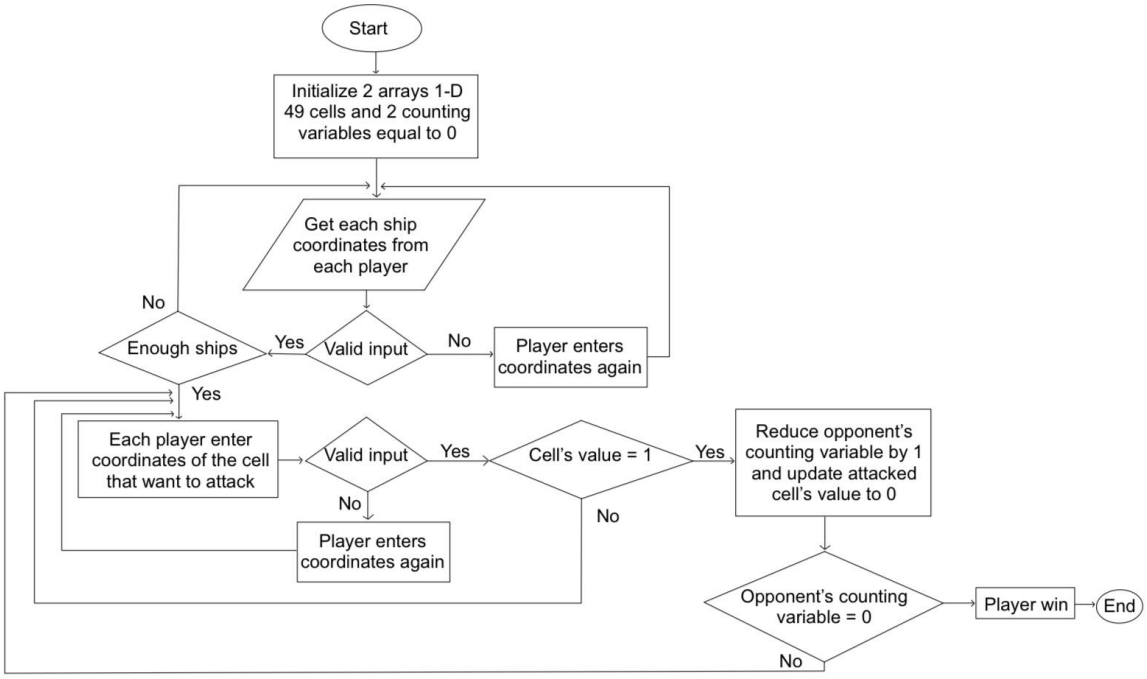
2.1 Idea explanation

The structure of this code includes 3 parts: initializing arrays and related variables for 2 players, positioning ships for each player, and 2 players starting attacking in turn. Firstly, initializing 2 arrays 1-D 49 cells of value 0 as maps of 2 players and counting variables for players. The maximum value of each counting variable is 16 because we have 3 2x1 ships, 2 3x1 ships, and 1 4x1 ship, which means there are 16 cells in a map that will be marked. Secondly, each player enters the coordinates of 3 2x1 ships, 2

3x1 ships, and 1 4x1 ship. The cell that is part of a ship is marked as value 1. Finally, each player attacks in turn by entering the coordinates of the cell they want to attack. If that cell is part of an opponent's ship, the value of that cell will turn from 1 to 0 and the counting variable of the opponent reduced by 1, and the counting variable of the opponent will be checked if it is 0 or not. If the opponent's counting variable is equal to 0, the player is the winner, otherwise, the game continues.

As the aforementioned idea, I use 1 print function for printing the map of each player and 3 main function groups in the code of the Battleship game: group of initialization arrays and related variables, group of positioning ships, and group of attack. In each group, there are functions for conducting the purpose of that function group (for instance, positioning ships by marking them by value 1 in the array). These functions will be explained in detail in the next subsection.

The flowchart below gives a general and understandable sight of the idea and the code:



2.2 Functions explanation

2.2.1 Group of initialization arrays and related variables

First of all, declare the array size of 196 for $matrixA$ and $matrixB$ as maps of player A and player B. Next, in function $init\_matrixA$, assigning $matrixA$ to register $a1, counting variable of player A to register $s1. In function $init\_matrixB$, assigning $matrixB$ to register $a2, counting variable of player B to register $s2. After that, storing value 0 to all elements of both arrays.

2.2.2 Group of positioning ships

Mutual functions that are used for getting coordinates input from 2 players are $input2\times1$, $input3\times1$, and $input4\times1$. Since there is more than 1 ship for 2x1 ship and 3x1 ship, I use loop for these two input functions of these ships. For each input, the $validIndex$ and $validSize$ functions are called (checking valid functions will be explained in detail below). If the input is valid, the cells that are part of the ship that has been input immediately will be marked by value 1, otherwise, the player must enter coordinates

again. The row and column coordinates of the bow are stored in register $t3 and $t4, and the row and column coordinates of the stern are stored in register $t5 and $t6. If $t3 is equal to $t5, the ship is vertical, otherwise, it is horizontal.

The *fillShip* function is used for marking all the cells that are part of ships. To calculate the position of the bow and the stern, apply the formula: position = (row coordinate × 7) + column coordinate. The position of the bow is stored in the register $t3 and the position of the stern is stored in the register $t5. However, if the value of $t3 is greater than $t5, swap them, since the code is designed to fill from $t3 to $t5, a smaller position to a greater position, thus, swap registers is for easy filling. After filling a ship, the counting variable is increased by the ship's size.

In the *startgame* function, initializing register $s5 as a temp variable for checking which player is inputting ships. The value of register $s5 equal to 0 means that player A is inputting ships, and if the value of register $s5 is equal to 1, player B is inputting ships. After a player finishes inputting ships, the program will conduct branch instructions. If $s5 is equal to 0, the program jumps to *inputCoordinateB*, and $s5 is increased by 1. If $s5 is equal to 1, the program will branch to the *inGame* function.

2.2.3 Group of attack

Starting with the *inGame* function, the register that stores the array of each player is moved to $s3, and the counting variable is moved to $s4. The program jumps to the *attack* function. At first, a player enters the coordinates of the cell that they want to attack and the function *validAttack* is called. If the input is invalid, the player is required to enter again, otherwise, the value at the position of coordinates will be loaded to register $a3 for checking. If it is equal to 0, the program prints "MISS!", jumps, and links to the *displayCount* function, which is used to display the counting variable of the opponent. Otherwise, the program prints "HIT!", turns the value of the attacked cell to 0, reduces the counting variable of the opponent by 1, and jumps and links to the *displayCount* function.

After each round, the *playerWin* function is called to check if one of the players wins. If the counting variable of a player is 0, the opponent is the winner, and the game ends. If 2 counting variables have not reached the value 0 yet, the game continues. To create the loop for the attack, I also use $s5 to check.

2.2.4 Valid checking functions

The *validIndex* function is for checking whether the coordinates are out of range. A valid coordinate must be in the range $[0, 6]$. This function only uses simple branch instructions. The *validSize* function is used to check the size of the ship and also check whether the coordinates make a line. This function uses the counting variable to know the exact size that it must check. If the counting variable is in the range $[0, 4]$, the size of the ship must be 2. If the counting variable is in the range $[6, 9]$, the size of the ship must be 3. If the counting variable is in the range $[12, 16]$, the size of the ship must be 4. To make a line, the row coordinates of the bow and the stern must be the same, or the column coordinates of the bow and the stern must be the same. If not, it is invalid. If the row coordinates of the bow and the stern are the same, get the difference of column coordinates to check size, it is similar to the case of the column coordinates of the bow and the stern are the same.

If coordinates are invalid, the program will jumps to *invalidInput* function. This function prints the announcement that the player enters invalid input and jumps to the *loop*2 × 1, or *loop*3 × 1, or *loop*4 × 1 to require player enter coordinates again.

2.2.5 Print function

This function is used to print the array of a player in square-matrix form and always being called whenever a ship input function finishes. A player can check if they entered the right expected coordinates. The function *printMatrix* has *printLoop*1 and *printLoop*2, which is similar to a nested loop for printing an array in square-matrix form. Before calling the function, I move a register that stores the array to $s0. To print a value, I load the value from the array to register $t2 and call the print instruction, then update the address and index for the next printing turn.

3 Conclusion

This report is written to explain the idea and algorithm that is applied to implement the MIPS code for the Battleship game. The source code is included in the BKel submission. I would like to express our gratitude toward Mr. Pham Quoc Cuong for teaching and helping me to conduct this project. I commit that the source code and this report are implemented only by me. For further information, I am pleased to answer in the presentation session or via email.