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

COMPUTER ARCHITECTURE (CO2007)

Assignment (Semester: 221)

"FOUR IN A ROW"

Advisor(s): Dr. Phạm Quốc Cường

Student: Lý Gia Huy - 2053038

HO CHI MINH CITY, OCTOBER 2022

Contents

1 Outcomes

After finishing this assignment, students can proficiently use:

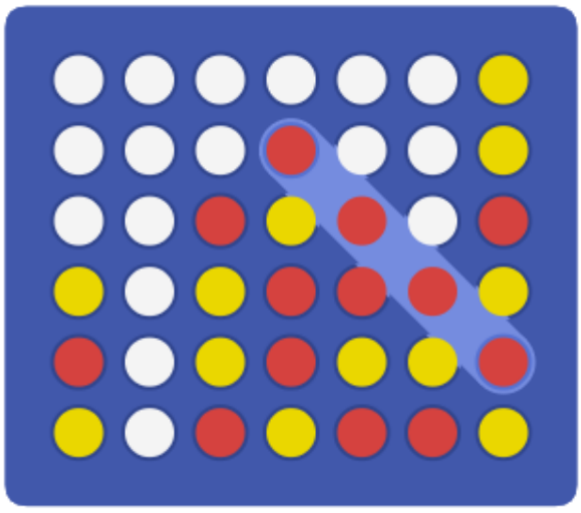
- MARS MIPS simulator.
- Arithmetic & data transfer instructions.
- Conditional branch and unconditional jump instructions.
- Procedures.

2 Introduction

2.1 Four in a row

Four in a row (also known as Connect 4, Four Up, Plot Four, Find Four, Captain's Mistress, Four in a Row, Drop Four, and Gravitrips in the Soviet Union):

- Is a board game for two players where each player selects a color.
- Then, alternately drop colored tokens into a seven-column, six-row, vertically hanging grid.
- The pieces are positioned at the lowest spot in the column as they descend straight down.
- Being the first player to arrange four of their own tokens in a line that is either horizontal, vertical, or diagonal is the game's goal.
- Connect Four is a solved game. The first player can always win by playing the right moves.



Hình 1: Winning the game

2.2 From the real game into MIPS code

Starting from a classic game, this assignment is a MIPS design based on the following guidelines:

- First, randomly choose the starting player and let this player pick the piece (X or O). The other one has to stick with the remain.
- Then, let the game begin. Four in a Row rules are based on the description at section
- Moreover, in the middle of the game (after their first move), each player has 3 times to undo their move (before the opponent's turn)
- Finally, the output of the program is the result of the game.

3 Into the program

At the first touch of my program, we all know that can't create a seamless programming structure with numerous continuous lines of code if we want to create a game that can be finished entirely with repeated loop gameplay actions.

As a result, I divided the entire project into **11 distinct tasks**, including the main function, when creating this assignment. Most of the tasks are run as an iterative operation with the hope that users can have a special experience with this game. Here are some introductions to the tasks that are broken down to complete the code

3.1 Module introduction

For interaction and visuals

- GameRules: Used to explain the rules of the game for users to easily play the game
- BoardPrint: Used to print the image of the board game on the screen
- ReadUser1Input: Mark user 1's piece into the board game after reading user 1's actions.
- ReadUser2Input: (The same but for user 2)
- PlayerChoosePiece: Select user 1's piece at the start of the game (X or 0)

Four main states throughout the game

- mainbegin: The place where the game's home page starts
- GameExit: Where to declare the system call command to end the game
- GamePlay: Where the game takes place in real-time
- GameEqual: Where both players tie and all cells are closed. A variable to track the overall number of game cells will be present here.

Some logic tasks, arithmetic calculations, and test cases in the game

- BoardEmpty: Empty the board after the game end to ready a new gameplay.
- WinConditionForPlayer: This will be the hardest and most important task, we will check the necessary conditions to have a winner in the game right here.

3.2 Iterative operation all over the I/O interaction of the program

To optimize the user's experience in this game, all operations on the game were used by me using iterative methods. The data type that the user will enter through the keyboard is the **char** data type instead of **int**. The reason for this is to avoid the game crashing during play when the user, instead of entering a number, accidentally presses a key with a text format and the game is forced to stop.
Thereby my game received some advantages as follows:

- The game will run continuously without interruption unless the user presses 0 to branch to the end of the game.
- For each option that is not included in the pre-built branching structure, the user receives an invalid input message and is allowed to re-enter the data into the game.

Of course, there will be a small disadvantage to this. As a result, my code will be longer, and since I'll have to deal with the **char** data type rather than the **int** data type, managing the arithmetic operations will be more challenging. I use a small trick to change the input from char to int and then use that to solve this for each input operation.

The data read in will be subtracted for an interval with the letter **'0'** so that the value of the data in the ASCII table is returned to the correct number to be processed. Please take a look at what I've done.

```
    #Read input, $s1 will contain the number
    li $v0, 12        #Read player's input
    syscall
    move $s1, $v0
    lw $t0, char0
    sub $s1, $s1, $t0
    la $a0, endl        #Endl
    li $v0, 4
    syscall
    ########### End reading #########
```

Thereby, the input data will be stored in register **$s1** as an **int**, and arithmetic operations are performed more easily.

4 Penetrate deeper into arithmetic operations

It is obvious that a complete game depends not only on input operations but also on the arithmetic manipulation of game logic. Thereby, I have divided it into 4 parts to handle the game logic when finding the winner during the game.

4.1 Checking vertical results

Because the input data will be located at the bottom of the playing area. So the player's pieces when placed last will be on top of the available pieces. This will make the job even easier.

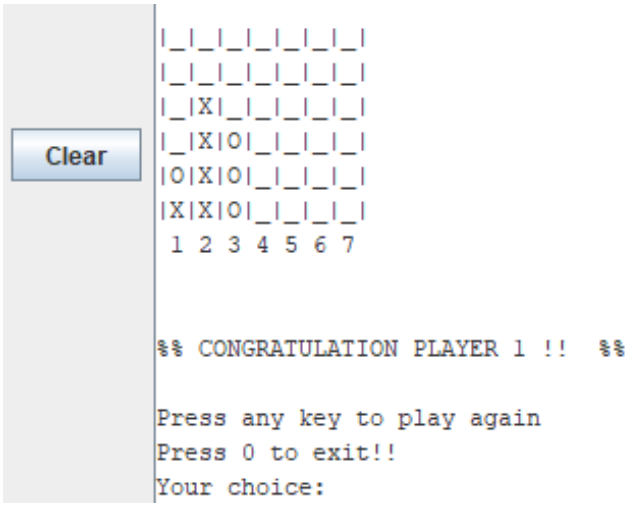
The algorithm of the vertical check is as follows:

- First I will have the system get the address of the last piece in the board.
- After that, an iterative process will be performed four times. There will be a register to determine how many times the iteration has been performed.
- I will test by having the address of the test variable jump 4 times to the address of the row below. Since I have 16 characters per row, each jump will have to skip 16 letters. Therefore, the variable will add the value of 64 (Because each char carries 4 bytes).
- With each jump we compare whether, at the location of the test variable, its actual value is equal to the player's piece (X or O). If not, we will continue to check the horizontal line. And similar to the variable that counts the number of times, if the variable counts to 0 we also jump to the next operation.
- The winner is determined when all 4 tests are satisfied that their value is equal to the user's symbol value (X or O).

Please take a look at my work this time

```
checkingVertical:
  add $s3, $s0, $zero       # $s3 have the address of the checking position
  li $s4, 4         # $s4 is the counter for win-condition
  li $t2, 4         # $t2 is the counter for loop
  checkingVertical_loop:
    lw $t0, playercheck     # $t0 have player1 piece
    lw $t1, ($s3)       # $t1 have the UUT piece
    bne $t1, $t0, checkingHorizontal  # if $t0 != $t1 => check next stage
    #if their equal, we update counters
    add $s3, $s3, Row_byte      # jump to next ROW
    #Win condition when $s4 = 0
    sub $s4, $s4, 1
    beqz $s4, win
    ############################
    sub $t2, $t2, 1
    beq $t2, $zero, checkingHorizontal
  j checkingVertical_loop
```

Upon investigation, it is discovered that there is still no winner. I changed to looking at the outcomes horizontally.



Hình 2: Winning by vertically

4.2 Checking horizontal results

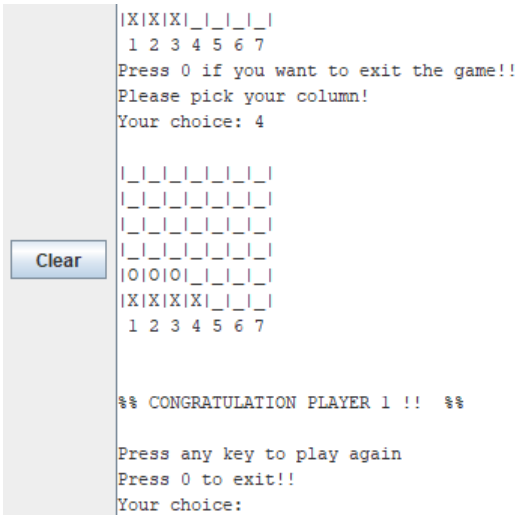
This time the algorithm will be a bit different because we encounter more cases. For instance, in the check position, we collect four in a row with a result of
3 left; 2 left - 1 right; 1 left - 2 right; and 3 right

Therefore, the algorithm to check the horizontal row will take place as follows:

- First, I will give the address of the last fragment similar to when checking the vertical line (Of course in all operations this step is included).
- Then I will save another variable whose value is the address of the whole board. This is also the address of the first char on the whole board. This variable will be used to compare with the test variable if we accidentally have the test variable at the top of the table. This will make it easier for me to scroll left to check and not be afraid of reading data out of range
- Next, the iterative operation to check the left will take place 4 times and the algorithm is similar to the vertical check. However, the value to subtract will be different. I'll subtract 2 times, the first time to check if we're out of range, and the second to check if the piece is equal to the player's piece value, subtracting 4 bytes each time.
- When we notice that there is a piece that is different from the player piece, we will not stop the check operation but will jump to the nearest piece to the right to continue checking. If jumping left all 4 times is correct, the player will win.
- Then from the updated position, we check to the right 4 times. This will help us not to miss any fragment when during the test we pass the position of the fragment that was loaded with the original address from the time of the test.

Listed below is the test work's MIPS code.

```
checkingHorizontal:
  la $s2, board         # $s2 will have address of the first char
  add $s3, $s0, $zero       # $s3 have the address of the checking position
  li $s4, 4         # $s4 is the counter for win-condition
  li $t2, 4         # $t2 is the counter for loop
  MoveLeftLoop:
    sub $s3, $s3, 4
    beq $s3, $s2, checkingHorizontal_Start1
    sub $s3, $s3, 4
    lw $t0, playercheck     # $t0 have player1 piece
    lw $t1, ($s3)       # $t1 have the UUT piece
    bne $t1, $t0, checkingHorizontal_Start  # $t0 != $t1 => check next stage
    sub $s4, $s4, 1
    beqz $s4, win
    sub $t2, $t2, 1
    beq $t2, $zero, checkingHorizontal_Start
    j MoveLeftLoop
  checkingHorizontal_Start:
    add $s3, $s3, 4
  checkingHorizontal_Start1:
    add $s3, $s3, 4
    li $s4, 4         # $s4 is the counter for win-condition
    li $t2, 4         # $t2 is the counter for loop
  checkingHorizontal_Loop:
    lw $t0, playercheck     # $t0 have player1 piece
    lw $t1, ($s3)       # $t1 have the UUT piece
    bne $t1, $t0, checkFirstDiagonal  # if $t0 != $t1 => check next stage
    #if their equal, we update counters
    add $s3, $s3, 8       # jump to next Col
    #Win condition when $s4 = 0
    sub $s4, $s4, 1
    beqz $s4, win
    ############################
    sub $t2, $t2, 1
    beq $t2, $zero, checkFirstDiagonal
  j checkingHorizontal_Loop
```



Hình 3: Winning by horizontally

4.3 Checking first diagonal results

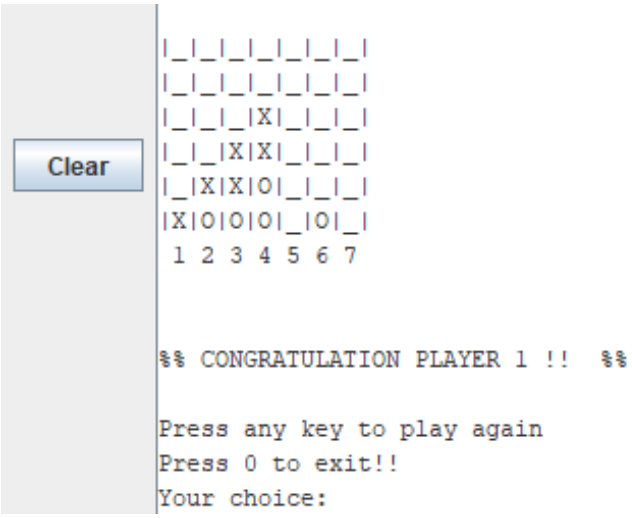
The algorithm is shown below

- First, get the address of the checking position.
- Then I will save a second variable similar to the horizontal check, but this time the variable will have to add a value of 64 (to jump to the first char in the next row). This time I will want to check all of the first row as well.
- Next, I'll give the variable the same jump as the horizontal test, but this time it's a four-fold diagonal jump to the left. The newly saved variable will let me know if I'm in the first row or not by comparing whether the address of the variable being checked is larger than the variable.
- The value that will be subtracted will be 72 (64 bytes for moving back 1 row and 8 bytes for 1 column).
- Finally, I'll give the same cross-down check diagonally to the right down as the horizontal. Each check move adds a value of 72.

MIPS code implementation

```
checkFirstDiagonal:
    la $s2, board         # $s2 will have address of the first char
    add $s2, $s2, Row_byte        # $s2 represent the edge of first row
    add $s3, $s0, $zero       # $s3 have the address of the checking position
    li $s4, 4         # $s4 is the counter for win-condition
    li $t2, 4         # $t2 is the counter for loop
  MoveCrossFirstDiagonalLoop:
    blt $s3, $s2, checkingFirstDiagonal_Start1  # if the checking position is at the first row
    sub $s3, $s3, firstCross      # Move to the left-up cross

    lw $t0, playercheck       # $t0 have player1 piece
    lw $t1, ($s3)         # $t1 have the UUT piece
    bne $t1, $t0, checkingFirstDiagonal_Start   # if $t0 != $t1 => check next stage
    sub $s4, $s4, 1
    beqz $s4, win
    sub $t2, $t2, 1
    beq $t2, $zero, checkingFirstDiagonal_Start
    j MoveCrossFirstDiagonalLoop
  checkingFirstDiagonal_Start:
    add $s3,$s3, firstCross       # $t2 is the counter for loop
  checkingFirstDiagonal_Start1:
    li $s4, 4         # $s4 is the counter for win-condition
    li $t2, 4
  checkingFirstDiagonal_Loop:
    lw $t0, playercheck     # $t0 have player1 piece
    lw $t1, ($s3)       # $t1 have the UUT piece
    bne $t1, $t0, checkSecondDiagonal # if $t0 != $t1 => check next stage
    #if their equal, we update counters
    add $s3, $s3, firstCross    # jump to next cross
    #Win condition when $s4 = 0
    sub $s4, $s4, 1
    beqz $s4, win
    ############################
    sub $t2, $t2, 1
    beq $t2, $zero, checkSecondDiagonal
  j checkingFirstDiagonal_Loop
```



Hình 4: Winning by first diagonal

4.4 Checking second diagonal results

The algorithm is shown below

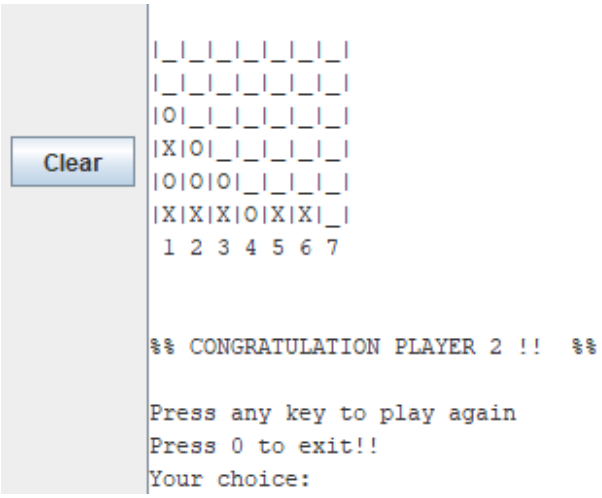
- The algorithm is just the same as the first diagonal
- Only the value to add and subtract is different since when want to check another cross. The value is 56 (64 bytes for a row but move back 8 bytes for a column).

Please take a look of my MIPS code

```
    checkSecondDiagonal:
        la $s2, board          # $s2 will have address of the first char
        add $s2, $s2, Row_byte         # $s2 represent the edge of first row
        add $s3, $s0, $zero        # $s3 have the address of the checking position
        li $s4, 4          # $s4 is the counter for win-condition
        li $t2, 4          # $t2 is the counter for loop
      MoveCrossSecondDiagonalLoop:
        blt $s3, $s2, checkingSecondDiagonal_Start1 # if the checking position is
    at the first row
        sub $s3, $s3, secondCross      # Move to the left-up cross

        lw $t0, playercheck        # $t0 have player1 piece
        lw $t1, ($s3)          # $t1 have the UUT piece
        bne $t1, $t0, checkingSecondDiagonal_Start    # if $t0 != $t1 => check
    next stage
        sub $s4, $s4, 1
        beqz $s4, win
        sub $t2, $t2, 1
        beq $t2, $zero, checkingSecondDiagonal_Start
        j MoveCrossSecondDiagonalLoop
      checkingSecondDiagonal_Start:
        add $s3,$s3, secondCross         # $t2 is the counter for loop
      checkingSecondDiagonal_Start1:
        li $s4, 4          # $s4 is the counter for win-condition
        li $t2, 4
      checkingSecondDiagonal_Loop:
```

```
25          lw $t0, playercheck     # $t0 have player1 piece
26          lw $t1, ($s3)       # $t1 have the UUT piece
27          bne $t1, $t0, End_check     # if $t0 != $t1 => check next stage
28          #if their equal, we update counters
29          add $s3, $s3, secondCross   # jump to next cross
30          #Win condition when $s4 = 0
31          sub $s4, $s4, 1
32          beqz $s4, win
33          ############################
34          sub $t2, $t2, 1
35          beq $t2, $zero, End_check
36      j checkingSecondDiagonal_Loop
```



Hình 5: Winning by first diagonal

5 Ending

This is the ending of my assignment report. If you have a problem downloading my source code, please use this GitHub link to download it.

https://github.com/HyHyZhaLee/CA_Project

Thank you for your time reading my report.

————END————