VIETNAM NATIONAL UNIVERSITY – HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FALCULTY OF COMPUTER SCIENCE AND ENGINEERING

COMPUTER ARCHITECTURE (LAB) REPORT

Instructor: Dr. Phạm Quốc Cường

Student: Trần Minh Trung

Class: CC01

Student ID: 2153073

Ho Chi Minh city, March 30th 2023

I. Introduction

This report will explain and analyze my algorithms which are shown in my .asm file about implementing the game “Connect Four” in MARS MIPS.

II. Initialization

1. Defined parameters

- To complete many tasks required, first I have to use some variables to store 3 times each player can undo their move as well as the number of violations, one time to block the next opponent’s move, and one time to remove one arbitrary piece of the opponent (for example in my code I use $s1 and $s2 to store the remaining violations possible of each player, $s3 and $s4 to store the remaining undo possible, …). And each time a player decides to use a typical function like undo or block the opponents, the corresponding variables will be reduced by 1 until they become 0.

2. Some basics concepts

- To create a table (6 rows and 7 columns) I use 6 strings to represent each rows as following format: “(n^{th} row) + |_|_|_|_|_|_|_|”. I also use one more string to store the position of column (1 2 3 4 5 6 7) so that players can easily choose a particular column.

  In conclusion, by using in total 7 strings I can create a game table that looks something like this:

String 7: 1 2 3 4 5 6 7
String 1: 1 |_|_|_|_|_|_|_|
String 2: 2 |_|_|_|_|_|_|_|
String 3: 3 |_|_|_|_|_|_|_|
String 4: 4 |_|_|_|_|_|_|_|
String 5: 5 |_|_|_|_|_|_|_|
String 6: 6 |_|_|_|_|_|_|_|

(Full table image)

- First I let two players to choose their name respectively, then I randomly assigned piece X or O to them by using *syscall 42* (li $v0,42) and set the upper bound $a1 to 2 (The function *syscall*

42 will return in $a0 randomly integer values in range $0 \leq \$a0 < \$a1 = 2$). I always set the X piece to player 1 and the O piece to player 2, but if the function *syscall 42* returns a value 1, then I will swap the name of two players (which means the player 1 now becomes the player 2 and starts the game with piece O, and vice versa). The player who starts the game with piece X will go first.

3. Functionality of algorithms

- In order to demonstrate the game, I will use a while loop contains following information:
  - In the first round, after having been requested a column, both players have to drop a piece at column 4 (any input besides 4 is counted as a violation).
  - After the first round (every round after the first one), each player will be asked if they wants to remove one arbitrary piece of the opponent (if they haven't used this function):
    - ♦ If the player wants to remove, then after completing removing an opponent's piece, the player will be asked if they want to block the opponent's next move (if they haven't used this function). If the player wants to block, then the game will continue with this player's turn but he/she will not be asked to remove a piece or block the opponent's move anymore (these function are one-time use only).
    - ♦ If the player do not want to remove, then they will be asked to pick a column to drop their piece (any input that is < 1 or > 7 will be considered as a violation which restarts the move). After dropping a piece, the player will be asked to undo their last move (if there is at least 1 remaining undo left). If the player choose to undo, then the program delete the recent piece and return to this player's selection phase. Finally, the player will be asked to block the opponent's next move (if they haven't used this function yet.
  - If the player already use the *remove* function then at the start of each turn they can only drop a piece. After that the *undo* function and *block* function appear or not depends on the remaining undo and block variables.
  - At the beginning, there will be two variables to store each player's total pieces. When each player decides to drop a piece, the corresponding variable will be increased by 1 (it will also be decreased by 1 if the other player choose to use *remove* function). Moreover, at the commence of each player's turn, there will be a condition to check if both pieces of players

add up to 42, if this condition is (total pieces of two players are 42) then the program will print “TIE GAME” before terminated.

- At the end of each player’s turn, my program will call functions (*Check_horizontal*, *Check_vertical*, *Check_diagonal_right*, *Check_diagonal_left*) to check if there is a winner or not.

- Generally, there are some functions that I wrote to support the program:
  - Function *Add_to_column*
  - Function *Block*
  - Function *Remove*
  - Function *Check_horizontal*
  - Function *Check_vertical*
  - Function *Check_diagonal_right*
  - Function *Check_diagonal_left*
  - Function *Endmove*
  - Function *Undo*

III. Details of each function

- Note: Some of below functions have two version corresponding to player 1 and player 2.

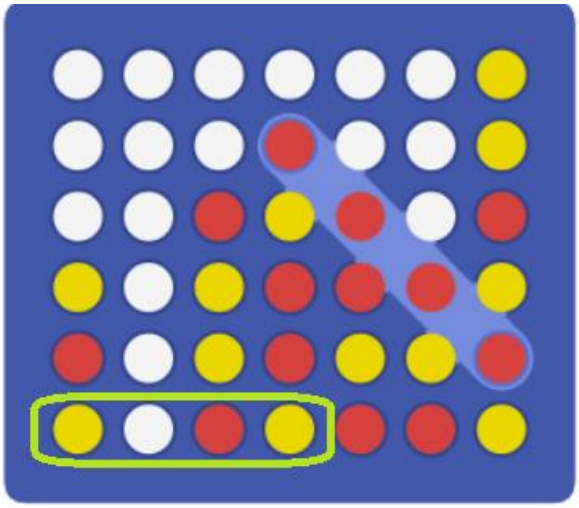
1. Function *Add_to_column*

- By definition, this function is used to insert X (or O) to the table after players pick a column. Recall, any input that is < 1 or > 7 will be considered as a violation which restarts the move. When the player enter a valid column, this function will be called. At first, it will start by loading address of string 6 (The lowest row of the table), then the program will compare the input of the player with numbers from 1 to 7, respectively. To be precise, if the input of the player is 1 then the address variable, which stores the address of String 6, will be increase by 3 (recall string 6: “6 |_|_|_|_|_|_|”. To get access to the first column which is the first character “_” we have to increase the base address by 3 as shown in the below picture).

- From the upper picture, I can figure out that if the player's input is 2, then I have to increase the address variable by 5, 3 will be 7, 4 will be 9, and so on, ….
- After successfully determining the correct column to insert a piece, I will check if there already existed a piece in rows 6 (by load the byte out (*lb*), then compare it with the byte '_' using *bne*) , if yes I will iteratively go to row 5 (increase the address variable like the way done for row 6) to check the existence of a piece using the same method for row 6. This progress runs ceaselessly until it find an empty spot, then the function will insert piece X (or O according to player's turn) by using store byte (*sb*) (for example *li* $s0,'X' then *sb* $s0, 0($t1) with $t1 is the position of byte '_').
- If the function reaches row 1 but cannot find an empty spot, then the program will print out a sentence to warn the player that he/she has violated (by dropping a piece at full column) and force the player to choose again another column.

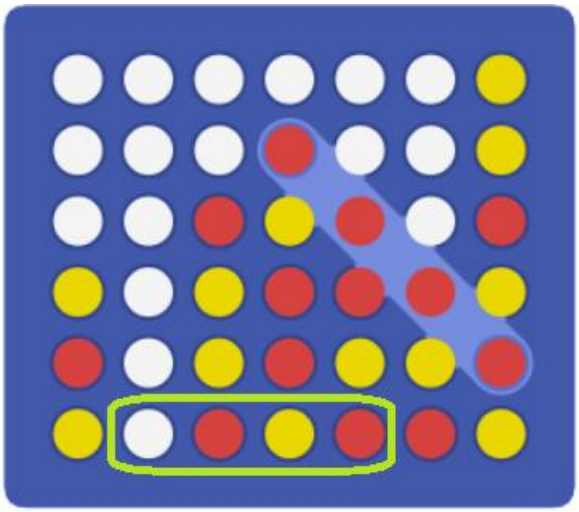
2. Function *Check_horizontal*

- To begin with, first I started to check only the row that has been filled with the last piece, but soon after there is a problem that when one player decides to use function *remove* then four consecutive pieces can be at any rows on the table. Therefore, I had to check the entire table, which means I must iteratively traverse all possible cases about horizontal win from row 6 to row 1.
- Let's take row 6 (and the win condidion is for player 1(X)) for example, first I will check the first four pieces surrounded by the green border as shown in below picture:

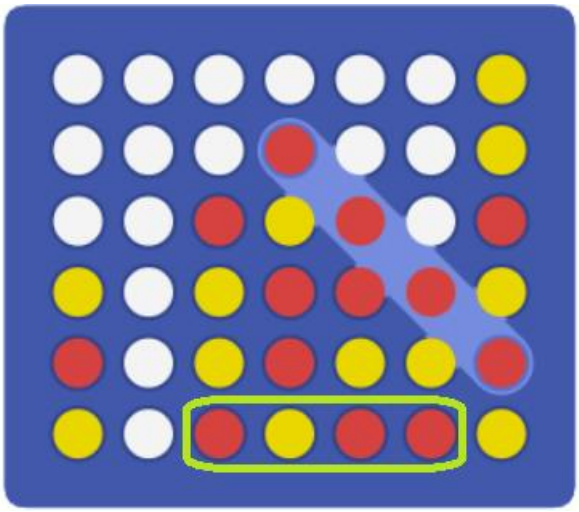


(Picture 1)

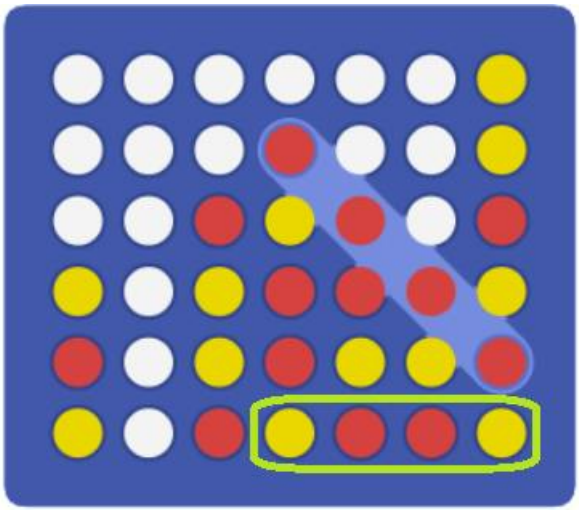
- First I will load the address of string 6, iteratively increase this variable from 3, then 5, then 7 and then 9 to check if all of them are X pieces (by using branch-if-equal (*beq*), at each comparison I will use *lb* to get the wanted character then compare it with a temporary variable containing 'X'). If the character at position 3 is 'X', then the function will move on to check the next position which is 5 and so on. At any time if the character at position 3 or 5 or 7 or 9 is not 'X' then the function will jump to examine the next four pieces as shown in the below picture:



- Same method is applied for this scenario, if one of these rounded pieces turns out to be 'O' or '_', then the function will continue to examine the next four like this:



and



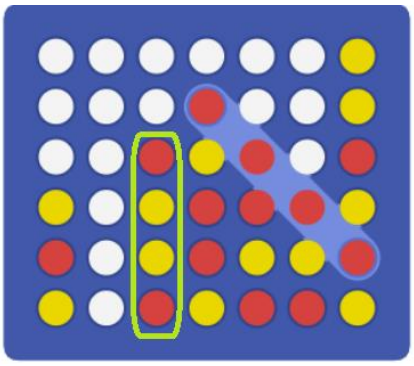
- The function will continue until it finds a four consecutive pieces. In conclusion, at each row the function will execute four times to check four possible conditions of horizontal win. If row 6 has been checked four times but still cannot find four consecutive pieces, then the function will continue at row 5 (same procedures are applied: load address of the row 5, then iteratively

increase the address variable by 3 5 7 9, compare with variable ‘X’, then 5 7 9 11, and so on, …).

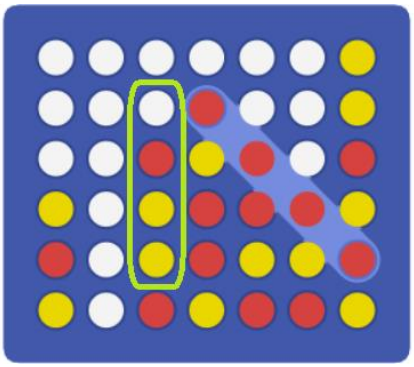
- The procedures for check win condition for ‘O’ is the same by changing the temporary variable to ‘O’.
- If the function finds four consecutive pieces at any position, then the program will print out the winner as well as the total pieces of this player and finish execution.
- In case the function finishes checking row 1 but still cannot find any four consecutive pieces then the program will jump to function *check_vertical*.
- There will be a bool variable (value only 0 or 1) to mark if any player already had 3 consecutive pieces. For example, the function is checking the row 6 at position 3, 5 , 7, and 9 (picture 1), if the function manages to come to check at position 9 then the bool variable will be marked to 1, else it will remain 0. This procedure will be repeated 24 times (4 times each row, total 6 rows). This bool variable will be used to check whether the opponent can use the function *block* or not later. (*)

3. Function *Check_vertical*

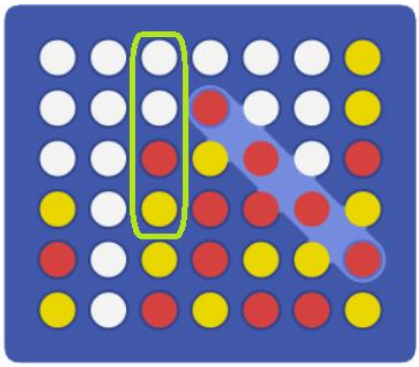
- Check four consecutive vertical pieces is quite simpler than horizontal method. All I have to do is just check only the column that has been filled with the last piece. For example, if player 1 drops a piece at column 3, then this function will only check the column 3 to find four consecutive pieces.
- In one column there are 3 possibilities as shown below:



(case 1)



(case 2)



(case 3)

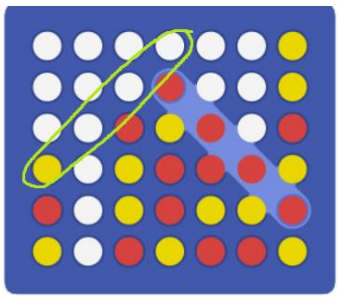
- The function will initially check case 1 first by loading the address of row 6, increasing the address variable to the same column that last piece has been filled. After that, the function will

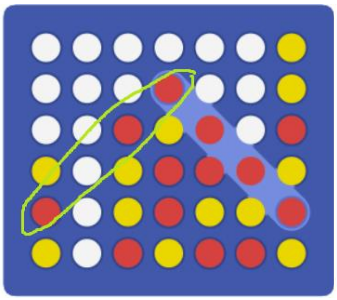
load byte (*lb*) the character at examined column (at row 6) to compare with 'X' (check for player 1) and 'O' (check for player 2) by using branch-if-equal (*beq*), if the condition is true then the function will continue by loading the address of row 5 and repeat same steps above. If the condition is false at any *beq* steps, the function will move to case 2 (repeat the same procedure above) or case 3.

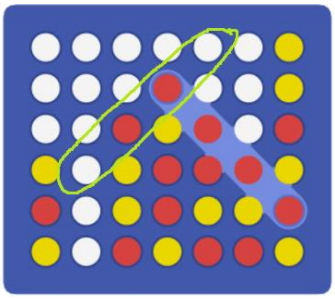
- If the function can find four vertical consecutive pieces at any case, then the program will print out the winner as well as the total pieces of this player and finish execution.
- If the function cannot find any four vertical consecutive pieces even if it has examined the last case (case 3), then the program will jump to function *check_diagonal_right*.
- Just before checking the case 1, this function *check_vertical* will check if it is called after function *remove*. If true then the function *check_vertical* will check the column where player have just removed an opponent's piece.
- Similar to function *Check_vertical*, there will also a bool variable to check whether the player has 3 consecutive pieces or not (same method as described in function *Check_vertical* (*)).

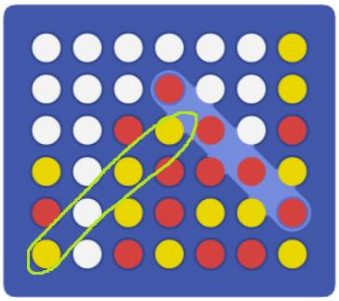
4. Function *Check_diagonal_right*

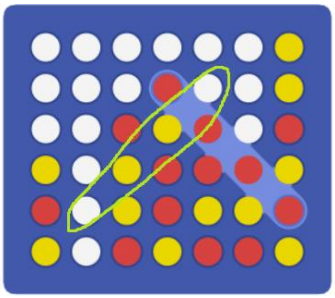
- In this function, I will check total 12 diagonal lines that starts from right to left (I mark them from 1 to 12) as shown:

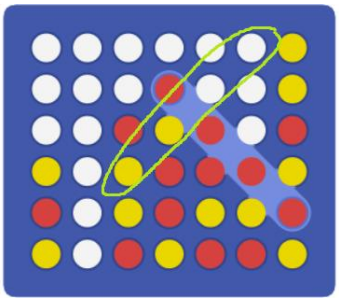

(1st line)

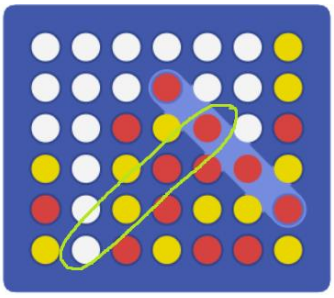

(2nd line)

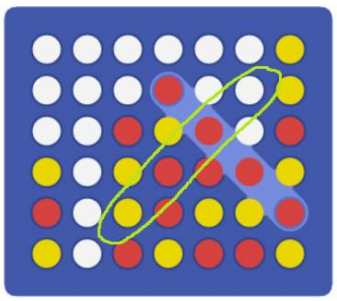

(3rd line)

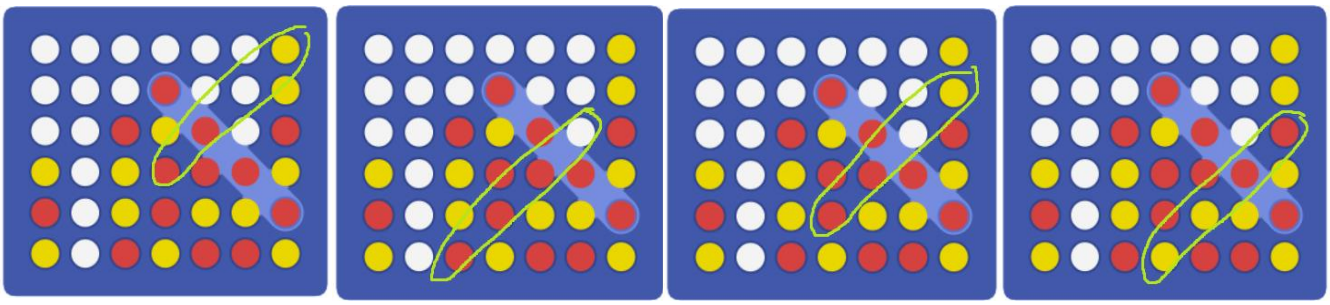

(4th line)


(5th line)


(6th line)


(7th line)


(8th line)

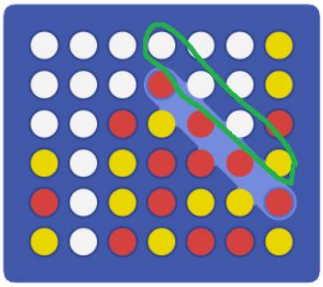


(9th line) (10th line) (11th line) (12th line)

- Take the 1st line as an example, first I will load the address of row 4, increase the address variable by 3, load the byte (*lb*) out to compare with piece 'X' or 'O' using branch-if-equal (*beq*). If the *beq* return 1 then I will continue to load address of row 3, increase the address variable by 5 and repeat the same steps above, if the condition is true then the function will load the address of row 2, increase the address variable by 7, repeat steps and again if the condition is true then I will load the address of row 1, increase the address variable by 9, if the condition is true then we will have a winner. Any *beq* statements that return 0 will cause the function to move to check the next line.
- If four consecutive *beq* return 1 which means there exists a diagonal line containing 4 piece 'X' or 'O' continuously, then the function will cause the program to print out the winner as well as the total pieces of this player and finish execution.
- In case the function has already checked all 12 lines but cannot find any consecutive diagonal line, the function will jump to function *Check_diagonal_left*.
- Similar to function *Check_vertical*, there will also a bool variable to check whether the player has 3 consecutive pieces or not (same method as described in function *Check_vertical* (*)).

5. Function *Check_diagonal_left*

- This function is simply just a copy and some small adjustments compared to the function *Check_diagonal_right*. There will still be 12 lines and they are symmetrical to 12 lines shown in function *Check_diagonal_right* respectively. For example the 1st line will look like this:



- All checking methods remain the same like function *Check_diagonal_right*, the only difference is that we will decrease the address of each row after iterations. For example, when we check the 1st line as shown in upper picture, first the function will load the address of row 4, increase the address variable to 15, then load the address of row 3, increase the address variable to just 13, 11 for row 2 and 9 for row 1. Same technique is used for remaining 11 lines.
- In case the function has already checked all 12 lines but cannot find any consecutive diagonal line, the program will return to function *Endmove*.
- Similar to function *Check_vertical*, there will also a bool variable to check whether the player has 3 consecutive pieces or not (same method as described in function *Check_vertical* (*)).

6. Function *Endmove*

- Actually, after players drop a piece or use function *Remove*, then program will jump to this function. Inside this funtion there will be four check functions above as well as function *Undo* and *Block*.

7. Function *Undo*

- There will be function *Undo* inside function *Endmove* which means the player will be asked to undo their move. If the answer is yes (input 1) then the function will use store byte (*sb*) to store '_' again in the most recent spot, else (the answer is no, input 0) *Endmove* will call function *Block* (if the player haven't used this function yet) or jump to next player's turn.
- Any input different from 0 and 1 will not be accepted and force the players to choose again.

8. Function *Block*

- The program will ask player if they want to block the opponent's next move, press 1 else press 0. Other input is not accepted and force the player to choose again.
- If the player choose 1, then the function will check if the bool variable is 0 or 1 (to see if the opponent has 3 consecutive pieces or not) to decide whether this player can block or not. If the player use this function *Block* successfully, then he/she will not be able to use this function once again (one time use only).
- If the player successfully block the opponent's next move, then the program will jump back to this player's turn and decrease the variable represent number of times player can use this function *Block* by 1, else nothing happens.

9. Function *Remove*

- At the beginning of each player's turn, the program will ask players if they want to remove a piece of their opponent or not by input 1 (yes) or input 0 (no). Any other input beside 1 or 0 is not accepted and force the player to input a number again.
- If the player press 1 (yes) then the function will ask the row index and column index containing the opponent's piece. Any input that exceed the range of row (from 1 to 6) and column (from 1 to 7) will be treated as error and cause the player to choose again. If the player choose a spot that does not contain the opponent's piece then the function will also force the player to choose again different coordinates.
- If the player press 0 (no) then the program will ask players to pick a column and jump to function *Add_to_column.*
- Once the player input valid coordinates, then the function first will determine the spot by load address of corresponding row, then increase the address variable to fit with the column choice of the player. After that, the function will delete the piece available in the spot by using store byte (*sb*) to store '_' again.
- After "delete" a piece, function *Remove* will call function *Drop* to determine whether there is (are) piece(s) that lie above the deleted one. In function *Drop*, it will check all rows that above the row containing deleted one. For example, if player 1 decide to remove a piece of player 2 at column 1 row 6, then after successfully deleting the piece, the function will check on row 5 and at the same column (column 1), the function will use load byte (*lb*) to check if the character at column 1 row 5 is '_' or not by using *bne*, if the condition is false then nothing happens. However, if the condition is true, then we have to copy the character of column 1 row 5 to column 1 row 6, after that repeat the step: go to row 4 to check if there is a piece then copy it to row 5. The function repeats until it find a character '_' at any upper row or when it reaches row 1.
- After executing function *Drop*, the program will jump to 4 check function to check whether after removing a piece, some piece dropped can cause a player to win. If two players simultaneously have 4 consecutive pieces, then the program will show the tie result.
- After function *Remove* execution, if the player still has 1 chance to block the opponent, then the program will jump to function *Block*, else jump to the other player's turn.

- After successfully use this function, the player will never be able to use this fucntion again (one time use only).

IV. Reference

[1] Four in a Row, AI Gaming, https://help.aigaming.com/game-help/four-in-a-row.

[2] Connect Four, Wikipedia, https://en.wikipedia.org/wiki/Connect_Four.