Matthew Finerty, Kayden Hung, Eric Jiang, Matthew Yu

Professor Cariaga

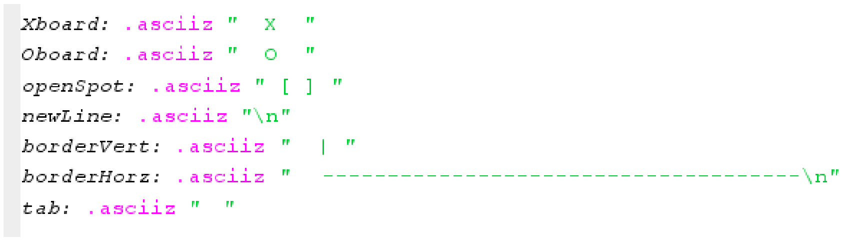
CS2640.02

14 May 2022

# Final Report Paper: Tic-Tac-Toe Project

The project that we decided to take on as a team was implementing a tic-tac-toe game through MIPs assembly. The reason we chose this specific project was because we wanted to challenge ourselves but not give us a project that seems a bit out of hand like the snake game or tetris. Another criteria was to pick a project that seemed a bit more exciting than a sorting algorithm so we decided on a simple tic-tac-toe game.Tic-tac-toe itself seemed intriguing to implement since there seemed to be a multitude of ways that it could be recreated in MIPs. The initial goal was the implementation of a tic-tac-toe game using a MIPs bitmap and updating the board visually to the user on the bitmap using an array coordinate system. We modified our goal to also use the I/O stream from the Mars MIPs IDE to allow for a backup plan in the event the bitmap implementation was not completed, but we were able to implement the bitmap display to play our game. Some objectives were to create a modifiable board, a check system to announce if the winner has met the conditions of a traditional tic-tac-toe game (3 in a row/column/diagonal), some sort of error handling for the game, and an input stream from the user to be inputted into an array. These are some abstract objectives for our total implementation of our tic-tac-toe game.

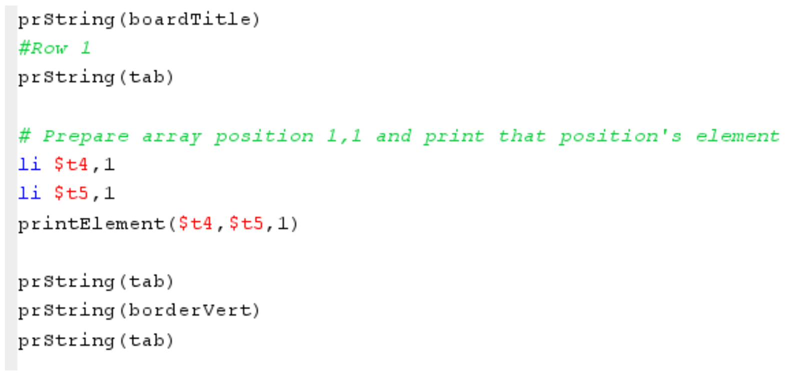
The beginnings of our implementation of the tic-tac-toe game began with understanding how a bitmap could be used to allow for a visualization of the game to the user. It was near the end of finalizing the game's code that we were successful in using the bitmap display. We have a macros section called Bitmap Macros that deal with the drawing and displaying of all the components for our game. These macros lay out a series of loading addresses and bits to build the board and allow for a connection between our already coded game in our main label. More specifically, the Bitmap Display contains macros like draw_rect_r that will draw the basic outlines of the board. Other macros like draw_X will utilize these macros to draw symbols and allow for the updating of the bitmap display. But, without having built the board using a bitmap display initially, we decided to go through the Mars MIPs IDE I/O stream to output a visualization of the board. The first challenge for the I/O stream was finding out a concrete way to implement the active changing of the board visually to the user. We first tried to find ways on how to fluidly update the game board in the I/O stream with some ideas such as making every possible combination of the board and printing it out or taking strips/rows of the board and printing the combinations. We finally landed on using components of the board which is shown from Figure 1.1 to allow us to build the board organically according to user inputs so we were not constrained to finding each combination of the board.



```
Xboard: .asciiz "  X  "
Oboard: .asciiz "  O  "
openSpot: .asciiz " [ ] "
newLine: .asciiz "\n"
borderVert: .asciiz "  | "
borderHorz: .asciiz "  -------------------------------------\n"
tab: .asciiz "  "
```

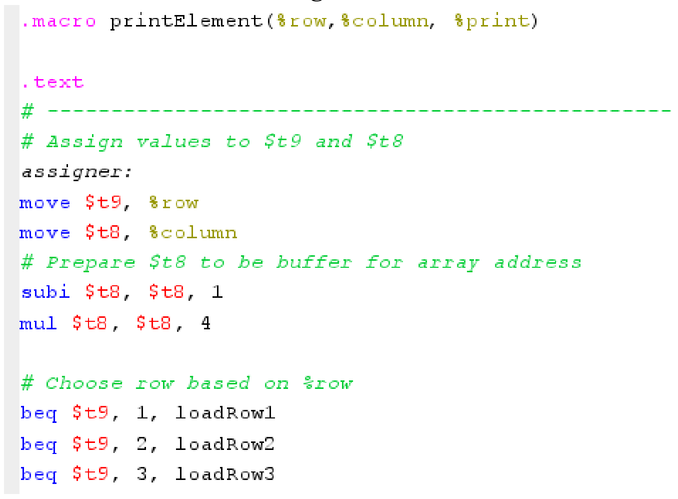
**Figure 1.1**

This way we were able to fluidly build the board through the implementation of a series of checks and buffers through the use of 3 arrays with each representing rows 1-3 and a column system that took an input and was modified to fit the constraints by subtracting 1 to be within each column which were 0, 4, and 8 then the integer was multiplied by 4 to correspond to any of the 3 so the row inputted value will be in the correct position. Rather than having a series of combinations we utilized a check for each position, for example 1, 1 (row 1, column 1) would check the array of row1 and the first element to see whether it had the pertaining value for an O or X or nothing which resulted in printing the corresponding symbol. Then we would run these checks, 9 in total, for each spot on the board through a loop that would also be in combination with the input of the user so the board would be continuously updated as shown in Figures 1.2 and 1.3.



```
prString(boardTitle)
#Row 1
prString(tab)

# Prepare array position 1,1 and print that position's element
li $t4,1
li $t5,1
printElement($t4,$t5,1)

prString(tab)
prString(borderVert)
prString(tab)
```

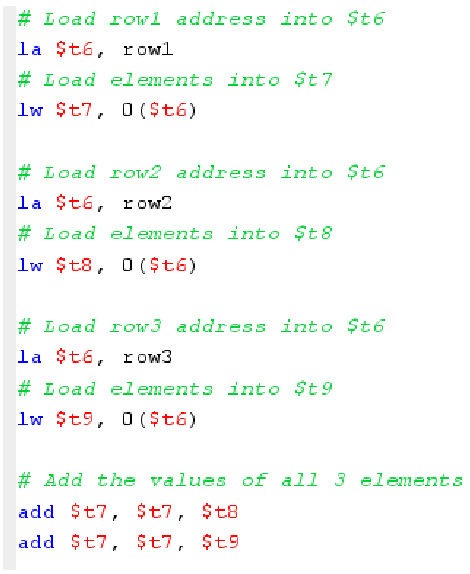
**Figure 1.2**



```
.macro printElement(%row,%column, %print)

.text
# -------------------------------------------------
# Assign values to $t9 and $t8
assigner:
move $t9, %row
move $t8, %column
# Prepare $t8 to be buffer for array address
subi $t8, $t8, 1
mul $t8, $t8, 4

# Choose row based on %row
beq $t9, 1, loadRow1
beq $t9, 2, loadRow2
beq $t9, 3, loadRow3
```

**Figure 1.3**

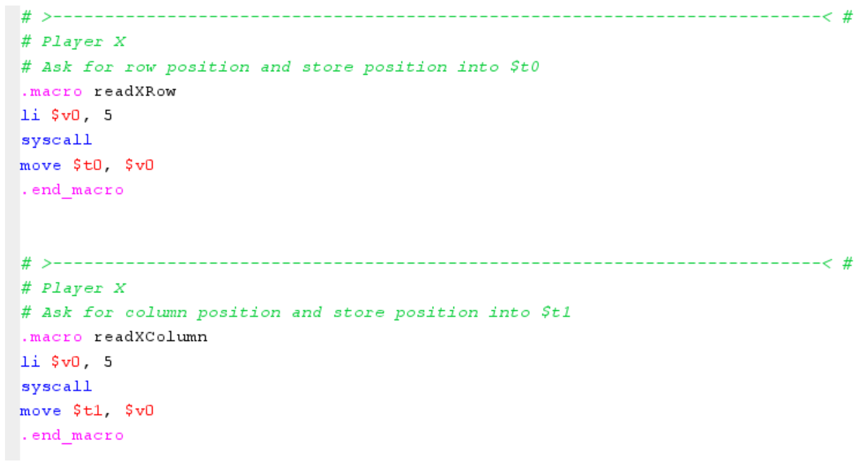
Something we could have done differently was rather than using 3 separate arrays, shown in Figure 1.4, to indicate each specific row we could have done a singular array of 9 elements like *row: .word 0, 0, 0, 0, 0, 0, 0, 0, 0* where each triplet was a row and every third would be apart of one column. This would've allowed us to load a singular address and use the corresponding position in the array when checking its value to build the board also. Although this would eliminate the repetitive calls for the array address, shown in Figure 1.4, it does make it a little more complicated in terms of calling each separate row by adding an additional call to load the address with an immediate value to direct to a specific position in the array. In the end we decided to utilize 3 arrays since it seemed more readable for us and for a reader, in our opinion, since it is easier to visualize each separate array being only a row rather than having to visualize a singular array having all the rows and columns.



```
# Load row1 address into $t6
la $t6, row1
# Load elements into $t7
lw $t7, 0($t6)

# Load row2 address into $t6
la $t6, row2
# Load elements into $t8
lw $t8, 0($t6)

# Load row3 address into $t6
la $t6, row3
# Load elements into $t9
lw $t9, 0($t6)

# Add the values of all 3 elements
add $t7, $t7, $t8
add $t7, $t7, $t9
```

**Figure 1.4**

Next, in order to take the input of a user, we asked the user for a column and row number for both X and O users, shown in Figure 2.1. We just implemented macros for the streamline use of asking for the ints where we call 4 separate macros, 2 for X and 2 for O.



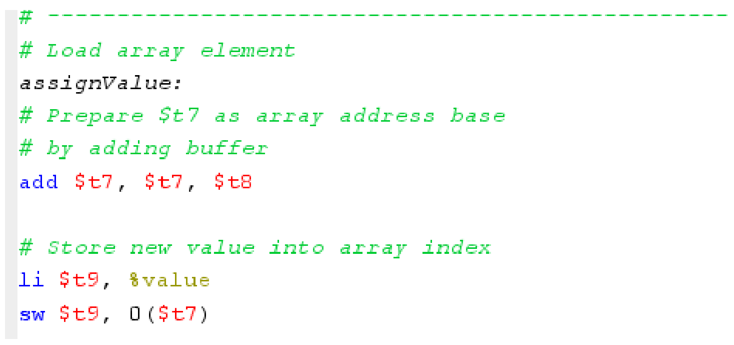
```
# >---------------------------------------------------------------------------< #
# Player X
# Ask for row position and store position into $t0
.macro readXRow
li $v0, 5
syscall
move $t0, $v0
.end_macro


# >---------------------------------------------------------------------------< #
# Player X
# Ask for column position and store position into $t1
.macro readXColumn
li $v0, 5
syscall
move $t1, $v0
.end_macro
```

**Figure 2.1**

For better efficiency for the reuse of our code we could have just had a singular macro that called for user input then moved the input into their specific registers, in this case we used register $t0,

$t1, $t2, $t3 as the inputted user's values. The registers $t0 and $t1 were for Player X's row and column inputs respectively and registers $t2 and $t3 were for Player O's row and column inputs respectively. But we eventually just stuck with the 4 macros since we had already figured out most of the implementation of the program by the time we got around to the implementation of asking for user input. Ultimately, this change would have been something that would have allowed us to maximize our reuse of code and prevent repetitive calling of different macros that all did the same thing.

The overall game requires user input in order to update the board corresponding to what position the user desires to play their symbol. Once obtaining the user's inputted values for row and columns we had a macro that would be called to use the registers and the corresponding player's value for the array to be inputted into the macro. For example, Player X would have assignValueToArray($t0, $t1, 1). This macro would then take the 3 components and similar to printing the board, the column input would be modified by subtracting 1 and multiplying 4 in order to know which position the user wanted for the column when inputting it into the array. Then taking that modified value a series of checks would be done to the row input by checking which specific row then load the specific row out of the 3 arrays for each row as shown below in Figure 3.1.

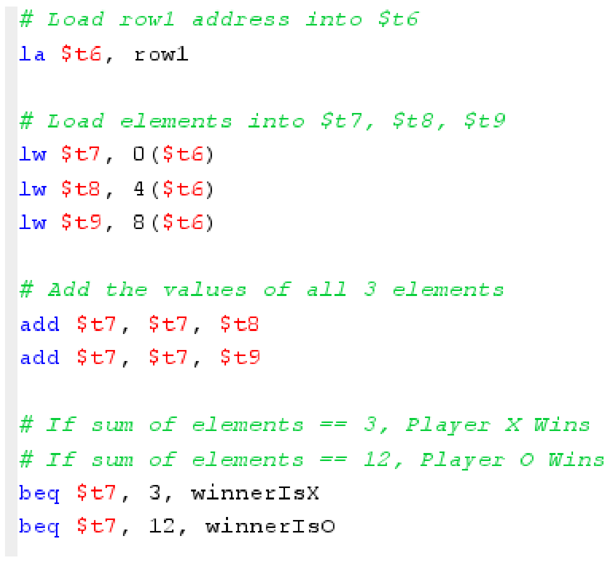


```
# ------------------------------------------------
# Load array element
assignValue:
# Prepare $t7 as array address base
# by adding buffer
add $t7, $t7, $t8

# Store new value into array index
li $t9, %value
sw $t9, 0($t7)
```

**Figure 3.1**

Then the macro would load the “word” or number into the specified array with the modified value as the buffer or what position in that array as the column placement.
This would be used in the general main label to be called every time when the user inputs their respective row and column values. But, we also kept in mind the errorhandling so we would have two different labels where the first one would make sure that the user’s inputted values were not already in use or were out of range. If the values were not duplicates or out of range then the next label would be jumped to in order to store the value into the array properly.

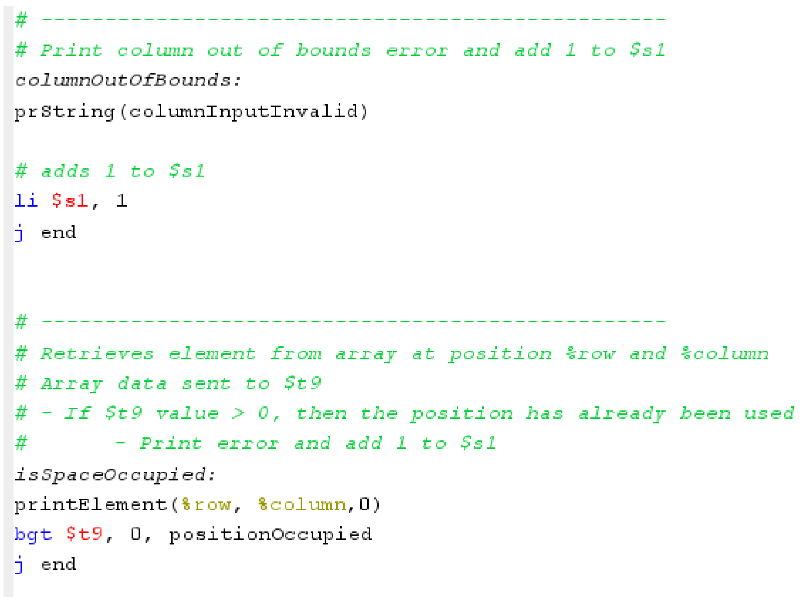
A key component to our implementation of the game would be a series of checks, which was mentioned briefly earlier, to see if either user had one. In our implementation, we had Player X represent 1 when inputting their values into the array while Player O represents 4. Thus, when we checked all the arrays from row1 to row3 we would add the corresponding values. If they added up to three, then Player X won. If they added up to 12, Player O won. We took this arithmetic and wrote each combination of how a player could have won, shown in Figure 4.1 which meant each row, column, and the two diagonals that a player could have done to win. In order to check the columns we would separately call each separate row and the diagonal position of the element in the array to see if they were added to 3 or 12. Lastly, we utilized a counter to simply call for a tie in the event that the board does not have a clear winner and this was achieved by having the counter increase after each user’s input and once it had reached 9, which is the corresponding number of positions on the board, the game will jump to read that the game was a tie.



```
# Load row1 address into $t6
la $t6, row1

# Load elements into $t7, $t8, $t9
lw $t7, 0($t6)
lw $t8, 4($t6)
lw $t9, 8($t6)

# Add the values of all 3 elements
add $t7, $t7, $t8
add $t7, $t7, $t9

# If sum of elements == 3, Player X Wins
# If sum of elements == 12, Player O Wins
beq $t7, 3, winnerIsX
beq $t7, 12, winnerIsO
```

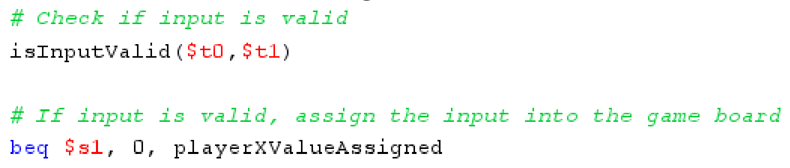
**Figure 4.1**

Then lastly we needed the potential error handling of two cases, one where the user inputs values that are out of the range of the board, and the other where one of the users attempts to input his or her position in an already filled position. We implemented this by calling a macro that would take in the positions that are inputted by the user and see if the user's inputs are valid in both cases. The checks will specifically see which input was invalid, whether it was the row or column input that was out of range, as shown in Figure 4.2. Then, the latter issue uses the same implementation as before for checking if the value was out of range but uses the macro of printing the board to see if the specific row and column already is in use. If the position is in use the macro will return a value that is greater than 0 which will then jump to another label to state that the position is taken and the user needs to input another position again. The entire macro uses a flag in register $s1 which is then used as a check to allow for proper input into the array to prevent overriding of already inputted values in a specific position in the array shown below in Figure 4.3.



```
# -------------------------------------------------
# Print column out of bounds error and add 1 to $s1
columnOutOfBounds:
prString(columnInputInvalid)

# adds 1 to $s1
li $s1, 1
j end


# -------------------------------------------------
# Retrieves element from array at position %row and %column
# Array data sent to $t9
# - If $t9 value > 0, then the position has already been used
#       - Print error and add 1 to $s1
isSpaceOccupied:
printElement(%row, %column,0)
bgt $t9, 0, positionOccupied
j end
```

**Figure 4.2**



```
# Check if input is valid
isInputValid($t0,$t1)

# If input is valid, assign the input into the game board
beq $s1, 0, playerXValueAssigned
```

**Figure 4.3**

The overall strength of our program was our deliberate use of macros to allow for repetitive calls in the logical decision making for declaring a winner to the calling of constantly updating the board for the user to know what position has been taken up. I believe that if we were given more time we could streamline and more efficiently use our code with the aforementioned changes mentioned throughout the paper. A more notable display change, if we had more time, would have been a polishing of our bitmap display to possibly show a winner box displaying which player (X or O) won or a tie screen to notify the player. But with our foundational understanding of the Assembly Language we accomplished our goals and objectives we had stated in the beginning. This project definitely helped our group grasp the repetitive nature of the Assembly language itself with the manipulation of registers being the most defining trait when writing the program.