VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY

Computer Architecture

ASSIGNMENT REPORT

Student: Truong Hai Nam
Lecturer:
Mr. Pham Quoc Cuong
Mr. Bang Ngoc Bao Tam
Class: CC01
Semester: 231

I) Data

The data section for my assignment is as followed:

```
.data

        HIT: .asciiz "\nHIT! \n"
        PREDEFINED_SHIP: .asciiz "Predefined ship numbers and sizes:\n1 4x1 ships\n2 3x1 ships\n3 2x1 ships"
        GETLINE: .asciiz "\n"
        P1_INPUT_4x1: .asciiz "\nPlayer 1 input 4x1 ship coordinates: \n"
        P1_INPUT_3x1_1: .asciiz "\nPlayer 1 input first 3x1 ship coordinates: \n"
        P1_INPUT_3x1_2: .asciiz "\nPlayer 1 input second 3x1 ship coordinates: \n"
        P1_INPUT_2x1_1: .asciiz "\nPlayer 1 input first 2x1 ship coordinates: \n"
        P1_INPUT_2x1_2: .asciiz "\nPlayer 1 input second 2x1 ship coordinates: \n"
        P1_INPUT_2x1_3: .asciiz "\nPlayer 1 input third 2x1 ship coordinates: \n"
        P1_ATTACK: "\nPlayer 1 pick your attack coordinates: \n"
        P1_WIN: "\nPlayer 1 won!\n"
        P2_INPUT_4x1: .asciiz "\n\nPlayer 2 input 4x1 ship coordinates: \n"
        P2_INPUT_3x1_1: .asciiz "\nPlayer 2 input first 3x1 ship coordinates: \n"
        P2_INPUT_3x1_2: .asciiz "\nPlayer 2 input second 3x1 ship coordinates: \n"
        P2_INPUT_2x1_1: .asciiz "\nPlayer 2 input first 2x1 ship coordinates: \n"
        P2_INPUT_2x1_2: .asciiz "\nPlayer 2 input second 2x1 ship coordinates: \n"
        P2_INPUT_2x1_3: .asciiz "\nPlayer 2 input third 2x1 ship coordinates: \n"
        P2_ATTACK: "\nPlayer 2 pick your attack coordinates: \n"
        P2_WIN: "\nPlayer 2 won!\n"
        B1_ANNOUNCE: .asciiz "\nThis is board 1: \n"
        B2_ANNOUNCE: .asciiz "\nThis is board 2: \n"
        P2_BOARD: .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'

        P1_BOARD: .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'
```

II) Functions

For this assignment, I have implemented 2 helper functions ‘checkBoard’ and ‘printBoard’.

Function ‘checkBoard’ essentially checks the board of the other player if it’s empty after a “shot” has been fired. This function takes the address of the targeted board as an argument and pass it into parameter register ‘$s0’. Inside this function, there will be a boolean register ‘$t3’ and it will take the value ‘1’ or ‘0’ (integer) depending on the “emptiness” of the board. ‘$t3’ is ‘1’ initially. Each element (character data type stored in byte) in of the board is checked via loop, if it comes a cross a ‘1’ value, it will subtract 1 from ‘$t3’, turning it into ‘0’ and quit the loop.

Else, it will keep checking other elements. Finally, the value in ‘$t3’ is stored into ‘$v0’ for returning.

Function ‘printBoard’ takes the address of the board stored in ‘$s0’ as an argument. This function doesn’t return any value but print out the 7x7 board. A loop is used for printing cells and there’s also a checking variable in the loop to check if it’s reached the seventh element. The checking register ‘$t2’ is initially 0 and it will count up along with the counting register ‘$t0’ for printing out cells. When it reaches 7, a branch instruction will allow the program to jump down a line and reset ‘$t2’ back to 0. ‘$t0’ will keep counting to 48 and ‘$t2’ starts over at 0 every time the program jumps down a line. The process is repeated till the whole 7x7 board is displayed in the terminal.

III) Validity check from user input

Input 4x1 ship → Check 4x1 ship validity → Store 4x1 ship into the board → Input 1st 3x1 ship → Check 3x1 ship validity → Store 3x1 ship into the board → Input 2nd 3x1 ship → Check 3x1 ship validity → Store 3x1 ship into the board → Input 1st 2x1 ship → Check 2x1 ship validity → Store 2x1 ship into the board → Input 2nd 2x1 ship → Check 2x1 ship validity → Store 2x1 ship into the board → Input 3rd 2x1 ship → Check 2x1 ship validity → Store 2x1 ship into the board

The algorithm for both players will be the same with Player 1 going through the above process first. The players will input the coordinates for the 4x1 ship, 2 3x1 ships and 3 2x1 ships, respectively. Each time the player inputs coordinates (row or col of one end of a ship), the program will check the validity of the coordinates, whether they’re out of bound (< 0 or >=7), diagonal (for the second end of the ship), or on an occupied cell, etc.

In particular, at the beginning the program displays the empty board of a player with all cells storing ‘0’. After that it enters a loop for inputing the coordinates of the 4x1 ship. One by one, the player inputs the coordinates (row, col) of each end of the ship (‘$s0’, ‘$s1’, ‘$s2’, ‘$s3’ for row, col, row, col, respectively). Every time the player input a row or col, the program checks if it’s out of bound. If so, it will loop back and request the player to reinput coordinates from the beginning.

Assuming the bound condition is satisfied, the program branches out to check the length condition. The length condition is checked when the rows of both ends or the cols of both ends have the same value ($s0 = $s2 or $s1 = s3). This means that the ship is now vertical or horizontal. The main purpose of the length condition here is to check if the ship's length is greater or less than 4. Here are the 2 cases:

First case: $s0 = $s2, the ship is horizontal. For this, the program will subtract 3 from $s1 and store the value in '$t0'. From here, it checks if $s3 is equal to $t0. If so, it means that the ship has correct length and the second end is to the left of the first end. If $s3 is not equal to $t0, the subtraction won't happen and $t0 would now store the sum of $s1 and 3 instead. From here, the program checks if $s3 is equal to $t0. If so, the ship has correct length and the second end is to the right of the first end.

Second case: $s1 = $s3, the ship is vertical. For this, the program will bypass the first-case branch condition to this branch condition right below it code-wise. It will then perform similar actions of subtracting and adding to check for the ship's length in two cases where the second end is the upper end or lower end.

In the case where all four inputs are different from each other, the program will jump back to the beginning where the player inputs the coordinates and the player will have to input again.

Note that with this implementation diagonal check is also included. From here, the program can move on to storing the ship onto the board.

The validity check implementation for 3x1 and 2x1 ships is relatively similar to this with a minor difference where instead of subtracting or adding 3 for length check, the program will subtract or add 2 for 3x1 ships' coordinates and 1 for 2x1 ships coordinates.

IV) Storing

1/ Storing for 4x1 ship

The first thing the program does here is load the address of the board to $t0. After finding both ends of the ship the program will be able to use them along with the address in $t0 to traverse the ship and store values appropriately.

First case: $s0 = $s2, the ship is horizontal. The program checks if the second end is to the left or to the right then do the following:

- Multiply \$s0 by 7, store the result in \$t1, add \$s1 to \$t1.

- Multiply \$s0 by 7, store the result in \$t2, add \$s3 to \$t2, add 1 to \$t2 (subtract by 1 if second end is to the left).

Second case: \$s1 = \$s3, the ship is vertical. The program checks if the second end is over or under the first end then do the following:

- Multiply \$s1 by 7, store the result in \$t1, add \$s1 to \$t1.

- Multiply \$s1 by 7, store the result in \$t2, add \$s3 to \$t2, add 7 to \$t2 (subtract by 7 if second end is over the first end).

By doing this, we acquire both ends of our 4x1 ship memory-wise. We need only traverse from the first end to the second end using loop (by ± 1 for horizontal and by ± 7 for vertical) and store '1' to each cell on the path and we can proceed to the next step – inputting the first 3x1 ship and checking its validity.

2/ Storing for 3x1 and 2x1 ships

The storing procedure for 3x1 and 2x1 ships are more complex due to some of the cells having been occupied by the previous 4x1 ship. Therefore, the program checks cell by cell whether one has been occupied. If so, it jumps back to the beginning for the player to input the coordinates for the 3x1 ship again.

For simplicity, I will stick to explaining the algorithm only. Basically, at first we do the same thing we did for 4x1 ship to find both ends of the 3x1 ship. Next, using the address and the first end, the program check cell by cell if one is occupied. By using multiple branch instructions, it can either jump right back to the inputting step or keep checking. Finally, if the path from the first end to the second end is unoccupied, it will perform the storing procedure similar to that for 4x1 ship.

Note that in checking occupancy, the number of branch instructions are different for 3x1 ships and 2x1 ships. Also, for vertical and horizontal cases the looping procedure is also different in terms of distance between indexes. The printBoard function is also called appropriately everytime after the program has successfully stored the ships.

V) Playing the game

The gameplay would be contained in a big loop with the return value of checkBoard = 1 as the break condition. Moreover, this function is called every time after a player hits a ship on other player's board.

First, Player 1 will input the coordinates to fire a shot. There are also check conditions to ensure that the targeted cell is in bound. These bound conditions are implemented similarly to those in the input procedure to ask repeatedly for the player to reinput if their coordinates are out of bound.

When the coordinates are valid, the program multiply the row coordinate by 7 and add to it the col coordinate, the result is stored in a register. It then loads the address of Player 2's board to another register. Using these 2 registers, the program retrieve the value of the cell. If the value is '0', the program goes to Player 2 and proceeds to perform the same actions. If the value is '1', that means Player 1 has hit a ship on Player 2's board. It prints out to the screen "HIT!" to announce to Player 1 and stores new value '0' into that cell on Player 2's board. The program then calls function checkBoard and pass the address of Player 2's board into it. If the return value of checkBoard is '1', that means Player 2's board is empty and Player 1 has won. The program quits the loop and announces the winner. If the return value of checkBoard is '0', the program moves on to Player 2's turn. The program will perform the same actions in Player 2's turn. Both players will take turn attacking the other until victory is achieved.