Description of Functions in Othello1.4.0.asm

**Before started**: This program, Othello, is written to have one-time play, player vs. computer. That is, as following the rule of Othello, turn is switching one another. Game over ends this program. This document mostly explains all functions in order, so it may mix up because some functions call other functions placed below. Also, explaining precisely what each line does is commented line by line in the program.

**main**: This loads a board with starting positions and begins game

**loadBoard**: Because this function is called by main and also calls other functions, address of caller has to be saved. Loading immediate value for each pixels: blue color, 0x0000ff, as black, red color, 0xff0000, as white, and green color, 0x00ff00, as empty. This draws 16 by 16 board with a 8 by 8 Othello board right in the middle, meaning first and last 4 rows and first and last columns are filled with blank, 0x00. It sets $a1 to first address of board. Before explaining each jal coming afterward, drawPixel saves element in $a2 into $a1. DrawEmptyRow saves color in $a2, saves register address of callee in $v0, and goes to DrawEmptyLoop. DrawEmptyLoop increments address of current position in board and jump to drawPixels and keeps storing value in board until it reaches last element of row. As similar to drawing pieces in DrawEmpty Row, DrawPlayRow draws 4 blanks, 8 empties, and 4 blanks. After all 16 rows are initiated, it jumps to DrawStartPieces to draw starting positions. In 16 by 16 board(8 by 8 board), index of white element is 119(44) and 136(55), and index of black element is 120(45) and 135(54). It multiplies those number by 4 to set them word size using sll 2. After drawing all starting positions, we call back callee address and jump back to the callee.

**beginGame**: this uses syscall 41 to call a random integer, and we mask first bit. Depending on the first bit, 1 lets player go first and 0 lets computer go first. Former player has blue(black) color and latter player has red(white). It sets and stores color elements into named valuable respectively. At the end, it jumps either to playerTurn if player goes first or to computerTurn if computer goes first.

---------Functions below are following a rule about arguments. $a0: address, $a1: current color, and $a2: whose turn the current is (0 as player and 1 as computer)-------------------------------------

**playerTurn**: This resets color elements again and goes to askUserInput. This sets $a1 to player color, $a2 to 0. Also this loads $s3 current color and $s4 computer color. This jumps to getValidMoves to get all valid moves in the current board condition as well as check game over condition before asking player input. If there exists at least one valid move, it jumps to askUserInputPrompt to ask user input. Return $v0 holds adjusted input number corresponding to 16 by 16 board address. Set this number to $a0 and jumps to checkInputIsValid. If the input is invalid, this function jumps to askUserInput. If the input is valid, jumps to storeTilesToFlip to get tiles to flip and jumps to flipTiles. At the end of the playerTurn, it gives a sound effect passing soundValid and switches turn to computer.

computerTurn: This resets color elements again, and sets $a1 to computer color, $a2 to 1. Also it loads $s3 current color and $s4 player color. After setting all necessary registers, this jumps to **isOnCorner** to check whether any corners are available. $v0 holds 0 if no corner is available and address of corner if the corner is available. If $v0 isn’t zero, go placeOnCorner to place a computer piece onto it and switches turn to player. Otherwise, this jumps to getValidMoves and randomlySelect to pick an address of valid move. Similar to playerTurn, once we find a valid move, it goes to storeTilesToFlip and flipTiles. Lastly, it changes turn to player.

**getValidMoves**: This checks all addresses of board if each address is valid using isValid. This return array of validMoves. This function is the most expensive function since it checks every single element whether it is valid or not by passing isValid. At start of this function, we need to initialize an array of validMoves in order to hold new valid moves corresponding to a current board. In loopValidMoves, it is very simple that take i as index of array, adjust i to 16 by 16 board via adjust0To64Board, make it word size, and pass through isValid. If this is valid, isValid returns valid address in $v0, so save it into validMoves. The most beautiful point of this function is to check game over as well. In returnValidMoves, we have counted how many elements have been placed in loopValidMoves, then if the counter reaches 64 meaning there is no empty elements. One line after that, we have checkNoValidMoves. This function checks condition 2 and 3 of game over, which are mono-color on board or both players have no valid move in a current board. $s7 is a counter for those condition and adds up 1 if changing turn because of these conditions. Once $s7 reaches 2, meaning both player have no valid move in a current board, it jumps to gameOver. $s7 is reinitiated anytime after it finds a valid move.

**askUserInputPrompt**: This asks user input; it adjusts column and row in 8 by 8 into 16 by 16. It asks columns in ASCII character and rows in numeric number. In inputToAddress, the logic to adjust input to address is at first multiply the row value by 16 since there are 16 elements per row and add 48 to the result to avoid the first 4 empty rows. Secondly, it changes the ASCII character to an integer value directly related to the rank of the letter and sum results of row and column. Finally, it adjusts it to word size and sum up with address of first element of board. It returns $v0 as the address of the input in board.

adjust0To64ToBoard: This adjusts any numbers between 0 to 63 for index of 8 by 8 board into index of 16 by 16 board.

**isValid**: This function checks whether the argument is valid or not. At first, it saves $t0, $t6, $t7 used for counter in getValidMoves. This has 2 loops: forDirection and whileDirection. forDirection checks all direction: [-1,1],[0,1],[1,1],[-1,0],[1,0],[-1,-1],[0,-1],[1,-1] corresponding to -17, -16, -15, -1, 1, 15, 16, 17 in 16 by 16 board. it takes an element of direction[i], adjust it to word size, and adds it to address of argument. Then it takes an element of the address. If the element is not the opponent tile, then back to forDirection loop to check next direction. If the element is an opponent color tile, then go to whileDirection. whileDirection keeps finding an opponent color tile either until it finds an empty then go next direction in forDirection or until it finds a current color tile. If it finds a current color tile, then $v0 holds the valid address. If this is invalid, then it returns 0 in $v0.

**storeTilesToFlip**: This function used to be constructed in isValid as storing address to flip going backward in whileDirection. However, as we have tested our program, it takes extra works to initialize tilesToFlip array each time isValid is executed. This function has been separated and been to execute only once after a valid move is found. This function uses mostly same instructions in isValid, plus initialize tileToFlip array and store address whose tiles can be flipped by current color tile as going backwards toward input address in loopStoreTilesToFlip.

**flipTiles**: This function flips tiles in addresses in tilesToFlip array. It takes each elements as increment and store the current color tile, $a1, into an address. Finally, it places the current color tile onto the input address manually.

**checkInputIsValid**: This function is to check whether a user input is valid using validMoves. The first element of validMoves array is either an address of valid move or zero. If this is zero, invalid, then go to returnInputIsValid to give a message and sound, and ask user input again. If this is valid jump back to callee.

**isOnCorner**: This function checks whether any corner is available. Indexes of corner in 16 by 16 are stored in onCorner array, so it takes each element and add it up with first address of board. Using this resultant, it checks whether the corner is valid via isValid. If any corner is available, it returns the address in $v0. Otherwise, 0 in $v0.

**placeOnCorner**: This function places a computer piece on corner, and changes turn. As always to flip tiles, it jumps to storeTilesToFlip and flipTiles.

**randomlySelect**: This function randomly chooses a valid move store in validMoves array. At first in generateRandomNumber, it needs to seeds the random number generator using a syscall 30 to take a system time and seeds from time. It randomly generates [0,5]. After the random number generated, it counts the number of elements in validMoves. Say, the random number is i and the number of elemets is j. It takes a i mod j, and the resultant k is used for the index of validMoves. In other words, this function return validMoves[k].

**gameOver**: This function is not a function stood by alone, but rather it is constructed in getValidMoes, so it ends the game responsively whenever the game has to end. There’s 3 conditions to end the game. 1)64 boxes are filled, 2)there’s no valid moves for both turn consecutively, and 3) all pieces on the board is either color. Condition 1 is mentioned above, and 2 and 3 are checked by checkNoValidMoves. If there is no valid move for current turn, it will increment a counter by 1 and change turn. Counter is always reset if there’s at least one valid move. If the counter counts 2, meaning both player and computer don’t have a valid move in a row, then the program has to end the game. This function will jump to getScore function to get scores.

**getScore**: This function checks all elements in the board after game is over. This returns the number of player tiles, number of computer tiles, and winner message.

\*soundValid, soundInvalid, soundWin, and soundLose are skipped. They are a syscall instruction which $a0 as pitch, $a1 as duration, $a2 as instrument code, $a3as volume, and $v0 as type of playing. $v0 holds 33 as playing tone and delay until tone completes or 31 as playing tone at every syscall without delay.