HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Computer Architecture - CO2007
Assignment's Report

Lecturer: Phạm Quốc Cường
Semester: 212
Class: CC01
Student: Cao Minh Quang - 2052221

HO CHI MINH CITY, 11^{th} April 2022

Contents

1 Brief Analyses Before Implementing MIPS Program

1.1 Storing, Back-up And Resetting Data Of The Board

In this assignment, a 5×5 text-based board are required for the Tic-tac-toe game. I've decided to use a two dimensional array to hold our board, in which each row is a sub-array of size 5.

Our board would be an array of character. Empty entries will be indicate by the minus sign '-'. Meanwhile, 'X' and 'O' will represent player 1 and player 2's move, respectively.

The following code segment is used to allocate memory for a 5×5 array:

```
board: .word R1, R2, R3, R4, R5
R1: .word '-', '-', '-', '-', '-'
R2: .word '-', '-', '-', '-', '-'
R3: .word '-', '-', '-', '-', '-'
R4: .word '-', '-', '-', '-', '-'
R5: .word '-', '-', '-', '-', '-'
```

Figure 1: Allocating memory for 2D-array.

Furthermore, one of the requirements is that players can undo 1 move before the opponent. So, we will need another array to store the previous state in case players want to undo their choice.

The procedure for this option is not so sophisticated. After one players have made their choice, we will print out the board and then ask whether they want to undo or not.

* If they agree to undo, copy data from back-up board to the main board. Allow them to move again and update the backup board.
* Else, update the current state to the back-up board.
* Then passing the turn to another player and do the same task again.

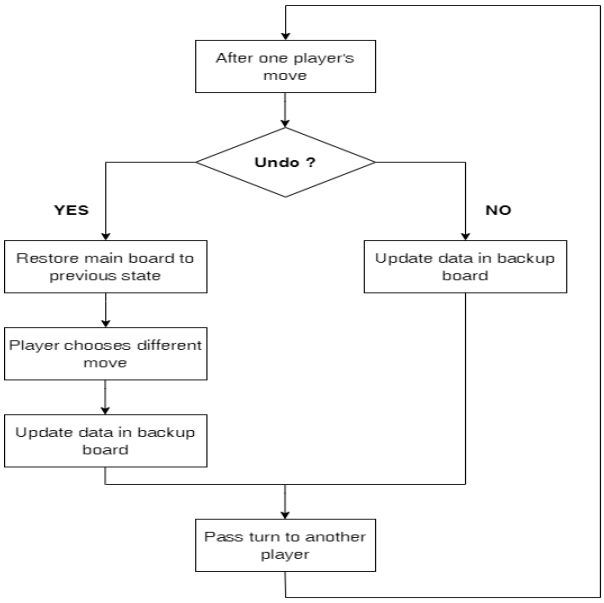

Figure 2: Undo procedure.

After finishing one game, players may want to play another one so we can simply use **j** instruction to jump to the beginning of the program.

But before performing that instruction, we need to wipe out all data in both main board and back-up board to prepare for a new game.

1.2 Maximum Number Of Moves

A 5×5 board consists of 25 cells in total. If we take a deeper insight to the worst scenario, which is when all 24 cells have been check but there's still no winner yet, then the 13^{th} move of player 1 will end the game.

The result then may be there is a winner or the game ends up with no winner. It depends on the 13^{th} move of player 1.

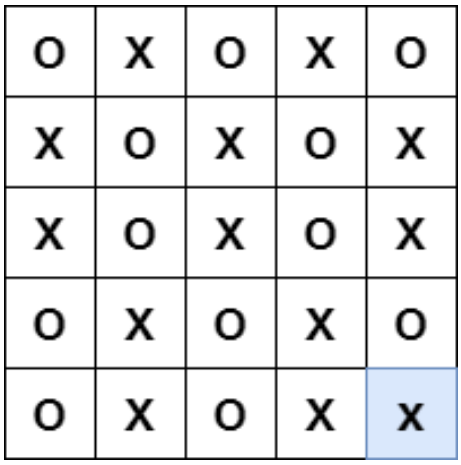

Figure 3: The 13^{th} move will bring a win for player 1 in this case.

Acknowledge this principle, we are going to organize 12 pairs of moves for 2 players and the 13^{th} move for player 1.

In each pair of moves, we will sequentially call other procedure to draw the board, read in players input and process it, request undo option and check for a winner. A specific sequence will be indicate in the following section.

1.3 Handling And Processing Player's Input

In each turns, players need to enter the number of row and column they choose. In order to make it easier for players, the index start from 1, so for example, if they choose the top left cell, they have to type in (1,1).

However, the index of array actually start from 0 in MIPS, so after receiving players input we have to minus 1 in both X- and Y- coordinates.

The first move of both players require checking if they belongs to the central point and only player 2's 1^{st} turn need to verify if it is the same with that of player 1. Since then we only need to confirm that players moves are not at a previously occupied cells. Furthermore, every turn need to make sure the value chosen by players are in the range $[1; 5]$.

As a result, we can break the process that receiving and handling players' input into 3 separated procedure, the first part is for the 1^{st} pair of move, the second one is from the 2^{nd} pair of move the 12^{th} and the last one is for the 13^{th} turn of player 1 .

1.4 Sequence Of Instructions To Run The Game

Base on our natural thinking, the flow of instruction can be organize as following:

1. Draw the board.
2. Read player 1 input and check if it is the central point or value is out of range, ask player 1 to choose again.

3. Draw the board again to show result.
4. Then continue with the undo procedure which has been mentioned above.
5. Pass the turn to player 2 and repeat the above process

The calling of procedure in the program for the 1^{st} pair of moves:

```
# Handle 1st pair of move separately
        jal Draw_board
        jal Move1_P1
        jal Draw_board
        jal Undo_Op
        beq $v0, $zero, Move1.2
        jal Draw_board
        jal Move1_P1
        jal Draw_board
        jal Update_backup
Move1.2:jal Move1_P2
        jal Draw_board
        jal Undo_Op
        beq $v0, $zero, Move2
        jal Draw_board
        jal Move1_P2
        jal Draw_board
        jal Update_backup
```

Figure 4: The following turns follow the same structure.

From the 2^{nd} pair, we can generate a loop and place the same structure in side.

```
# From 2nd pair of move to 12th pair of move needs 11 loops
Move2:
        addi $t1, $zero, 11
        addi $t0, $zero, 0
playingLoop:
        beq $t0, $t1, lastMove

        jal P1_Move
        jal Draw_board
        jal Undo_Op
        beq $v0, $zero, player2
        jal Draw_board
        jal P1_Move
        jal Draw_board
        jal Update_backup
player2:jal P2_Move
        jal Draw_board
        jal Undo_Op
        beq $v0, $zero, nextLoop
        jal Draw_board
        jal P2_Move
        jal Draw_board
        jal Update_backup

nextLoop:addi $t0, $t0, 1
        j playingLoop
```

Figure 5: The following turns follow the same structure.

The last turn only involve procedure that process player 1's move.

1.5 Checking Winner Condition

A winner is the one who has 3 consecutive checking cell in a row, a column or a diagonal.

A 5×5 board has 5 rows and 5 columns. At a result, we will need to check for each row and each column.

However, the number of diagonal, which has more than 3 cells, is greater than that. In details, there are 5 diagonal from top left to right bottom and the others 5 from top right to left bottom. In total, there are 10 diagonals that need to verify.

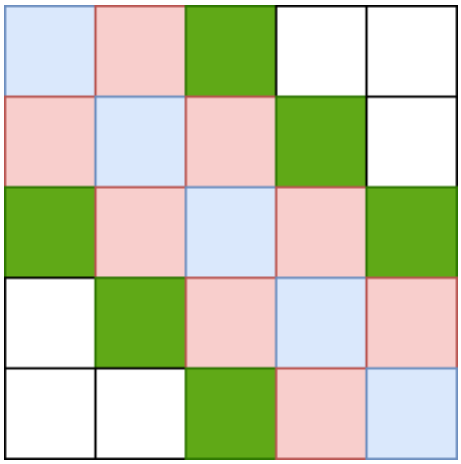

Figure 6: 5 diagonals from top left to right bottom.

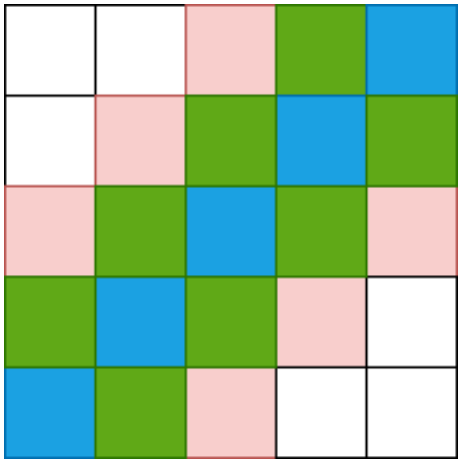

Figure 7: 5 diagonals from top right to left bottom.

To confirm that whether there are 3 consecutive locations contains 'X' or 'O', we will use the **bitwise AND operation**. The result will be the same type if all 3 cells have the same data. Then we can find out a winner at that move. Otherwise, the game still go on.

- For rows, columns or diagonals with 5 cells, we need to check 3 combinations:
  ($Cell_1$ and $Cell_2$ and $Cell_3$),
  ($Cell_2$ and $Cell_3$ and $Cell_4$),
  ($Cell_3$ and $Cell_4$ and $Cell_5$).
- For diagonals with 4 cells, we need to check 2 combinations:
  ($Cell_1$ and $Cell_2$ and $Cell_3$),
  ($Cell_2$ and $Cell_3$ and $Cell_4$).
- For diagonals with 3 cells, we only need to check one time:
  ($Cell_1$ and $Cell_2$ and $Cell_3$).

2 MIPS Implementation For Tic-tac-toe Game

2.1 Outline Of The Program

The content of the program will consists of the following procedure. Their specific purposes will be discussed in the next part.

1. Main
2. Draw Board
3. Reset Board (including one for resetting main board and the other is for backup board)
4. Undo
5. Update backup board
6. Set of procedure to get and process input
   Move1-P1 (Handle first move of player 1)
   Move1-P2 (Handle first move of player 2)
   P1-Move (Handle the other moves player 1)
   P2-Move (Handle the other moves player 2)
7. Set of procedure to place the move on the board
   P1-Play (Place moves of player 1)
   P2-Play (Place moves of player 2)
8. Result (Check for winner)

2.2 Detailed Functionality Of Each Procedure

2.2.1 Main

First of all, the main procedure will sequentially print out the greetings and all rules of the game for the players to keep up.

Then code segment for handling game will be implement including the first pair of move, from the second pair to the 12^{th} and the 13^{th} move for player 1.

After all 25 cells have been filled but there is no winner yet we will print out the drawn game message and ask if they want to replay. The replay option is also available when there is a winner.

```
# Last move
lastMove:jal P1_Move
# After all 25 cells have been filled but no winner is found, it's a drawn game
        jal Draw_board

        li $v0, 4
        la $a0, Drawn
        syscall

replay:  jal resetBoard
        jal resetBack_up

        li $v0, 4
        la $a0, Newgame
        syscall

        li $v0, 5
        syscall
        addi $t0, $v0, 0

        beq $t0, $zero, end_game
        j newgame

quit:    li $v0, 4
        la $a0, noplay
        syscall

end_game:li $v0, 10
        syscall
```

Figure 8: The rest of main procedure.

2.2.2 Draw Board

Draw board procedure will print out the current state of the main board in 5×5 text-based table. In order to that, we will need two counter registers to loop through each row and column then print the data of the cell out.

```
----------LET'S HAVE FUN :D-----------

 --- --- --- --- ---
| - | - | - | - | - |
 --- --- --- --- ---
| - | - | - | - | - |
 --- --- --- --- ---
| - | - | - | - | - |
 --- --- --- --- ---
| - | - | - | - | - |
 --- --- --- --- ---
| - | - | - | - | - |
 --- --- --- --- ---
```

Figure 9: The empty board.

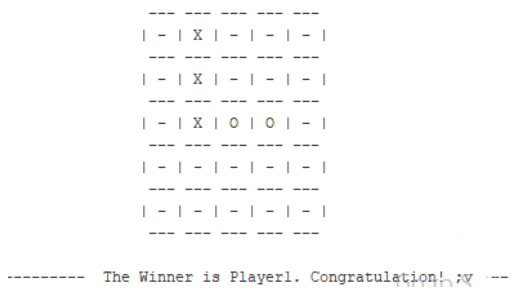

Figure 10: The board with a win belongs to player 1.

2.2.3 Reset Board

There will be two reset procedure for the main board and the backup board. However the structure of both procedure will be the same. The only difference is the register that hold the base address of the main board and the backup one.

The idea is still the same with drawing the board. We use two loops to traverse through each entry of the board instead of print out the data, we will update it with the minus sign '-', which represent for an empty cell as we all agreed before.

```
##############################    RESET BOARD PROCEDURE    ##############################################
##############################################################################################
resetBoard:
                addi $sp, $sp, -4
                sw $ra, 0($sp)

                la $t0, board           # Store address of the board
                lw $t3, size            # Stopping loop conditional
                addi $t1, $zero, 0      # Counter to traverse through each row

    resetBoard_loop_row:
                beq $t1, $t3, end_resetBoard
                sll $t4, $t1, 2
                add $t4, $t4, $t0
                lw $t5, 0($t4)          # base address of the row

                addi $t2, $zero, 0      # Counter to traverse through each column
    resetBoard_loop_column:
                beq $t2, $t3, resetBoard_next_row
                sll $t6, $t2, 2
                add $s0, $t5, $t6       # address of board[x][y]

                addi $t7, $zero, '-'
                sw $t7, 0($s0)

                addi $t2, $t2, 1
                j resetBoard_loop_column

    resetBoard_next_row:
                addi $t1, $t1, 1
                j resetBoard_loop_row

    end_resetBoard:
                lw $ra, 0($sp)
                addi $sp, $sp, 4

                jr $ra
```

Figure 11: The implement of reset procedure.

2.2.4 Undo

For undo procedure, we will get input from players, '1' represents the agreement to undo and '0' for the disagreement.

Two loops are needed for this procedure, at each location $(row; column)$ we will **load** the data from the **backup board** into a register and then **store** it back to the **main board** if the control signal is '1'.

```
########################################   UNDO PROCEDURE   ##################################################
#############################################################################################################
Undo_Op:
                addi $sp, $sp, -4
                sw $ra, 0($sp)

                la $t0, board           # Store address of the board
                la $t1, back_up         # Store address of the temparary board
                lw $t4, size            # Stopping loop conditional

                li $v0, 4
                la $a0, Undo
                syscall

                li $v0, 5
                syscall
                addi $t5, $v0, 0
        #If players agree to undo, copy data from back-up to board
                beq $t5, $zero, update_back_up

                addi $t2, $zero, 0      # Counter to traverse through each row
        undo1_loop_row:
                beq $t2, $t4, end_undo
                sll $t6, $t2, 2
                add $t6, $t6, $t1
                lw $t7, 0($t6)          # Base address of row of back-up

                sll $t6, $t2, 2
                add $t6, $t6, $t0
                lw $t8, 0($t6)          # Base address of row of board

                addi $t3, $zero, 0      # Counter to traverse through each column
        undo1_loop_column:
                beq $t3, $t4, undo1_next_row
                sll $t9, $t3, 2
                add $t9, $t9, $t7
                lw $s0, 0($t9)          # $s0 contain data in back_up[x][y]

                sll $t9, $t3, 2
                add $t9, $t9, $t8
                sw $s0, 0($t9)          # Undo board[x][y]

                addi $t3, $t3, 1
                j undo1_loop_column
        undo1_next_row:
                addi $t2, $t2, 1
                j undo1_loop_row
```

Figure 12: The reset the main board to previous state.

Otherwise we will call the update backup board procedure to overwrite data in the backup with data in current state.

```
        # Else update back-up with data in board
        update_back_up:
                jal Update_backup

        end_undo:
                lw $ra, 0($sp)
                addi $sp, $sp, 4

                jr $ra
```

Figure 13: Call the update backup board procedure if control signal is '0'.

2.2.5 Update backup board

This procedure still stick to the same idea with the reset one except for each location $(row; column)$ we will **load** the data from the **main board** into a register and then **store** it back to the **backup board** if the control signal is '0'.

2.2.6 Set of procedure to get and process input

2.2.6.a Move1-P1

The procedure Move1-P1 is implemented for processing the 1st move of player 1. Firstly, we will ask player 1 to enter the number of the row and column. Remember to minus 1 to make the value match with array's index.

```
#####################################   HANDLE 1st MOVE OF PLAYER 1  ######################################
Move1_P1:
                addi $sp, $sp, -4
                sw $ra, 0($sp)
    # Player1's input
    ReMove1_P1: li $v0, 4
                la $a0, Input1
                syscall

                # Player1 's X_coord
                li $v0, 5
                syscall
                addi $a1, $v0, -1  # Adjust to fix with array index start from 0

                # Player1 's Y_coord
                li $v0, 5
                syscall
                addi $a2, $v0, -1  # Adjust to fix with array index start from 0
```

Figure 14: Request input from player 1.

Then we need to check if the values input are out of the range $[1; 5]$. If so request the player to enter again.

```
    # Check if values are out of range
                addi $t0, $zero, 4
                slt $t1, $t0, $a1
                bne $t1, $zero, outRange1

                slt $t1, $a1, $zero
                bne $t1, $zero, outRange1

                slt $t1, $t0, $a2
                bne $t1, $zero, outRange1

                slt $t1, $a2, $zero
                bne $t1, $zero, outRange1
                j checkMid1

    outRange1:  li $v0, 4
                la $a0, Warn2
                syscall
                j ReMove1_P1
```

Figure 15: Make sure the values always belong to the interval $[1; 5]$.

Next, we need to ensure that the first move is not at (row3, column3).

```
# Check if 1st move in the central point
checkMid1: addi $t0, $zero, 2
        bne $a1, $t0, next1
        bne $a2, $t0, next1

        # If the move is on the central point, enter again
        li $v0, 4
        la $a0, Warn1
        syscall
        j ReMove1_P1
```

Figure 16: Make sure the 1^{st} move is not at the central point.

If the values pass all condition, we will call procedure P1-Play to place 'X' to the board. The functionality of P1-Play will be present later on.

```
# Else update board[x][y] with 'X'
next1:    jal P1_Play

        lw $ra, 0($sp)
        addi $sp, $sp, 4

        jr $ra
```

Figure 17: Call P1-Play to place 'X'.

2.2.6.b Move1-P2

Move1-P2 is used for handled the first move of player 2. Similar with Move1-P1, it also request player 2 input, check for out of range and central point condition then call P2-Play to make the placement of 'O'.

However, we still have one more situation which is when player 2's move is same as player 1's previous move. Then we will ask player 2 to choose another location that is still empty.

```
# Else update board[x][y] with '0'
next3:      jal P2_Play

            # If P2_Play procedure returns -1, it indicates that cell has been occupied
            addi $t0, $zero, -1
            bne $v0, -1, next4

            # Then ask to choose again
            li $v0, 4
            la $a0, Warn3
            syscall
            j ReMove1_P2

            # Else board[x][y] has successfully upfated to '0'
next4:      lw $ra, 0($sp)
            addi $sp, $sp, 4

            jr $ra
```

Figure 18: Procedure P2-Play returns -1 meaning the cell has previously occupied.

2.2.6.c P1-Move and P2-Move

P1-Move and P2-Move are designed for processing from the 2^{nd} move till the end.

The structure will act in accordance with Move1-P1 and Move1-P2. However, we do not need the central point check anymore, instead there will be checking if values are out of range and if the latest moves belong to previously occupied location.

```
########################################   HANDLE FROM 2nd MOVE OF PLAYER 1  ########################################
P1_Move:
            addi $sp, $sp, -4
            sw $ra, 0($sp)
    # Player1's input
    ReP1_Move: li $v0, 4
            la $a0, Input1
            syscall

            # Player1 's X_coord
            li $v0, 5
            syscall
            addi $a1, $v0, -1  # Adjust to fix with array index start from 0

            # Player1 's Y_coord
            li $v0, 5
            syscall
            addi $a2, $v0, -1  # Adjust to fix with array index start from 0

    # Check if values are out of range
            addi $t0, $zero, 4
            slt $t1, $t0, $a1
            bne $t1, $zero, outRange3

            slt $t1, $a1, $zero
            bne $t1, $zero, outRange3

            slt $t1, $t0, $a2
            bne $t1, $zero, outRange3

            slt $t1, $a2, $zero
            bne $t1, $zero, outRange3
            j play1

    outRange3: li $v0, 4
            la $a0, Warn2
            syscall
            j ReP1_Move

    play1:  jal P1_Play

    # If P1_Play procedure returns -1, it indicates that cell has been occupied
            addi $t0, $zero, -1
            bne $v0, $t0, continue1

    # Else enter again
            li $v0, 4
            la $a0, Warn3
            syscall
            j ReP1_Move

    continue1: lw $ra, 0($sp)
            addi $sp, $sp, 4

            jr $ra
```

Figure 19: Implementation of procedure P1-Move.

2.2.7 P1-Play and P2-Play (Set of procedure to place the move on the board)

These two procedures will mainly be used to place the 'X' and 'O' on the board. Before placing the move it will check if the location has been occupied or not. If the cell has been filled previously, these procedure will return -1 and jump back to the move procedure to ask for re-enter. Otherwise, it will place 'X' or 'O'.

After the placement, it will call the Result procedure to check for winner. If the latest move causes a win, the announcement will be made and jump to the replay label in main procedure which contain the code segment allowing the players to start a new game.

If no winner is found, it will jump back to the place of call in the move procedures.

Both procedure have the exact flow so the following demonstration will about P1-Play only.

+ Initially, we will need to allocate memory in stack for the return address, row's number and column's number. Then we load data into a register to check if it's different than the '-' character, we update the location with 'X'; else we jump to Occupied label where the return value is set to -1

```
#####################################  PROCEDURE SET TO PLACE A MOVE  #####################################
#############################################################################################################
# Procedure P1_Play: Update board[x][y] = 'X', return 0 if success, 1 if P1 wins, -1 if the location is previously occupied
P1_Play:
                addi $sp, $sp, -12
                sw $ra, 0($sp)
                sw $a1, 4($sp)
                sw $a2, 8($sp)

                # Fetch the data at board[x][y] to check if it is occupied or not
                la $t0, board      # Store address of board
                sll $t1, $a1, 2
                add $t1, $t1, $t0
                lw $t2, 0($t1)     # base address of the row

                sll $t1, $a2, 2
                add $s0, $t1, $t2  # base address of board[x][y]
                lw $t2, 0($s0)

                # If board[x][y] != '-', this mean it has been previously occupied
                addi $t5, $zero, '-'
                bne $t2, $t5, Occupied
                # Else board[x][y] = 'X'
                addi $t3, $zero, 'X'
                sw $t3, 0($s0)
```

Figure 20: Implementation of procedure P1-Play.

```
Occupied:
                addi $v0, $zero, -1     # Return -1 to require players enter again

                lw $ra, 0($sp)
                lw $a1, 4($sp)
                lw $a2, 8($sp)
                addi $sp, $sp, 12

                jr $ra
```

Figure 21: Free memory in stack and return -1.

Next, the call of Result procedure is involved. If return value of Result is 1, we print out the announcement and offer replay option. Otherwise, free memory in stack and jump back to the place of call.

```
                # Check for winner
                jal Result

                # Procedure Result return 1 if a winner is found
                addi $t4, $zero, 1
                beq $v0, $t4, P1_Winner

                # Else return 0 success updating board[x][y] with no winner
                addi $v0, $zero, 0

                lw $ra, 0($sp)
                lw $a1, 4($sp)
                lw $a2, 8($sp)
                addi $sp, $sp, 12

                jr $ra

P1_Winner:
                jal Draw_board

                li $v0, 4
                la $a0, P1_Win
                syscall

                lw $ra, 0($sp)
                lw $a1, 4($sp)
                lw $a2, 8($sp)
                addi $sp, $sp, 12

                j replay
```

Figure 22: Find out the winner or continue with the game.

2.2.8 Result

The result procedure's major role is to successively check each row, column and diagonal for 3 consecutive cells of same type 'X' or 'O'. The idea was to use the Bitwise AND operator for data in 3 cells as we mentioned before. The result of this AND operation will be 'X' or 'O' if all 3 cells are all 'X' or 'O'. Then certainly, we can find out a winner.

As has been analysed before, there are 5 rows, 5 columns and 10 diagonals in total, that need to verify. So the Result procedure will continuously call other procedure for the checking out. At the end, if there still no winner we return the value of 0 and jump back to the play procedures.

```
##############################   PROCEDURE TO FIND OUT WINNER   ##############################
#############################################################################################
# Result procedure check if the lastest move of any players cause a win, return 1 if there is a winner, otherwise 0
Result:
                addi $sp, $sp, -4
                sw $ra, 0($sp)

                jal checkRow0
                jal checkRow1
                jal checkRow2
                jal checkRow3
                jal checkRow4

                jal checkCol0
                jal checkCol1
                jal checkCol2
                jal checkCol3
                jal checkCol4

                # From top left to right bottom, there are 5 diagonals with more than 3 cells
                jal checkLeftDiag0
                jal checkLeftDiag1
                jal checkLeftDiag2
                jal checkLeftDiag3
                jal checkLeftDiag4

                # From top right to left bottom, there are also 5 diagonals with more than 3 cells
                jal checkRightDiag0
                jal checkRightDiag1
                jal checkRightDiag2
                jal checkRightDiag3
                jal checkRightDiag4

                # After checking all rows, columns and diagonals if there is no winner return 0
                addi $v0, $zero, 0
                lw $ra, 0($sp)
                addi $sp, $sp, 4

                jr $ra
```

Figure 23: The Result procedure.

Previously, we have declared our board as type word so we have to keep in mind that index values are always multiple of 4.

All the check procedure follow the same structure, which include fetching data in consecutive cells, performing the AND operation to check whether the result is the same type, if so we return 1, otherwise return 0.

Because of that, I will demonstrate the checkRow0 and checkCol0 procedures only. The others can be obtained easily by changing the index.

All cells on the same row will have the same row index and the column index from $cell_0$ to $cell_4$ will be 0, 4, 8, 12, 16.

```
checkRow0:
        la $t0, board           # Address of board
        lw $t1, 0($t0)          # Address of row 0
        lw $t2, 0($t1)          # board[0][0]
        lw $t3, 4($t1)          # board[0][1]
        lw $t4, 8($t1)          # board[0][2]
        lw $t5, 12($t1)         # board[0][3]
        lw $t6, 16($t1)         # board[0][4]

        # There are 3 posible combination
        # If 3 consecutive cells are all 'X' or 'O', the result will be the same type
        and $t7, $t2, $t3
        and $t7, $t7, $t4

        and $t8, $t3, $t4
        and $t8, $t8, $t5

        and $t9, $t4, $t5
        and $t9, $t9, $t6

        addi $s0, $zero, 'X'
        addi $s1, $zero, 'O'

        # If a winner is found, return 1
        beq $t7, $s0, returnWinner
        beq $t7, $s1, returnWinner
        beq $t8, $s0, returnWinner
        beq $t8, $s1, returnWinner
        beq $t9, $s0, returnWinner
        beq $t9, $s1, returnWinner

        # Else return 0
        addi $v0, $zero, 0

        jr $ra
```

Figure 24: The checkRow0 procedure.

Conversely, cells on the same column will have the same column index and the row index will range from 0, 4, 8, 12, 16. However, we need to get the row address first so two instructions are needed to fetch data at a specified location.

```
checkCol0:
        la $t0, board           # Adrres of board

        lw $t1, 0($t0)
        lw $t2, 0($t1)          # board[0][0]

        lw $t1, 4($t0)
        lw $t3, 0($t1)          # board[1][0]

        lw $t1, 8($t0)
        lw $t4, 0($t1)          # board[2][0]

        lw $t1, 12($t0)
        lw $t5, 0($t1)          # board[3][0]

        lw $t1, 16($t0)
        lw $t6, 0($t1)          # board[4][0]

        # There are 3 posible combination
        # If 3 consecutive cells are all 'X' or 'O', the result will be the same type
        and $t7, $t2, $t3
        and $t7, $t7, $t4

        and $t8, $t3, $t4
        and $t8, $t8, $t5

        and $t9, $t4, $t5
        and $t9, $t9, $t6

        addi $s0, $zero, 'X'
        addi $s1, $zero, 'O'

        # If a winner is found, return 1
        beq $t7, $s0, returnWinner
        beq $t7, $s1, returnWinner
        beq $t8, $s0, returnWinner
        beq $t8, $s1, returnWinner
        beq $t9, $s0, returnWinner
        beq $t9, $s1, returnWinner

        # Else return 0
        addi $v0, $zero, 0

        jr $ra
```

Figure 25: The checkCol0 procedure.

Whenever the outputs of the And operation is the same as its inputs, we jump to the

returnWinner label to return value of 1, which indicates that a winner is found when the latest move has been made.

```
returnWinner:
                addi $v0, $zero, 1
                lw $ra, 0($sp)
                addi $sp, $sp, 4

                jr $ra
```

Figure 26: The returnWinner label.