**Tran Anh Khoa Huynh  
TXH220015**

**Project Report**

**Tic -Tac- Toe Project Report**

Tic-Tac-Toe is not a very challenging game for human beings. Movements on the board are not necessarily mathematically calculated; one just looks at the board and knows exactly which cell of the board they want to pick. On the other hand, computers are not sentient, thus such decisions as which board cell to pick in a tic-tac-toe game require some mathematical instruction.

**I. Program description**

Tic-Tac-Toe is a game where 2 players take turns drawing symbols (X and O) into a 3x3 board until the same symbol matches across 3 rows or 3 columns or 2 diagonals. In the event of a tie no such combination has been achieved.

We numbered our tic-tac-toe board with numbers 1-9 to represent each blank cell. Cells 1,3,7,9 are the corner cells, 2,4,6,8 are the side cells, and 5 is the middle cell (side note: maybe draw a little icon instead of this). We use the word position to refer to one of these numbered squares as they are gradually filled with Xs and Os.

The first player to make a move is the computer. We have programmed the computer move to pick a random empty cell in which to place its X.

The second to make a move is the human player. When prompted, they enter a cell number in which they want to put their symbol (in our program: O). This digit will then serve as an index where we place the symbol.

We continue to go back and forth between computer and player moves until there is a win or until there is tie.

**II. Methodology**

.data:

In order to print out the board and display outputs to the user, there are an abundance of strings defined in the .data section of the program. Most notable are the variables “row1”, “row2”,  and “row3,” due to the program’s heavy reliance on them to display the board to the user. Additionally, in the section with initialized data, there are two arrays containing 32-bit words: the “board” variable and the “win” variable. The board array holds 9 words each initialized to the value of 0; the board array represents the board’s given state throughout the state. As the game progresses, the zeroes within the array will change value to mark the index of the array as “taken” by either the player or the computer. The win array holds all the possible win combinations.

Main:

Main itself is a relatively concise matter. Here, we call functions “printInstructions” and “initializeBoard” right in the beginning of main. Those two functions serve to prepare the board for a Tic-Tac-Toe match. After the two functions are called in main, a counter, stored in $s0, set to 0 is set. This counter used is “loop,” which cycles through multiple turns of the game. The counter is set to 0 and incremented each time the loop finishes its tasks: pushing 3 elements to stack (the register address, the counter, and the maximum number of turns), calling functions playerMove and computer Turn, printing the table update, popping the 3 elements from stack, and then incrementing the loop counter. It’s important to note that in each iteration of the loop, both the playerMove and computerTurn are called; when the loop counter equals four, the loop jumps to a branch called “lastTurn, ” which calls playerMove, printTable, and checkBoardWin. If checkBoardWin does not determine a winner, then the lastTurn branch notifies the player of a tie before prompting the player for a new game.

Function – printInstructions

Location called: Main

Purpose: This function exists to print the instructions and initial board placement to the user.

How it works: This function first pushes the register address to the stack before loading a welcome message as well as an explanation of the game. Then it proceeds to print out the board through strings. When it’s done printing, it restores the stack parameter and adds back to the stack before jumping to where the register address saved.

Function – initializeBoard

Location called: Main

Purpose: This function exists to load the board.

How it works: It simply loads variable “board” to $v0 before jumping back to where it was called.

Function – playerMove:

Location called: “loop” in main

Purpose: The function prompts the user for the desired index on the board. Then, that index is passed to boardUpdate. This function essentially takes care of all the tasks needed to be done for the player’s move.

How it works: First, the function loads the board array address. Then it adds the register address to stack before prompting and intaking the desired board index. The function moves the desired board index into $a1. After getting the desired board index and moving it, the function checks if the value of the index is from 1 – 9 through a series of blt and bgt branch conditions. If the index is less than one or more than 9, then the function is called again, prompting for a new index value. After checking that the index is within the suitable range, the function subtracts the index value by one so it lines up with the board array. Then we move the index from $a1 to $a3. After storing the input in $a3, we multiply the index by 2 and store it in $t3 in order to convert it properly to add it to the array address of the board. After adding the offset with the board address in $t3, the function checks if the current position is empty through a bne branch condition: if the value of the random position is not equal to 0, then call playerMove again, else, continue with throughout the function. Next in the function, the function loads the value “1” into $a0 to act as a flag for distinguishing between the user and the computer. The value “1” represents the user while the value “0” represents the computer. After setting the flag variable to 1, the function calls boardUpdate, and then pops the register address from the stack before jumping back to where the function was called.

Function – computerTurn:

Location called: “loop” in main    
Purpose: The function automates the computer choosing a random value against the player.

How it works: The function loads the array address of the board to account for changes made within the array before generating a random number through li $v0, 42 and li $a1,9. The instruction with the “42” informs the simulator that a random number call is behind made while the line with “9” tells the simulator to create a random number with a range from 0-8. The random number is stored in $a1 before it is multiplied by two and added to the current memory location of the array. After, the function increments $s4, which serves as a counter for how many times the computer function will run in one turn in case it keeps calling chosen numbers already: aiding the counter to make sure data isn’t overwritten are branch conditions including loop “NoSpace” and “computerTurn.” If the index is already chosen, the function will simply call itself again. In the case that the index has not been chosen yet, it moves the chosen value from $a1 into $a3. The function then changes the flag distinguishing user and pc to 2 before calling boardUpdate.

Function – boardUpdate:

Location called: playerMove & computerTurn

Purpose: Here, the function takes the values of the indexes in the board array from playerMove and computerTurn and updates the board to reflect the changes.

How it works: The function reads $a0, which is the register that represents the user/computer flag. If the flag is 1, it is a user’s input, and the program jumps to branch condition “userChange”; if it’s 2, then it is the computer’s input, and the program jumps to branch condition “computerChange”. Within computerChange and userChange, the program saves the value representing the respective player (user(1)/computer(2)) into the board array before returning back to where boardUpdate was called.

Function – printTable:

Location called: “loop” in main

Purpose: This function loads the address of the table, and if the board array equals “0”, then it’ll just print the space character. If it equals “1”, then it will print out “O” to symbolize the user. If it equals “2”, it will print out “X” to symbolize the computer.

How it works: The function first saves the return address to stack before loading the address of the board. It prints the table through a loop, and keeps count of all the places that have been printed out using a counter initialized outside of the array.

Function – checkBoardWinCondition:

Location called: “loop” in main

Purpose: The code checks to see if there is a winner by checking for a winning row, column, and diagonal befire returning a value indicating if either party won.

How it works: The function uses multiple functions to check whether there is a winning condition. “checkRow” works by loading the address of the board and taking each value at an index and finding the sum. If the sum of the values equal 3, then the user wins; if the sum equals 6, then the computer has won. If neither conditions are met, it moves on the next row and repeats the process. A similar logic is applied to “checkColumn” and “checkDiagonal. ”

Branch Condition – userWin:

Location called: checkBoardWinCondition

Purpose: This function executes in the event that the player wins. It’ll display the board and prompt the user for if they want to play the game again and depending on the user input, it’ll restart or exit the game.

How it works: The function calls function printTable and does a system call for “pwin,” a variable that is meant to be displayed in the event of a player victory. The function then does another system call for “playAgain,” which prompts the user if they want to play the game again. Another system call is initiated in order to collect the user’s choice. Two branch conditions are prepared. If the user choice is equivalent to 1, then the player has chosen to restart the game, which makes the branch condition jump to a different branch: restartGame. If “2” is selected, then the program jumps to “exitGame,” where the branch condition terminates the game.

Branch Condition – computerWin:

Location called: checkBoardWinCondition

Purpose: This function executes in the event that the computer wins. It’ll display a message informing the player that they have lost, and it asks the user if they want to play the game again: exiting or restarting the game depending on the input.

How it works: This function works by first displaying the board of the game by calling printTable. It displays the variable “cwin,” which holds the message in the event that the computer wins. After prompting the user for if they want to play the game again, it collects the answer with “1” being yes and “2” being no. If the player says yes, then the branch jumps to loop “restartGame.” If the player says no, the branch jumps to “exitGame”

Branch Condition – restartGame:

Called in: computerWin & playerWin

Purpose: This function is meant to completely clear the board array to zero before jumping back to main and restarting the game.

How it works: First, the program loads the board array’s address and sets a counter to zero. Inside a loop “loop\_restartGame,” it checks if the counter has reached 9 before exiting out of the loop and jumping back to main. However, if the counter is still less than 9, then it will calculate the address of the current board by multiplying the counter by 4 and adding it to the current board array address. After updating the array address, it sets the current board element to 0 before incrementing the counter and looping again.

Branch Condition – exitGame:

Called in: computerWin & playerWin

Purpose: This function is called in the event the player wants to exit the game. It simply exits the program through a system call.

**III. Team challenges**

 Some of the challenges we faced as a team was on deciding how to approach this project and begin coding. There were many ways to implement each cell of the board and we decided to go with an array and use flags to represent the board symbols.

Another challenge was splitting the work for this project evenly so that each member contributed equally to the project, and no one was left out. It took multiple meetings and a lot of collaboration between us to get to the final product, but this way we were able to build on our communication skills and collaborate to code our respective assigned parts.

**IV. Personal challenges and lessons** Throughout the tic tac toe project, I encountered various challenges that helped me grow as a programmer and team member. One particular challenge was the need to wait for other code sections to finish before executing my own code, which slowed down progress. Furthermore, the requirement to not change primary variables made it difficult to solve my assigned tasks. To address these challenges, I used code snippets with assigned table arrays and variables to test my algorithm and ensure that it worked properly before integrating it into the main code. This approach allowed me to make progress without waiting for others' code and saved time during the debugging process.  
 From this project, I learned several valuable lessons about working in a team. One important lesson is to establish specific deadlines for all team members to meet and avoid procrastination to ensure that the project stays on track. Additionally, I gained experience in scripting code that assigns necessary values to run the simulation for my code before integrating it into the main code. Finally, I also learned about branch and jal conditions and how to use them effectively in programming. Overall, these experiences taught me the importance of collaboration, problem-solving, and attention to detail in achieving project success.

**V. Peer evaluation**

**Tran Anh Khoa Huynh:** I contributed to the project by wrote a computer move function that randomly selects a move, as well as a print-table function that allows the program to keep track of both the player's and computer's movements. Additionally, I spent time rewriting and fixing Ariel's main, user move, and updated board functions, as I discovered that the condition to check return variables of Ariel's code was incorrect and needed correction. This required me to meticulously debug each line of code to identify and fix the problems. Finally, I also completed the Play Again function, which enables the player to reset the game and start a new one, or quit the game entirely. I don't want to procrastinate, so I always get the job done and fixed in a day so I can speed things up and avoid time risks. In summary, I successfully completed all of the assigned tasks, including computer move, print table, play again functions, and made correction and necessary improvements to the main, user move, and updated board functions.  
 **Ariel Ong:** Ariel implements an outline for the basic main function, user move, and updated the board function to start this project. She also works on description of the algorithm and techniques used on the report. All functions are based on the idea that Ariel proposed to solve this program. She is very responsible for explaining in detail what primary variable ​​will be used to store value to help the program work. She also completes assigned program tasks and functions on time. Besides, she is always happy to answer any queries I have regarding her function code part, which will enable me to merge it with my own function. Overall, Ariel contributed a good work and provided much-needed feedback.

**Isindi Cela**: Insidi is the person who participated in writing the code print Instruction function that prints the beginning instructions for the players in this project. She also wrote a description of this program, a user manual, and made a video for the project. Besides, she is active in testing the game to find the bugs that need to be fixed and what needs to be improved to make the game better. She also completes the work within the deadline set by the group, and she also actively contributes ideas when participating in group meetings. Overall, Insidi has done every task on time and given the necessary feedback for the program.  
  
Lawson Lay: Lawson is the one who wrote the conditions to decide the winner. He also fixed some bugs and improved the concept to make the program more complete. He worked diligently and seriously when proactively completing the work earlier than expected to have time to correct and improve the errors that arise. In every meeting, he actively asks each member to present their work progress and present difficulties they are facing so that the work can be divided appropriately and set new goals. Overall, he did good on his work and success in linking and dividing all the work as well as helping us have a clear plan to complete after each meeting.

-----THE END-----