VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
School year 2023 – 2024 – Semester 1

ASSIGNMENT REPORT
BATTLESHIP
COMPUTER ARCHITECTURE LAB (CO2008)

Lecturer: **Dr. Phạm Quốc Cường**
Class: **CC08**
Student Name: **Nguyễn Quang Phú**
Student ID: **2252621**

Contents

I. INTRODUCTION

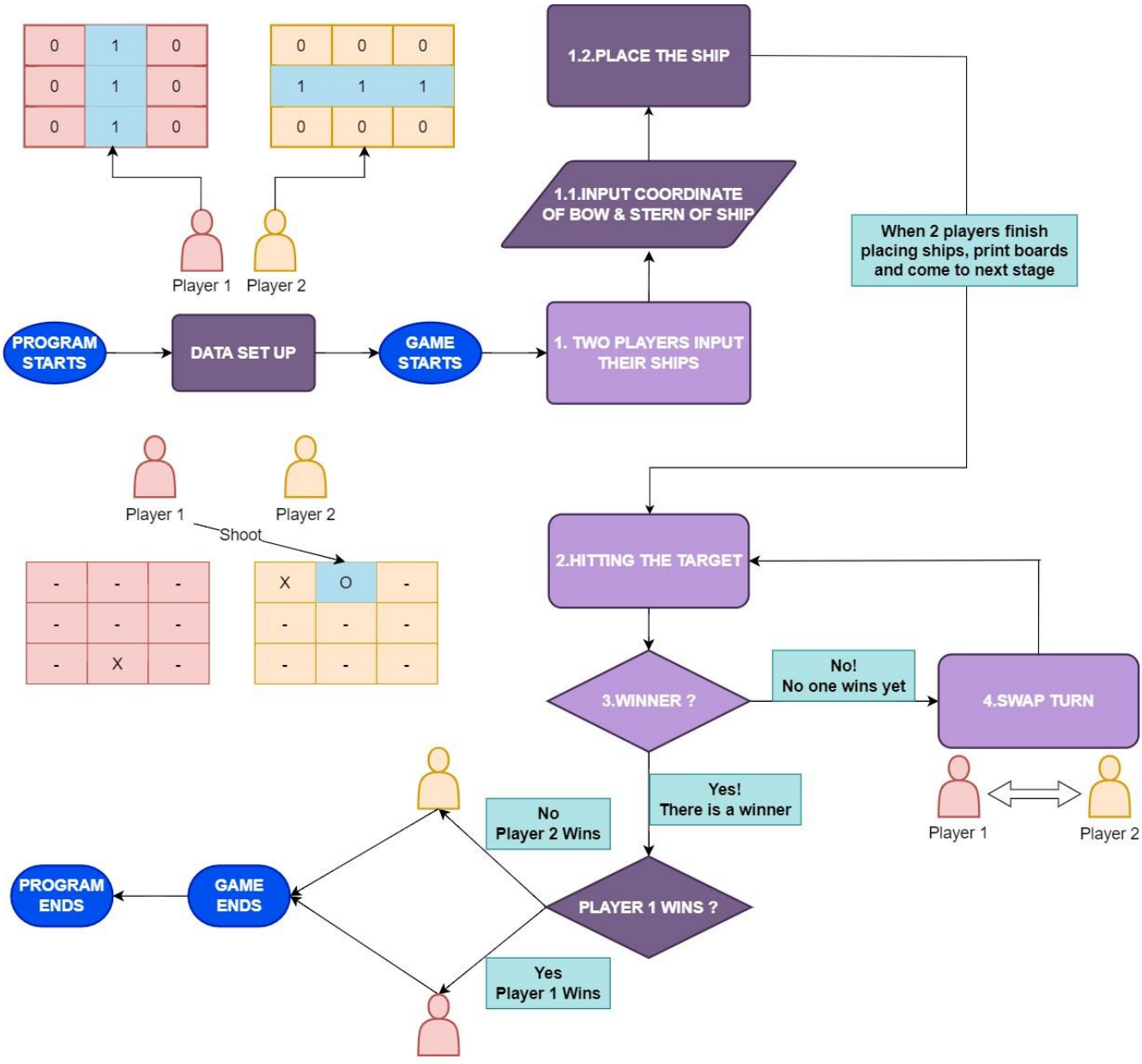
This report has been prepared for my Assignment of the course Computer Architecture. It includes all the details related to the Assignment, my approach, the ideas together with some code and the explanations.

This Assignment is a study of how to use the MARS MIPS simulator proficiently to apply things that we have learnt on the class. To be more specific, this Assignment is the mixing of the following topics: arithmetic and data transfer instructions, conditional branch together with unconditional jump instructions, and lastly it is the design of the procedures. Combining all of these topics to build a fantastic Battleship game is such a great way to revise what we have learnt so far.

Finally, it is my pleasure to write a report about what I have done on this Assignment, how I approach it and explains the things I do on this Assignment. Hopefully, this report will bring a comprehensive information and satisfy the need for a report.

II. IDEAS, PROCEDURES AND EXPLANATION

When receiving the Assignment, the first thing I think of is about the flow of the game, how it would be and what is the logic behind it. Then after thinking for a while, I have decided to make my game work within this flow.



***Picture 1**: The flowchart of how the logic of the game works*

From the flowchart, we can see that there are 4 main stages which are used in the flow of my game, they are: inputting-ship stage, hitting-target stage, check-winner stage, and how the data be stored in the data declaration stage. There are some other small stages too such as: swap-turn stage, or printing-board stage, and the stage for writing out the move of players into separate file.

GAMES RULES

Below is the Rule for Setting up players' boards:

- Players can input whatever ship of the size 2, 3 or 4 as they want, except that they can only put that ship onto their board if it is available.

- Errors will be displayed as player input the wrong coordinates for their ship, wrong size, …

```
THIS IS THE RULE FOR THE BATTLESHIP GAME.
1. THE SETTING STAGE
-  Player 1 takes the turn to place the ship.
-  When Player 1 finishes placing the ships, Player 2 takes turn.
-  Each Player has 3 ship 2x1, 2 ship 3x1, 1 ship 4x1.
-  Enter the ship using the format this format:
   <row_head> <col_head> <row_tail> <col_tail>
-  Player can place any ship in any order (except that there is still enough to place)
```

Picture 2: *This picture shows the rule for the setting stage of the game.*

Below is the Rule for the Hitting Stage of the Game:

- Player takes the turn to perform a hit on other board (the hit will be saved on player's view board).

- The Game will announce if it is a valid hit: if invalid, the game will ask the player to input the hit again, otherwise it will tell if it is a hit or a miss.

```
THIS IS THE RULE FOR THE BATTLESHIP GAME.
2. THE HITTING TARGET STAGE
-  Each player takes turn to place the hitting target.
-  Player inputs the row of the target, then the column of the target.
-  If Player hits the target before, Player is allowed to take another hit.
-  If the target is MISS or HIT, swap the turn, other Player takes turn to play.
```

Picture 3: *The picture shows the rule for the hitting stage of the game.*

INPUTTING SHIPS

First of all, I will discuss about the first process: Players inputting their ships.

```
#############################################################################################################
|                                        START PLACE YOUR SHIP.                                             |
#############################################################################################################
#                                Hello player 1, please place your ship!                                    #
#############################################################################################################
THIS IS THE RULE FOR THE BATTLESHIP GAME.
1. THE SETTING STAGE
-  Player 1 takes the turn to place the ship.
-  When Player 1 finishes placing the ships, Player 2 takes turn.
-  Each Player has 3 ship 2x1, 2 ship 3x1, 1 ship 4x1.
-  Enter the ship using the format this format:
   <row_head> <col_head> <row_tail> <col_tail>
-  Player can place any ship in any order (except that there is still enough to place).
#############################################################################################################
This is player 1 board.
     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 |
    --- --- --- --- --- --- ---
The remaining ship of size 2 = 3; size 3 = 2; size 4 = 1
```

Picture 4*:This picture shows what the users might see when they enter the game and play*

At first, player 1 takes turn to input all the ships he wants on the board. But there are some several cases that should be consider:

- The coordinates that the player input(s) is negative.

- The coordinate(s) is out of bound, which means it is larger than or equal to 6.

- Not enough input for `<row_head> <col_head> <row_tail> <col_tail>`

I solve the 2 above problems by immediately ask the user to input again until it is correct because the size of the board is 7x7 so the minimum index of the coordinate should be 0 and the maximum should be 6 ***(Picture 3).***

```
Please input a string of coordinates (<row_head> <col_head> <row_tail> <col_tail>): -1 1 1 1
Invalid input! Please input again!
Please input a string of coordinates (<row_head> <col_head> <row_tail> <col_tail>): 10
Invalid input! Please input again!
Please input a string of coordinates (<row_head> <col_head> <row_tail> <col_tail>): -1
Invalid input! Please input again!
Please input a string of coordinates (<row_head> <col_head> <row_tail> <col_tail>):   1      1
Invalid input! Please input again!
Please input a string of coordinates (<row_head> <col_head> <row_tail> <col_tail>):   1 1 1
Invalid input! Please input again!
Please input a string of coordinates (<row_head> <col_head> <row_tail> <col_tail>): 2 0 2 3
-------------------------------------------------------------------------------------------------------------

Success place ship of size 4.

-------------------------------------------------------------------------------------------------------------
```

Picture 5*: This picture shows that when the user input not correct, MARS will immediately ask the user to input again until it right*

But there are still some other problems:

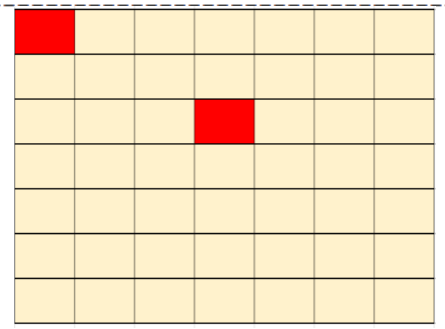
- The input ship is overlapped, or in the wrong direction (the right direction should be vertical or horizontal), out of the ship, or has the wrong size (the size of the ship should be 2, 3 or 4 as the description of the Assignment states that.

I choose to solve these situations by using a helper function called `placeShipPlayer`.

This function would help me to check if the 3 problems immediately above appears or not, if it appears, jump out of the helper function and ask the user to input again, otherwise if it does not appear, place the ship on the board by putting number of 1s on.

For checking if the ship is in the right direction, I might check if the 2 pairs `(row_head, row_tail)` and `(col_head, col_tail)` has ONE pair similar (not 2 pair as if it appears, it means that the ship has size 1, not a valid size in this game), otherwise MARS will print the announcement that your ship is in wrong direction and force the user to input again.

```
The remaining ship of size 2 = 3; size 3 = 2; size 4 = 1
Please input a string of coordinates (<row_head> <col_head> <row_tail> <col_tail>): 2 3 0 0
--------------------------------------------------------------------------------
The ship you input must be horizontal or vertical. Please reinput.
--------------------------------------------------------------------------------
```



Picture 6: *The picture shows that with that input, the ship direction is not horizontal or vertical, which is a wrong input*

For checking if the ship at the size the user wants to input is available, so I have to use 3 variables to tackle with the problem (each stores the number of available ship at each size). MARS will announce if the input ship is not available to use.

For checking the validity of the size of the ship, I still consider the 2 above pairs but now I take the one that has 2 different numbers, not losing the generality I assume that it is the pair `(row_head, row_tail)`. Then I calculate the size of the ship by the below formula and if it has the value in the range [2, 4] then it is valid, otherwise MARS will print the announcement that your ship has wrong size and force the user to input again.

$$size = Absolute(row\_head - row\_tail) + 1$$

```
Please input a string of coordinates (<row_head> <col_head> <row_tail> <col_tail>): 2 0 2 4
--------------------------------------------------------------------------------------------
The ship you input has invalid size. Please reinput.
--------------------------------------------------------------------------------------------


The remaining ship of size 2 = 3; size 3 = 2; size 4 = 0
Please input a string of coordinates (<row_head> <col_head> <row_tail> <col_tail>): 1 0 1 3
--------------------------------------------------------------------------------------------
There is no more ship of size 4! Please reinput.
--------------------------------------------------------------------------------------------


The remaining ship of size 2 = 3; size 3 = 2; size 4 = 0
Please input a string of coordinates (<row_head> <col_head> <row_tail> <col_tail>): 4 1 4 3
--------------------------------------------------------------------------------------------
Success place ship of size 3.
--------------------------------------------------------------------------------------------
```

Picture 7*: The picture shows what MARS announces when the input ship has invalid size, when there is no more ship available and when the user successfully places a ship*

We have finish discussing about the correct input of the ship, now let's consider how to place the ship on the board.

Assume that now it is player 1 turn and I will call his board `player1Board` for easier follow. Assume that the address of the player1Board starting at 0 (Picture below).

0	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	32	33	34
35	36	37	38	39	40	41
42	43	44	45	46	47	48

Picture 8: *This is the assumption address of a 7x7 player1Board of player 1*

Now only consider the case that the coordinate the user input is correct (as if it is incorrect, the user has to input again until it is right).

Assume the input is: `(row_head, col_head, row_tail, col_tail) = (2, 2, 2, 4)`

Then the bow of the ship has the coordinate (2, 2) and the stern has (2, 4).

Consider 2 pairs `(row_head, row_tail)` and `(col_head, col_tail)` we might see that the pair `(col_head, col_tail)` is different, which means that the ship will be place on row (as the ship can only be placed horizontally or vertically but the head and

tail of it have different value of y). So we will place a ship on the following cells: `player1Board[2][2]`, `player1Board[2][3]`, `player1Board[2][4]`. We will try to figure out how to calculate the address at these cells.

$$address = address\_of\_board + (\_x * 7 + \_y)$$

With: _x: is the row index, _y: is the column index.

Using the formula above, we can easily calculate the address of those cells we will use to place the ship is 16, 17, and 18 respective to the cell (2, 2), (2, 3) and (2, 4) (note that we have assumed that the `address_of_board` is 0 for easier calculation)

```
The remaining ship of size 2 = 3; size 3 = 2; size 4 = 1
Please input a string of coordinates (<row_head> <col_head> <row_tail> <col_tail>): 2 2 2 4
--------------------------------------------------------------------------------------------------------
Success place ship of size 3.
--------------------------------------------------------------------------------------------------------
This is player 1 board.
     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 | 1 | 1 | 1 | 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 |
    --- --- --- --- --- --- ---
The remaining ship of size 2 = 3; size 3 = 1; size 4 = 1
```

0	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	32	33	34
35	36	37	38	39	40	41
42	43	44	45	46	47	48

***Picture 9**: This picture shows where the ship might be placed with the input of 2, 2, 2, 4*

Using the same idea and approach above, we can place a vertical ship too.

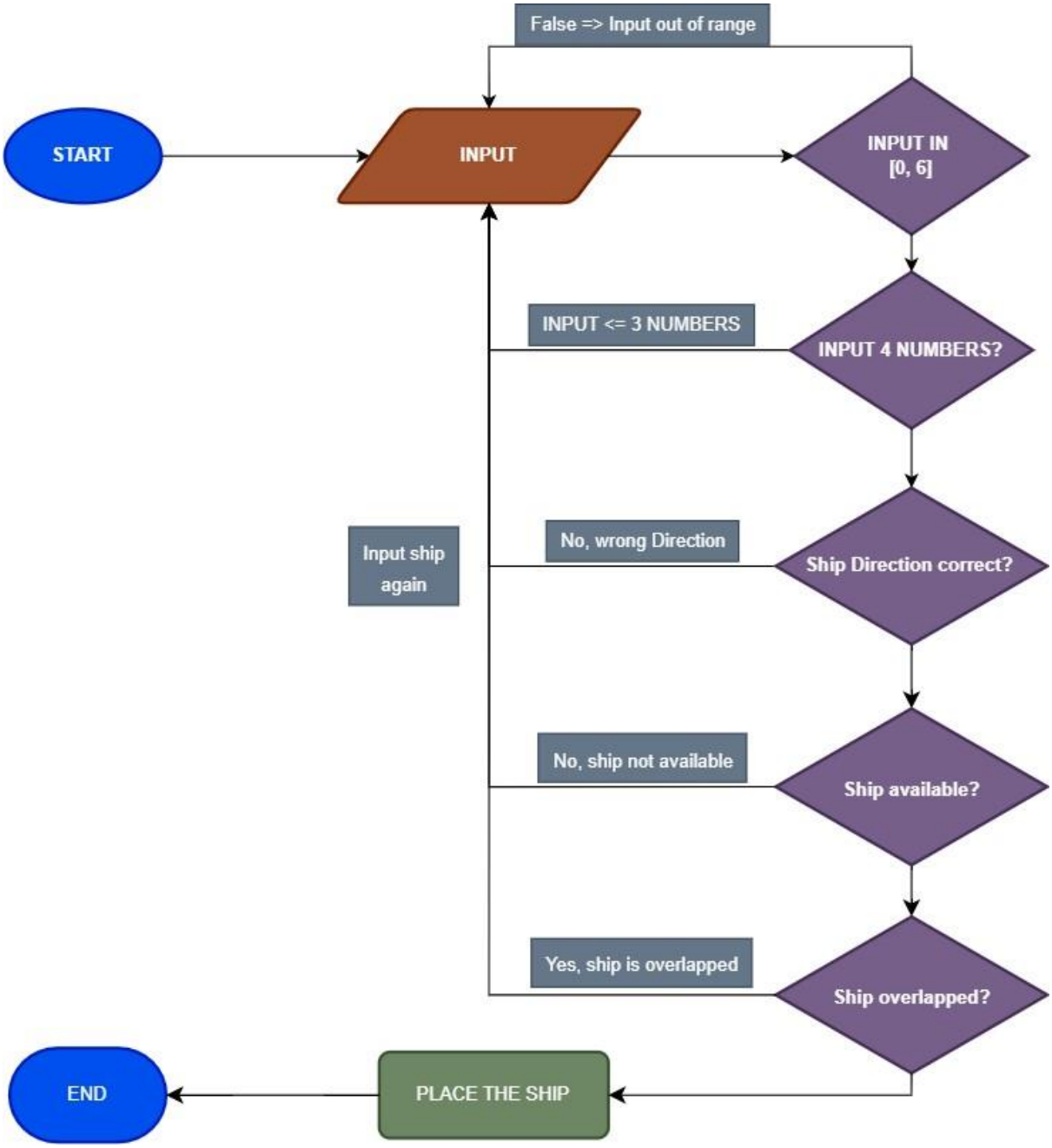
```
The remaining ship of size 2 = 3; size 3 = 1; size 4 = 1
Please input a string of coordinates (<row_head> <col_head> <row_tail> <col_tail>): 1 5 4 5
--------------------------------------------------------------------------------------------------------
Success place ship of size 4.
--------------------------------------------------------------------------------------------------------
This is player 1 board.
     0   1   2   3   4   5   6
    --- --- --- --- --- --- ---
0  | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
    --- --- --- --- --- --- ---
1  | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
    --- --- --- --- --- --- ---
2  | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
    --- --- --- --- --- --- ---
3  | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
    --- --- --- --- --- --- ---
4  | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
    --- --- --- --- --- --- ---
5  | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
    --- --- --- --- --- --- ---
6  | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
    --- --- --- --- --- --- ---
The remaining ship of size 2 = 3; size 3 = 1; size 4 = 0
```

0	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	32	33	34
35	36	37	38	39	40	41
42	43	44	45	46	47	48

***Picture 10**: The picture shows what happens when the input is 1, 5, 4, 5*

When both players finish placing their ships, it time for them to perform the hit on the boards of others.

THIS FLOWCHART SUMMARIZES HOW INPUTTING-SHIP STAGE WORKS



***Picture 11**: This picture shows how the program will process when the player wants to place a ship.*

HITTING THE TARGET

There are 3 situations occurs when the users input the coordinates of their hit:

- If input invalid coordinate (less than 0 or greater than 6), MARS will force the player to input again until it is correct.

- If they hit this before, MARS will announce that to the user and allow them to perform another hit. In my code I consider this hit INVALID. To check that the target is hit before, I will check on the view board of the player, if it is an 'X' (which means a hit) or a 'O' (which means a miss), then it means that we have hit this before. Also note that I choose to user a dash on the view board to show that this target has not been touched before, so I might choose the equivalent condition, if the target is not a dash, to check if the target is hit or not.

```
    0   1   2   3   4   5   6
  --- --- --- --- --- --- ---
0 | - | - | - | - | - | - | - |
  --- --- --- --- --- --- ---
1 | - | - | - | - | - | - | - |
  --- --- --- --- --- --- ---
2 | - | X | - | - | - | - | - |
  --- --- --- --- --- --- ---
3 | - | - | - | - | - | - | - |
  --- --- --- --- --- --- ---
4 | - | - | - | - | - | - | - |
  --- --- --- --- --- --- ---
5 | - | - | - | - | - | - | - |
  --- --- --- --- --- --- ---
6 | - | - | - | - | - | - | - |
  --- --- --- --- --- --- ---
Enter the position for your shot.
Enter the row for your shot:    2
Enter the column for your shot: 1
You have hit this before. Please reinput.
```

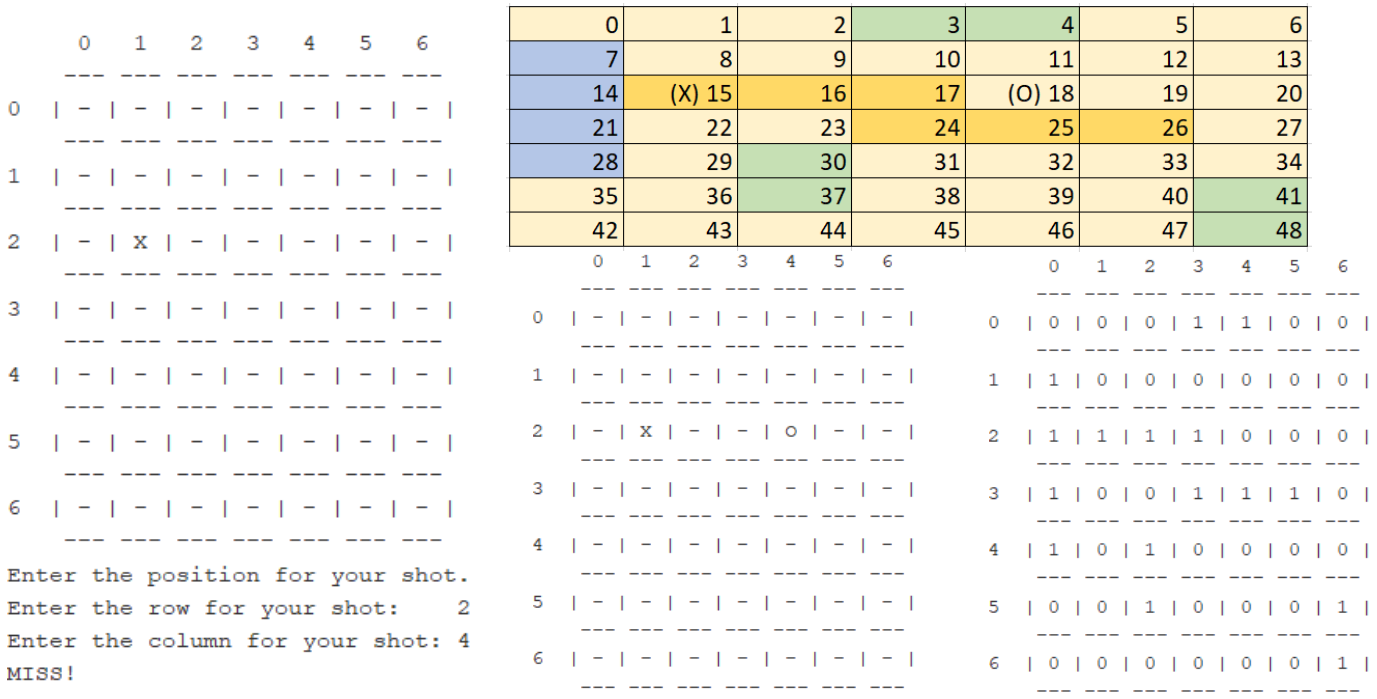
```
    0   1   2   3   4   5   6
  --- --- --- --- --- --- ---
0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
  --- --- --- --- --- --- ---
1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
  --- --- --- --- --- --- ---
2 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
  --- --- --- --- --- --- ---
3 | 1 | 0 | 0 | 1 | 1 | 1 | 0 |
  --- --- --- --- --- --- ---
4 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
  --- --- --- --- --- --- ---
5 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
  --- --- --- --- --- --- ---
6 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
  --- --- --- --- --- --- ---
```

0	1	2	3	4	5	6
7	8	9	10	11	12	13
14	(X) 15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	32	33	34
35	36	37	38	39	40	41
42	43	44	45	46	47	48

Picture 12: *This picture shows that at the cell (2, 1), we already hit it before, as at that position there is an 'X', then when we shot that again, MARS will announce like above*

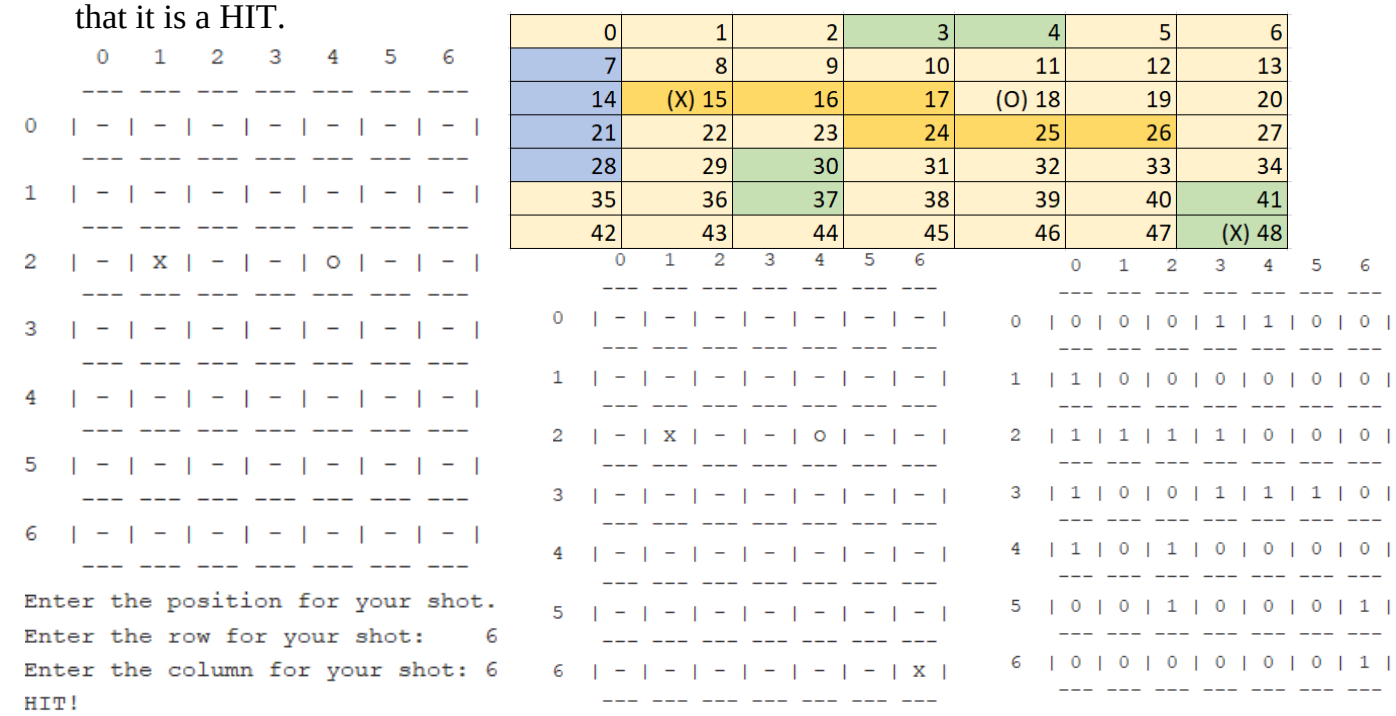
After passing the above situation, it means that now on the view board at the target it is a dash, which means we have not hit this before.

- If the target they choose on other board is not a part of a ship (which means it is a '0' on other board), then it is a miss (in my code I also consider them as MISS). I might set an 'O' (which represents a miss) on the view of this player and MARS also announce to the user that it is a MISS.



Picture 13: *This picture shows that we have not hit the target (2, 4) before, which presents by a dash -, and as the cell target is not a part of the ship, we put an 'O' to represent for a miss on the view later. MARS also announce that it is a MISS*

- If the target they choose on other board is a part of a ship (which means it is an '1' on other board), then it is a hit (in my code I also consider them as HIT). I might set a 'X' (which represents a hit) on the view of this player and MARS also announce to the user that it is a HIT.



Picture 14: *This picture shows that we have not hit the target (6, 6) before, which presents by a dash -, and as the cell target is a part of the ship, we put an 'X' to represent for a hit on the view later. MARS also announce that it is a HIT.*

```
------------------------------------------------------------------------------------------------------------
|                                          PLAYER 1 TAKES TURN.                                            |
------------------------------------------------------------------------------------------------------------

     0   1   2   3   4   5   6
    --- --- --- --- --- --- ---
0  | - | - | - | X | X | - | - |
    --- --- --- --- --- --- ---
1  | X | - | - | - | - | - | - |
    --- --- --- --- --- --- ---
2  | X | X | X | X | O | - | - |
    --- --- --- --- --- --- ---
3  | X | - | - | X | X | - | - |
    --- --- --- --- --- --- ---
4  | X | - | X | - | - | - | - |
    --- --- --- --- --- --- ---
5  | - | - | X | - | - | - | X |
    --- --- --- --- --- --- ---
6  | - | - | - | - | - | - | X |
    --- --- --- --- --- --- ---
Enter the position for your shot
Enter the row for your shot:    3
Enter the column for your shot: 5
HIT!
```

0	1	2	(X) 3	(X) 4	5	6
(X) 7	8	9	10	11	12	13
(X) 14	(X) 15	(X) 16	(X) 17	(O) 18	19	20
(X) 21	22	23	(X) 24	(X) 25	(X) 26	27
(X) 28	29	(X) 30	31	32	33	34
35	36	(X) 37	38	39	40	(X) 41
42	43	44	45	46	47	(X) 48

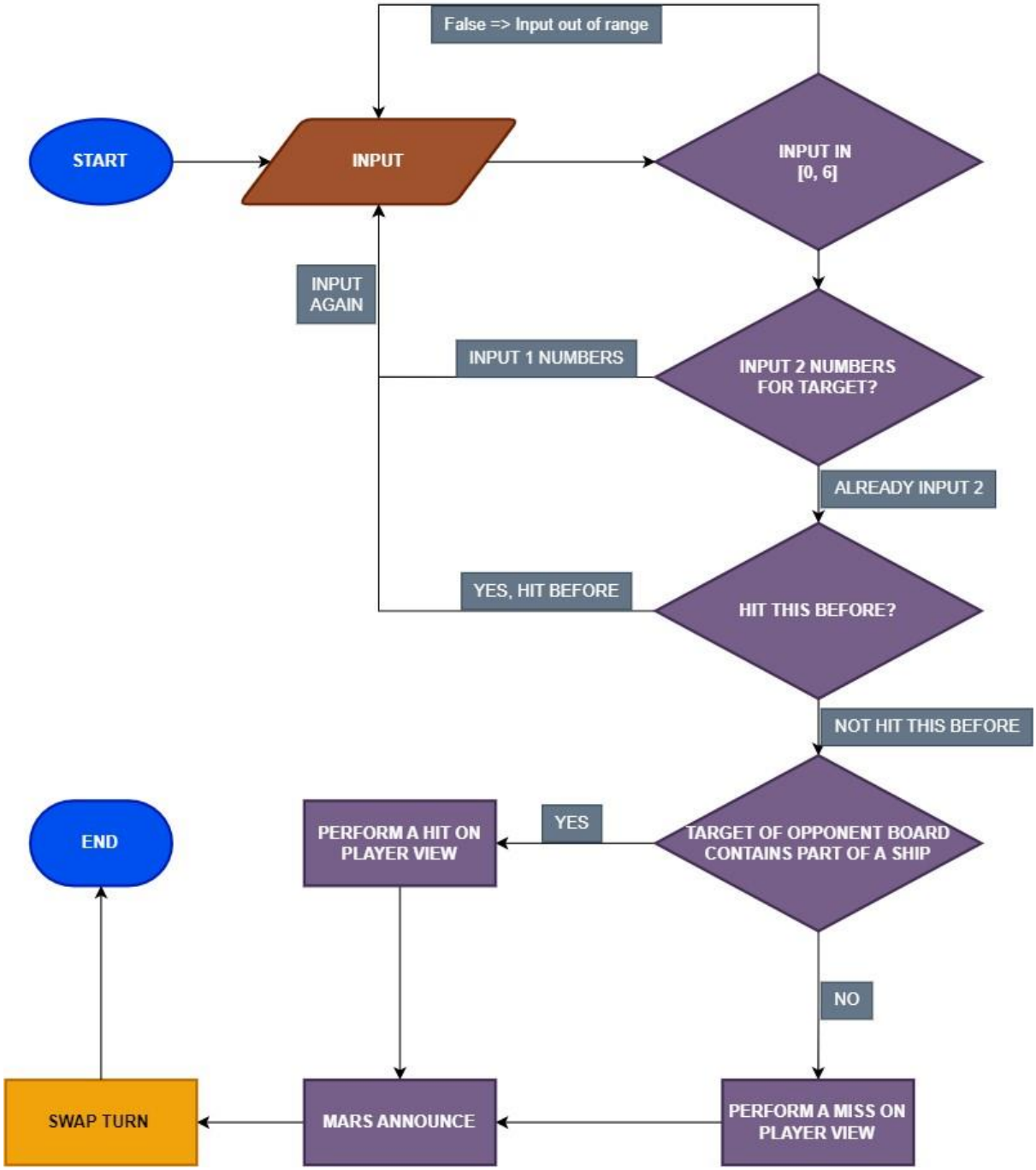
```
################################################################################################################
#                                              PLAYER 1 WINS!                                                  #
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
|                                              THE GAME ENDS!                                                  |
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
------------------------------------------------------------------------
|                          PLAYER 2 TAKES TURN.                        |
------------------------------------------------------------------------

HIT!
########################################################################
#                          PLAYER 2 WINS!                              #
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
|                          THE GAME ENDS!                              |
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Picture 15: *The picture shows what happens when the player hit the last part of the ship, which is colored with red on the colorful assumption board. MARS announces the Winner. In this picture, it shows the 2 cases when player 1 wins and when player 2 wins to see what MARS will announce when they happen.*

THIS FLOWCHART SUMMARIZES HOW HITTING-TARGET STAGE WORKS



Picture 16: *This picture shows how the program would process when the player want to hit a target*

CHECK-WINNER

In this part, I will discuss about how I decide the winner during the process of the game. As in the description of the Assignment, it has included a hint that can be understood and summarize like this: whenever we hit the target, change that target value from '1' to '0', then check the winner is easier for just a loop to check if the board of a player containing any '1' value left or not.

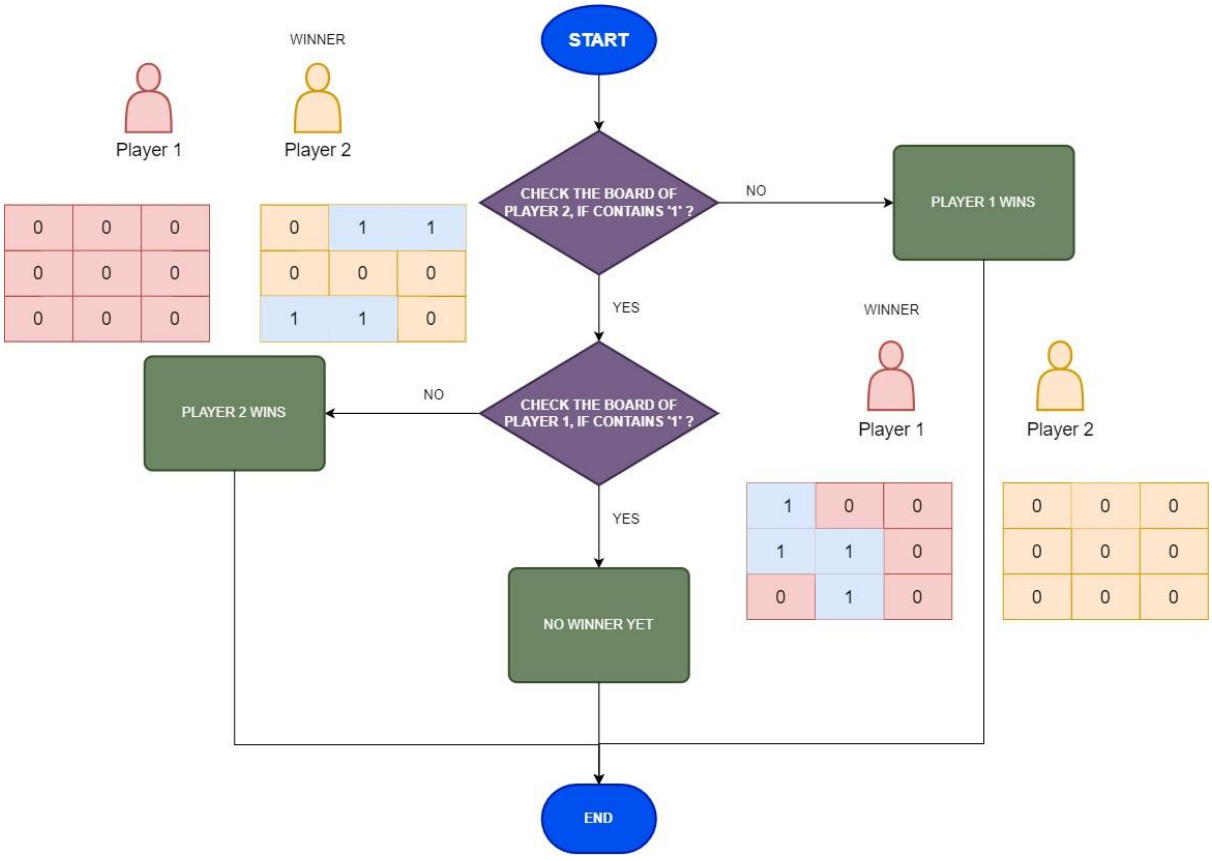
- If it still exists some value '1's, it means that the other player is not the winner.

- Otherwise there is no value '1' on the player board, it means that the other player wins the game (other player here means that if the player that we check the board is player 1, then the other player here means player 2).

If the other player has not won yet, check their board similarly to figure out if the player is the winner.

If both player have their boards still contain some value '1's on, it means that there is no winner yet at the recent time, then let's they continue to perform hit on other until we find the winner.

The above logic can be summarized and describe using the flowchart below:



Picture 17:

This picture shows how the game would check for if there is an existence of the winner or there is no winner the recent time.

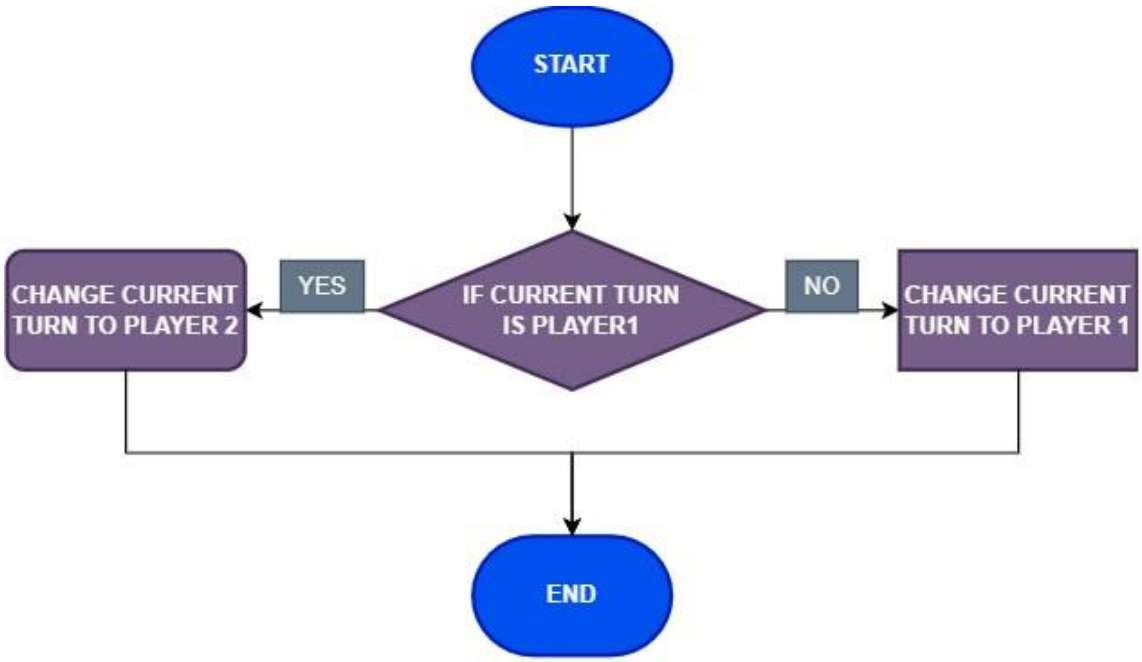
SWAP TURN

In my code, I also want to take care of whose turn it is so I might use a variable called `whose_turn` to control this. The logic of this function is quite simple:

- If the current turn is of player 1, change it to player 2.

- Otherwise, change it to player 1.

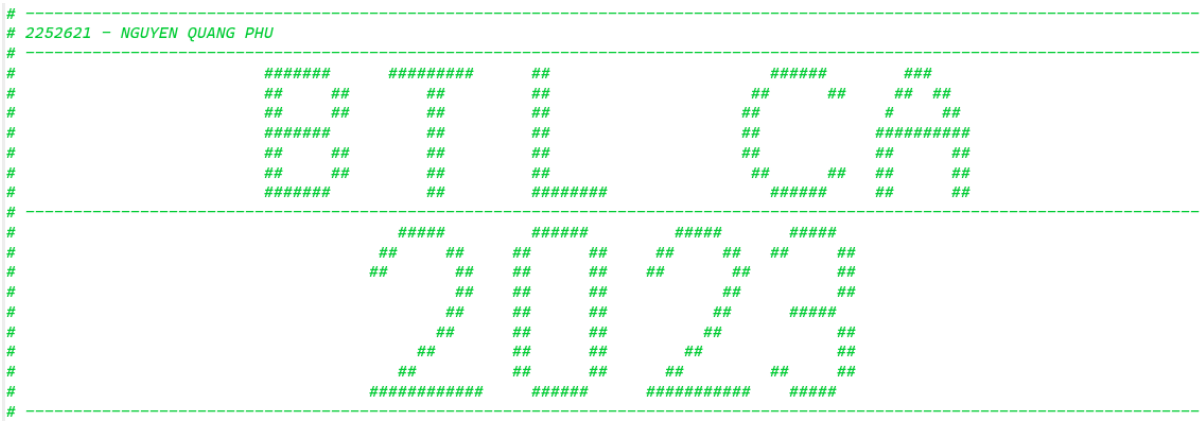
By its simplicity, the flowchart which shows how the logic would go quite simple too:



Picture 18: *The picture shows how the function using for swap turn works*

HOW I ORGANIZE MY DATA IN MARS

The first thing, not necessary, is that I have noted at the beginning at my coding mips file for battleship assignment like below. As I really respect the time I spent to do this assignment, so I think I need to make it as clear, as beautiful as I can.



Picture 19: *The beginning (not necessary) of my coding battleship file*

Then in the data declaration part, I have written out some strings that I might use to make MARS announce what happens with the players' inputs, some other strings I used to separate different stages, for example at the beginning and the end of the inputting stage of 2 players, I might use a string called divide1 to separate it with the next stage, which is the hitting-target stage of the 2 players. Here are the strings I have used.

```
# string output
divide1:                    .asciiz     "##########################################################################\n"
divide2:                    .asciiz     "--------------------------------------------------------------------------\n"
divide3:                    .asciiz     "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n"
xuongdong:                  .asciiz     "\n"
outputGameStart:            .asciiz     "|                          THE GAME STARTS!                            |\n"
outputGameEnd:              .asciiz     "|                           THE GAME ENDS!                             |\n"

outputPlayer1Board:         .asciiz     "This is player 1 board.\n"
outputPlayer2Board:         .asciiz     "This is player 2 board.\n"

startInitializeBoard:       .asciiz     "|                       START PLACE YOUR SHIP.                        |\n"
endInitializeBoard:         .asciiz     "|                        END PLACE YOUR SHIP.                         |\n"
player1InputBoard:          .asciiz     "#               Hello player 1, please place your ship!               #\n"
player2InputBoard:          .asciiz     "#               Hello player 2, please place your ship!               #\n"
inputRowOfHead:             .asciiz     "Input the row of head: "
inputColOfHead:             .asciiz     "Input the column of head: "
inputRowOfTail:             .asciiz     "Input the row of tail: "
inputColOfTail:             .asciiz     "Input the column of tail: "

inputOverlap:               .asciiz     "The ship you input is overlapping. Please reinput.\n"
shipInvalidSize:            .asciiz     "The ship you input has invalid size. Please reinput.\n"
shipInvalidDirection:       .asciiz     "The ship you input must be horizontal or vertical. Please reinput.\n"
inputPlaceSize2:            .asciiz     "Success place ship of size 2.\n"
inputPlaceSize3:            .asciiz     "Success place ship of size 3.\n"
inputPlaceSize4:            .asciiz     "Success place ship of size 4.\n"
outOfSize2:                 .asciiz     "There is no more ship of size 2! Please reinput.\n"
outOfSize3:                 .asciiz     "There is no more ship of size 3! Please reinput.\n"
outOfSize4:                 .asciiz     "There is no more ship of size 4! Please reinput.\n"

player1Play:                .asciiz     "|                         PLAYER 1 TAKES TURN.                        |\n"
player2Play:                .asciiz     "|                         PLAYER 2 TAKES TURN.                        |\n"
enterShot:                  .asciiz     "Enter the position for your shot.\n"
enterRow:                   .asciiz     "Enter the row for your shot: "
enterCol:                   .asciiz     "Enter the column for your shot: "
```

Picture 20: *This picture shows some strings I used in my code to make MARS announce things happen during the process*

```
hitBefore:              .asciiz     "You have hit this before. Please reinput.\n"
hitSuccess:             .asciiz     "HIT!\n"
hitFail:                .asciiz     "MISS!\n"

player1Win:             .asciiz     "#                                      PLAYER 1 WINS!
player2Win:             .asciiz     "#                                      PLAYER 2 WINS!

strDoubleSpace:         .asciiz     "  "
strSpace:               .asciiz     " "

# ADDING EXTRA THINGS FOR EASIER CHECK WHEN INPUT THE COORDINATE !!!!!!!!!!
strRemain2:             .asciiz     "The remaining ship of size 2 = "
strRemain3:             .asciiz     "size 3 = "
strRemain4:             .asciiz     "size 4 = "
strSemiSpace:           .asciiz     "; "

# ADDING NEW THINGS FOR INPUTTING A STRING AS COORDINATES
strInput:               .asciiz     "Please input a string of coordinates (<row_head> <col_head> <row_tail> <col_tail>): "
.align 2
buffer:                 .space      100
strError:               .asciiz     "Invalid input! Please input again!\n"
```

Picture 21: *This is the continuous part of Picture 17*

```
# Specify whose turn to play (0: player 1, 1: player 2)
whose_turn:             .word       0
.align 2
# THE BOARD OF 2 PLAYERS
player1Board:           .byte       '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'
.align 2
player2Board:           .byte       '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'
.align 2
# THE VIEW OF THEM
player1View:            .byte       '-', '-', '-', '-', '-', '-', '-',
                                    '-', '-', '-', '-', '-', '-', '-',
                                    '-', '-', '-', '-', '-', '-', '-',
                                    '-', '-', '-', '-', '-', '-', '-',
                                    '-', '-', '-', '-', '-', '-', '-',
                                    '-', '-', '-', '-', '-', '-', '-',
                                    '-', '-', '-', '-', '-', '-', '-'
.align 2
player2View:            .byte       '-', '-', '-', '-', '-', '-', '-',
                                    '-', '-', '-', '-', '-', '-', '-',
                                    '-', '-', '-', '-', '-', '-', '-',
                                    '-', '-', '-', '-', '-', '-', '-',
                                    '-', '-', '-', '-', '-', '-', '-',
                                    '-', '-', '-', '-', '-', '-', '-',
                                    '-', '-', '-', '-', '-', '-', '-'
```

Picture 22*: This picture shows the declaration for the variable* ***whose_turn*** *and the 4 initialized boards*

Inside the data declaration part, I also have to declare 3 more things: `whose_turn` variable, 2 boards of the 2 players, and lastly are the 2 view boards of 2 players. The reason why there are 4 boards is that I want to keep track of their hitting-target shots on a different board, which is different from their own boards. The next reason is that inside the `performHit` function, whenever there is a HIT target, it will force an '1' on the opponent board to be '0'. So there is no way for us to keep track of if we have hit that target already before or not, or we cannot show the player where they have hit since then (although we have tackle with the situation that the player will hit the target again by using another called view board, but I still want the user to have a thing to look at when they want to decide where to hit on the opponent board).

For the reason why I have a `whose_turn` here it is because I might use it to know the current turn belongs to which user, and this could be changed easily inside the swap turn function by just loading the address, loading the value at that address and then change it and storing it back. Here are how these things appear in my code.

WRITING PLAYERS' MOVES TO A SEPARATE FILE

There are some SYSCALL functions available in MARS for opening, writing and closing a file. These following pictures demonstrate the way to declare a name for the output file, way to open a file, way to write to the file and the way to close the file

```
# ADDING OUTSOURCE A FILE
file_out:                       .asciiz         "CA_output.txt"
```

Picture 23: *This picture shows how to declare a name for a txt file in the data declaration part.*

```
#################################################################
# Open (for writing) a file that does not exist
#################################################################
li   $v0, 13       # system call for open file
la   $a0, file_out # output file name
li   $a1, 1        # Open for writing (flags are 0: read, 1: write)
li   $a2, 0        # mode is ignored
syscall            # open a file (file descriptor returned in $v0)
move $a3, $v0      # save the file descriptor
#################################################################
# Write to file just opened
li   $v0, 15       # system call for write to file
move $a0, $a3      # file descriptor
la   $a1, divide1   # address of buffer from which to write
li   $a2, 106      # hardcoded buffer length
syscall            # write to file
#################################################################
```

Picture 24: *This picture shows how to open and write to a txt file using SYSCALL function with v0 equals to 13 and 15 (for opening and writing to a file respectively) (we have demonstrate in picture 21)*

```
#################################################################
# Close the file
li   $v0, 16       # system call for close file
move $a0, $a3      # file descriptor to close
syscall            # close file
#################################################################
```

Picture 25: *This picture shows how to close a file using the SYSCALL function with v0 equals to 16.*

CA_output.txt - Notepad

File Edit Format View Help

```
#####################################################################################################
|                                       START PLACE YOUR SHIP.                                     |
#####################################################################################################
#                               Hello player 1, please place your ship!                            #
#####################################################################################################
This is player 1 board.
  --- --- --- --- --- --- ---
 | 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 |
  --- --- --- --- --- --- ---
Please input a string of coordinates (<row_head> <col_head> <row_tail> <col_tail>): 0 3 0 4
-----------------------------------------------------------------------------------------------------
Success place ship of size 2.
-----------------------------------------------------------------------------------------------------
This is player 1 board.
  --- --- --- --- --- --- ---
 | 0 | 0 | 0 | 1 | 1 | 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 |
  --- --- --- --- --- --- ---
Please input a string of coordinates (<row_head> <col_head> <row_tail> <col_tail>): 1 0 4 0
-----------------------------------------------------------------------------------------------------
Success place ship of size 4.
-----------------------------------------------------------------------------------------------------
```

Picture 26: *This picture shows how the output txt file would look like.*

III. CONCLUSION

This is the end of my report for the battleship game of the Assignment of Computer Architecture Lab (CO2008). I find this assignment interesting and very helpful as I have a chance to think of how the game would work, how to design some procedures to help me control the game better, how to tackle with those wrong inputs from the players, how to consider all the cases that can happen during the game and last but most important, doing this Assignment help me to remember much longer, to revise all the knowledge that I have learnt on the class.

I hope that my report satisfies all the needs for the Assignment, also, contains much information, easy to understand and follow by reading the information I wrote and looking at some example figures, which I dedicate much time to design to help the reader understand my report clearer.

Thanks for reading.

Phu.

Nguyễn Quang Phú – 2252621.