HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY

---------***----------

FINAL PROJECT REPORT

COMPUTER ARCHITECTURE

CODE: CC2007

FOUR IN A ROW

(using MIPS)

Student: TRẦN NGUYỄN GIA PHÁT

ID: 2153681

Cohort: CC01

Semester: 221

Academic year: 2022-2023

Lecturer: BĂNG NGỌC BẢO TÂM

Ho Chi Minh City, December 2022

TABLE OF CONTENT

I. OVERVIEW

1.1. Gameplay

It is the goal of the game to connect four of your tokens in a line in a 6x7 grid. All directions (vertical, horizontal, diagonal) are allowed. Players take turns putting one of their tokens into one of the seven slots. A token falls down as far as possible within a slot. The game ends immediately when one player connects four tokens.

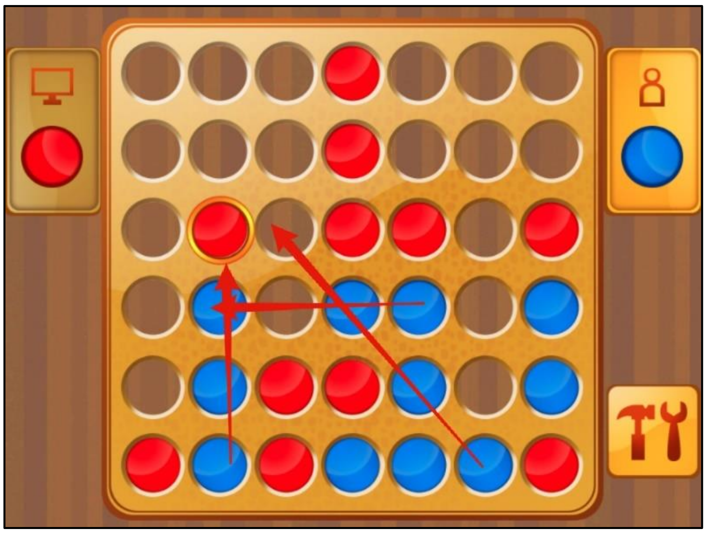


Figure 1: Four in a row board game

1.2. MIPS as an assembly language

The assembly language of the MIPS processor is referred to as MIPS assembly language. MIPS stands for Microprocessor without Interlocked Pipeline Stages.

Because many embedded systems run on the MIPS processor, learning the MIPS assembly language is quite beneficial. Learning how to code in this language allows for a more in-depth grasp of how these systems work at a lower level.

II. PREREQUISITES

2.1. Data storage

First, we must store the playing board in a 6x7 matrix for constant access time. This can be done with ease using a 2D array in higher-level languages. However, in MIPS assembly, we will use a 1D array with the same number of cells but with a different indexing approach. This method is called row-major order with the following formula:

$$array2D[i][j] = array1D[i * M + j]$$

where,

M is the number of rows, **i** and **j** are row and column index in 2D array.

i\j	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	7	8	9	10	11	12	13
2	14	15	16	17	18	19	20
3	21	22	23	24	25	26	27
4	28	29	30	31	32	33	34
5	35	36	37	38	39	40	41

Figure 2: Indices of 2D array alternative using row-major order

2.2. Flowchart

In most games, there will be a main while loop in which the game events will be processed. The game loop can only be stopped when the win conditions have been satisfied. We will apply the same approach to this project. Below is the flowchart of how this game should operate:

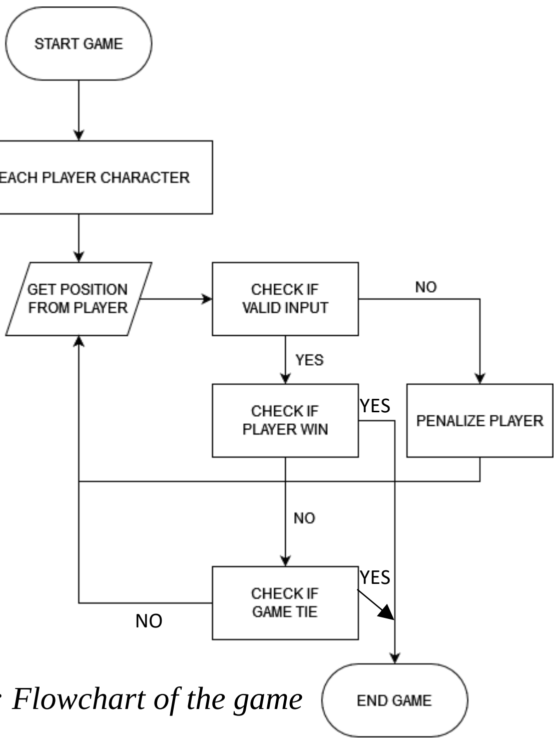


Figure 3: Flowchart of the game

2.3. Start game

Before the main loop can be called, we first need to collect some of the players' data and initialize essential game values, such as the chosen token, the number of remaining undos and invalid input of each player and the available slots. We will store them in the registers for later access.

Moreover, the program will print out game rules and ask the player to choose either X or O as their character. Player 1 will always go first.

2.4. Error handling

The program will only read user input as a single character to handle unexpected errors. That means the following process will automatically be called after the user presses any key. MIPS then reads the typed key as an ASCII value (in decimal) stored in register $v0. The program will then compare whether the given value is the valid input for that action or execute the error handling code.

2.5. While loop

In the while loop, we will have the following actions:

1. Get input and place player 1's token in the desired position. (player_go)
2. Print the grid. (print_matrix)
3. Check if player 1 wins. (check_win_player)
4. Check if there is a tie. (player_tie)
5. If player 1 has positive undos remaining, ask them whether they would like to retake their move. (player_undo)
6. Perform the same actions from 1. to 5. but for player 2.
7. Loopback.

III. FUNCTION EXPLANATION

3.1. Print the grid (print_matrix)

Using nested for loop, we iterate through every cell by row-major order, load the data stored in that byte and print it to the console. Empty cells will be initialized by the '*' character, while occupied cells are stored as 'X' or 'O'. The array size will be 42 bytes as each character in MIPS take up 1 byte of memory.

3.2. Place the player's move (player_go)

Argument: $a0 (used to identify the current player)

First, we will check if the input is valid from 1 to 7 (49 to 55 in ASCII code). If this is not the case, we penalize the current player by decreasing their trials by one and asking for input again.

Else, we check if the bottommost cell with the input column index is occupied, by adding 35 (five rows multiply seven columns each row) after subtracting 48 (converting ASCII value to decimal), finally adding the base address stored earlier in the register.

To identify which player is taking a turn, we compare $a0 to see if it holds player 1's token.

Now we load the character of the above address. If the character is "*", the slot is not occupied yet, and we can safely place the player's token. If not, we subtract the address by 7, which is the cell of the above row in the same column. We then check if it is a valid address (compare if it is lesser than the base address), then from the comparison result, decide to place the token or penalize the player.

We also subtract the number of slots stored at the register by 1. This will be used to check if the game is a tie later.

3.3. Undo move (player_undo)

Argument: $a0 (used to identify the current player)

After the player has placed their token, we check if the player has positive undo moves. If yes, we display a message asking them whether they would like to undo their move. We read the input as a single character; if ‘y’ or ‘Y’ is typed, we undo their last move, subtract their chances by one, or we continue to the other player’s turn.

In the player_go function, each time we place the token, we save its location to a register. Then we store back ‘*’ at the last inserted address and print the grid again.

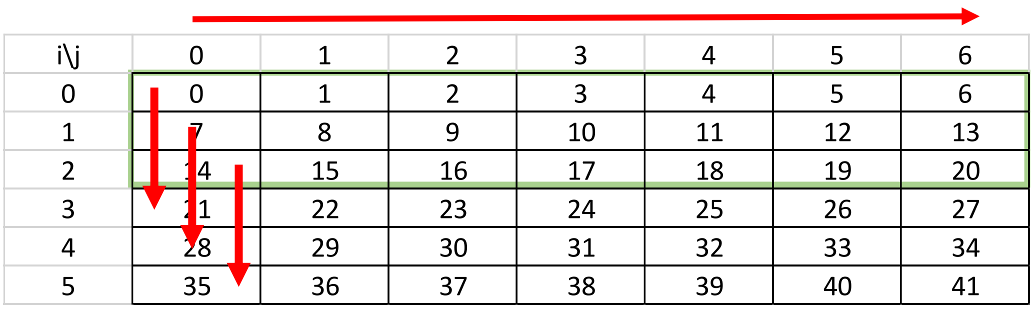
3.4. Check for a win (check_win_player)

Argument: $a0 (used to identify the current player)

At any moment, if four same tokens are lined up in a vertical, horizontal or diagonal direction, the player holding that token type will win the game. Therefore, we need to check all four possible directions.

1. Vertical check:

We will check the grid top-down, iterate through all columns and iterate only to row 2th to avoid index out of bound. (memory error). See the following picture:



i\j	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	7	8	9	10	11	12	13
2	14	15	16	17	18	19	20
3	21	22	23	24	25	26	27
4	28	29	30	31	32	33	34
5	35	36	37	38	39	40	41

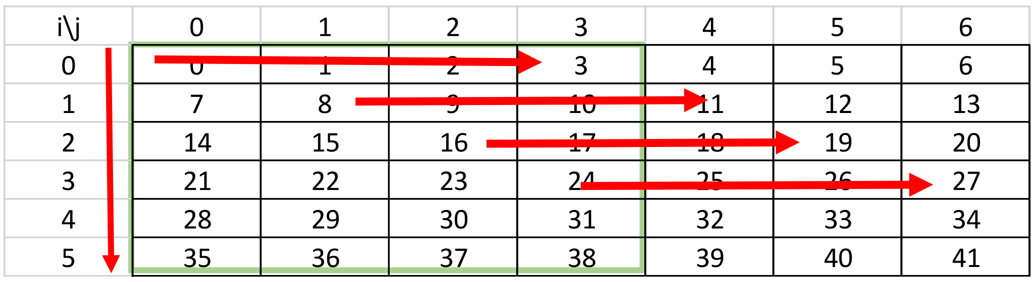
Figure 4: Veritcal check for winning

This involves a nested for loop; the outer loop index will go from 0 to 2, while the inner will go from 0 to 7 at each iteration. If all four consecutive cells in any row contain the same token, that token’s player wins. We continue with the next inner iteration if at least one is not like the others.

Accessing the lower row cell can be done by adding 7 to the current address. Iterable zone are marked with green borders in the above figure.

2. Horizontal check:

The idea is as same as vertical check but with a different iterable zone. See the following picture:



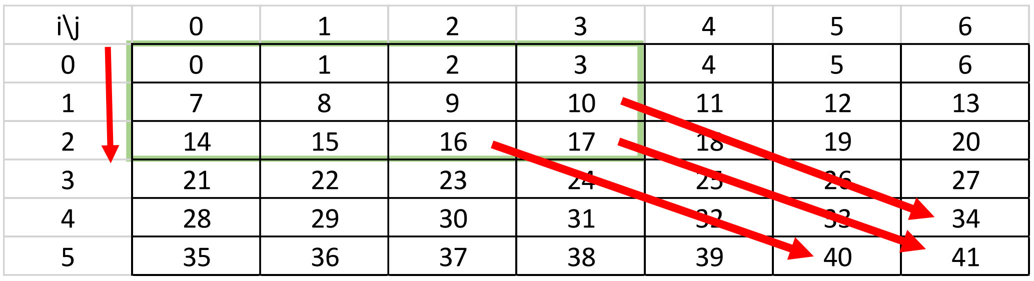
i\j	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	7	8	9	10	11	12	13
2	14	15	16	17	18	19	20
3	21	22	23	24	25	26	27
4	28	29	30	31	32	33	34
5	35	36	37	38	39	40	41

Figure 5: Horizontal check for winning

The outer for loop will go from 0 to 5, while the inner will iterate from 0 to 3. The lower next right cell can be accessed by adding 1 to the current address.

3. Main diagonal check (top left to bottom right):

The idea is as same as mentioned checks by using nested for loop. See the below figure:



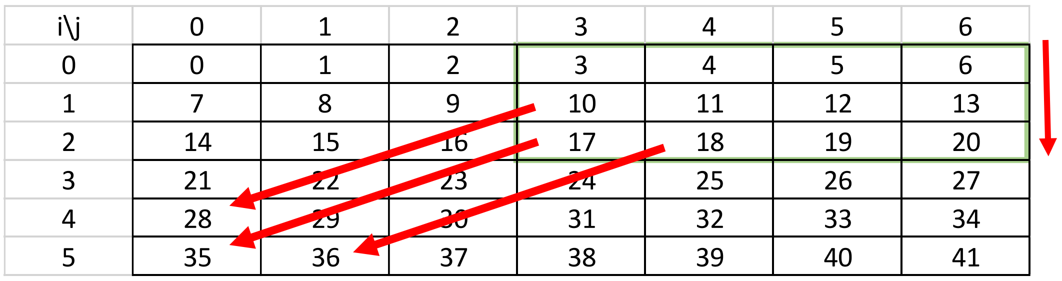
i\j	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	7	8	9	10	11	12	13
2	14	15	16	17	18	19	20
3	21	22	23	24	25	26	27
4	28	29	30	31	32	33	34
5	35	36	37	38	39	40	41

Figure 6: Diagonal check for winning

To access the lower cell to the right, add 8 to the current address.

4. Anti diagonal check (top right to bottom left):

The idea is as same as mentioned checks. See the below figure:



i\j	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	7	8	9	10	11	12	13
2	14	15	16	17	18	19	20
3	21	22	23	24	25	26	27
4	28	29	30	31	32	33	34
5	35	36	37	38	39	40	41

Figure 7: Anti diagonal check for winning

Accessing the lower cell to the left can be done by adding 6 to the current address.

3.5. Check for a tie (check_tie)

After each valid player's move, if the number of slots saved in the register at the early game reaches 0, no more moves can be made. The game declares a draw and ends the program.

Source code is available at:

https://github.com/Zaphat/Four_In_A_Row_mips