Collin Tran

CS 2340.004

Nhut Nguyen

Project Report

Program Description:

The program is a recreation of the popular game “Connect 4” in MIPS. “Connect 4” is a game that takes place on a vertical 7 x 6 board. In it, players attempt to connect 4 of their pieces before their opponent can. In order to “place” pieces on the board, the player selects a column to drop their piece into. Since the board is vertical, pieces drop to the bottommost available space in the selected column; if a column already has a piece in it, the dropped piece stacks on top of it. Additionally, columns that have been completely filled are not allowed to take in any more pieces. 4 pieces can be connected either vertically, horizontally, or diagonally. If the board completely fills before anyone can connect 4 of their game pieces, the game ends in a draw.

All the rules that apply to the real-life variant of “Connect 4” also apply to this MIPS recreation of it. For example, floating pieces aren’t allowed since pieces naturally fall to the bottom of the board, so in the program, floating pieces also aren’t allowed. Although “Connect 4” is played between two people, the program pits the player and the computer against each other. The computer has no winning strategy, and for the most part, randomly selects columns to place pieces in.

The board starts completely blank, and the player always has the first move. The game board is represented using a 2D array that contains characters. The player’s pieces are represented by “Y”, while the computer’s pieces are represented by “R”, meant to mimic the yellow and red pieces that appear in the actual game. When the game begins, the player is prompted to pick a column from 0-6 with 0 being the left column and 6 being the right column. Picking a piece outside of this range, or entering a non-integer character causes the program to prompt the player for an integer within the range once more. Furthermore, selecting a column that has already been filled causes the program to prompt the player to choose a column that isn’t full. Once the player has selected a valid column, their piece is “dropped” in, appearing at the bottommost available space in the column. Then, the computer places its own piece onto the board, with each of the computer’s moves going through their own validation checks. The details of these checks will be discussed in the algorithm section of this report. After either the player or the computer places a piece, the program checks for if 4 pieces have been connected. If it finds that 4 pieces have been connected, the game ends, and the program displays which player has won. The program continues to let the player and computer place pieces on the board until either player wins, or the game ends in a draw due to the board filling completely.

Challenges Faced:

Since I worked on this project alone, one of the largest obstacles was finding all the information by myself, and then implementing it. I didn’t have anyone else to source ideas from, and I also didn’t have

anyone to rely on when it came to programming, but I chose to work alone since I wanted to challenge myself. Another problem I faced was accessing the pieces in the game board. From the beginning, I knew that the game board would be a 2D matrix, but I didn't know how I was supposed to load and store anything in it. In C++ or Java, you'll see matrix[i][k] used to access elements as specific indexes of a 2D matrix, but that doesn't exist in MIPS. Eventually, I found out how to do it, which will be discussed in the third section of this report, but matrix operations were undoubtedly confusing and exhausting at first. The final major challenge I faced was debugging the program. Since registers aren't used in HLLs, debugging programs written in them are a lot easier for me. While I was programming this, there were many times where I had a bug due to having registers in the wrong order in operations. Sometimes, I just ended up using the wrong registers since I always having trouble tracking them on this large of a scale. However, utilizing register standardization made what I was doing a lot clearer to myself.

What I learned:

One of the major things I learned was how to do access the elements inside a 2D array. From what I learned, a 2D array is a normal array that has been split into multiple layers.

For example, the array:

1 2 3 4 5 6

can be represented as the 2D array:

1 2 3

4 5 6

However, to incorporate row and column indexes, another step must be taken. Using the address equation for a 2D array (address = baseAddress + (rowIndex * rowSize + colIndex) * DATASIZE), all of the elements in the array can be accessed using row and column indexes. The base address is the address where the array label is located, the row and column indexes are the indexes of the element you want to access, and the row size is the number of elements that can fit in one row. "Datasize" is the size of the data types used in your array. For example for words, datasize would be 4. Using all of these elements with the equation provided above, the address of the element you want to access can be obtained.

Another thing I learned was how to use register standardization. For the first time, I used the argument registers ($a) to pass arguments to different subroutines, and I used the $s registers to save values that I wanted to use throughout the life of the program. I also used the return registers ($v) to return values after exiting a subroutine. Speaking of subroutines, I also learned how to incorporate subroutines into my programs using the jal and jr commands.

I also learned how to traverse through strings. Since I wanted to prevent the user from crashing the game by inputting non-integer values when using the syscall for reading integers, I decided to use the

syscall for reading strings, check the string to make sure that there weren't any non-integer values, and then convert the string into an integer. Accessing strings at specific indexes is like accessing an array in MIPS, so it was easy to get used to. However, converting a string of integers into an actual integer was a lot more difficult to do. To do this, you need an accumulator, which is a register that will hold the final value of the integer. When converting, you need to get an integer character from the string and subtract 48 from it, multiply the value in the accumulator by 10, and then add the difference that was calculated earlier to the accumulator. This cycle loops until all of the characters of the string have been accessed and converted.

Outside of programming, I learned the value of planning out how I want to create a programming before programming it. Weeks before I started programming, I was already thinking about the logic of the program, particularly how I was going to check for 4 pieces being connected. Consequently, when I did begin programming, I just had to put my ideas into place rather than thinking about the logic while I was programming. This saved me a lot more time than I would've imagined, and it also saved me from stressing out, which is a problem I usually have when coding.

Algorithms:

As explained in the program description, the main function of the program prompts the player for a valid input, places their piece on the board, and then checks to see if 4 player pieces are connected. If no connection was found, the program then generates the computer's move, places it on the board, and checks if 4 computer pieces are connected. If no connection was found, the program checks to see if the game board has been completely filled, signifying a draw. If the board isn't completely filled, the program re-loops to the player actions, and the same process occurs until either the game ends in a draw or one of the players wins the game. I'll be going into more depth into the algorithms used in the program.

Printing the game board:

Before the player's turn and after the player or computer places a piece, the game board is printed to show the player its current state. The function to print the board uses nested for loops, much like you would see in a program that uses HLLs. The outside loop is used to increment the row index, and the inner loop is used to increment the column index. Together, these loops can traverse the entire 2D array.

In the function, $t0 is used as the row index, and $t1 is used as the column index, with both being set to 0 at the beginning of the function. $t0 being used as the row index and $t1 being used as the column index is a standard throughout the program. Before entering the inner loop, the outer loop prints "|", which is used to represent the border of the board. Inside the inner loop, these series of commands are used to access the address of the element at the current indexes:

```
mul $t3, $t0, $s2          #$t3 = rowIndex * rowSize
```

```
add $t3, $t3, $t1          #$t3 = rowIndex * rowSize + colIndex
mul $t3, $t3, DATASIZE  #$t3 = (rowIndex * rowSize + colIndex) * DATASIZE
add $t3, $t3, $s0          #$t3 = baseAddress + (rowIndex * rowSize + colIndex) * DATASIZE
```

This same set of commands is used throughout the program whenever the 2D array needs to be accessed. After the element at the indexes has been obtained and printed, another "|" is printed, and the column index is incremented. The printing of the element at the specified indexes, the printing of "|", and the incrementation of the column index is repeated until the final element in the row has been accessed. Then, another "|" is printed to represent the border of the board, the row index is incremented, and the column index is set to 0. The program then jumps back to the top of the outer loop, and the process repeats until all rows have been printed sequentially.

Checking if the game board is full:

Since floating pieces aren't allowed in Connect 4, the program just needs to check if there are any empty spaces in the top row of the board to determine if the board has been completely filled. Since the row index is 0, we don't need to use the 2D array equation to access the elements, and we can instead access them as if we were accessing an ordinary array. The function scans the top row from left to right looking for a "_", as this represents empty spaces in the 2D array. If one is found, the function returns 0, signifying that the game board isn't full. If the function reaches the end of the row without having found a "_" it returns 1, signifying that the game board is full. Then, when returning to main, the program prints that the game has ended in a draw.

This check to see if the game board is full only needs to be done after both the player and the computer both place a piece. Since the game board has an even number of spaces (7 x 6 = 42), the game board can only be filled after both the player and the computer both take their turns.

Player move validation:

The string input by the player is first checked to see if it only contains integers. If it doesn't, the player is prompted to input a string only containing integers. If it does, the player's move is checked for validity. For a player's move to be valid, two conditions must be checked:

1. The column the player has chosen is within the range of the game board (0-6).
2. The column that the player has chosen isn't full.

The function first checks to see if the player's input is greater than or equal to 0, but less than or equal to 6. Since the string that the player input can only have integers, negative numbers will have already been eliminated before this step since they begin with "-". The function just needs to check if the integer is greater than 6. If it is, the input is out of range, and the player is prompted for another input that is in range, and 0 is returned. If the 1st condition has been met, the function then checks to see if the selected column is full. Much like checking to see if the board is full, to see if a column is full, we only need to see if the top of it is full (which is the column index of the specified column in the top row of the board). If the column is full, the function prompts the player to select a column that isn't full, and 0 is

returned, but if both conditions have been met, 1 is returned. If any of these conditions aren't met, the player is prompted to input another column, and the new input must meet both conditions.

Computer move and validation:

The generation of the computer's move and its validation are done in the same function. For the computer's move to be valid, it must meet the same two conditions that the player's move must. To meet the first condition, a random integer from 0-6 is generated. Then, much like validating the player's move, the top of the column is checked to see if the column is full. If the column is full, another random integer is generated, and the process repeats itself, but if it isn't full, then that column is used as the computer's move. Since this has the possibility to infinitely loop or at least loop for a large amount of time (although the chances are very low), after 5 random integers have been generated, if an empty column still hasn't been found, the function begins to traverse from the right side of the board, checking to see which column is empty. The first empty column it encounters is selected as the computer's move.

Placing player/computer pieces:

The same function is used to place player and computer pieces. This function is used after the validation of the player's move or the computer's move. In the actual game of Connect 4, when a player places their piece, they drop their piece through the top of the column they want it in, and it falls to the bottommost available space; if there are pieces below it, it'll stack on top of them. To mimic this behavior, the function searches for the bottommost available space in the row specified by the player or the computer.

For the function to know whether to place a player piece (Y) or a computer piece (R), an argument is passed to it through $a2. If $a2 is 0, the function knows to place a player piece, but if $a2 is 1, the function places a computer piece instead. The value of $a2 is set at the start of the player and the computer turns.

At the beginning of the function, the column index is set to either the player's selected column or the computer's selected column, depending on whose turn it is. The row index is set to 5, representing the bottom row of the board. Using the equation, the address of the bottom space of the column is obtained. The function then checks to see if the space is a "_", an empty space. If it is, the function loads the player or computer piece into the 2D array at the specified indexes. These indexes are then saved into $s6 and $s7, the saved row index and the saved column index respectively. These are used later for checking to see if 4 pieces have been connected. However, if the space isn't a "_", the row index is decremented by 1, and the check happens again. This happens until an available space is found. We don't need to worry about no spaces being available in the column since the input has already been validated and no floating pieces are present using this approach of placing them.

Checking for connections:

The last algorithm used in the program would be the one to check for game-winning connections. Much like the piece placing function, the program uses one function to check for connections for both the player and the computer. For this function, the $a2 register is also used to tell the function whether to check connections made by the player's pieces or the computer's pieces. The function assumes that there are four possible ways to win the game:

1. Vertical connection
2. Horizontal connection
3. Diagonal connection (from bottom left to top right direction)
4. Diagonal connection (from bottom right to top left direction)

To avoid checking the whole board for a connection after every input, the function also operates on the assumption that all game winning connections can be derived from the latest input from the player or the computer; since the game will end the moment either player places a piece that causes 4 of their pieces to be connected, all connections can be derived from the piece that the player just placed, which is where the values in $s6 (saved row index) and $s7(saved column index) become necessary. This means that we only have 4 conditions to check for when finding if a connection has been made:

1. Vertical connection in the column of the piece just placed.
2. Horizontal connection in the row of the piece just placed.
3. Diagonal connection (bottom left to top right) in line with the piece just placed.
4. Diagonal connection (bottom right to top left) in line with the piece just placed.

Additionally, that means we must check for a connection after the player or computer place their piece on the board.

<u>Checking for vertical connections:</u>

Checking for vertical connections is the easiest condition to check for and is the first one checked for by the function. Due to the nature of the game, when a vertical connection is made, it starts from the very top of the column and extends downwards since there are no floating pieces in Connect 4. Therefore, we only need to check the first four pieces from the top of the column a piece was just placed in to see if there is a vertical connection. If one exists, 1 is returned, and the program prints who won the game. If one isn't, the function then checks for a horizontal connection.

<u>Checking for horizontal connections:</u>

The function next checks for the existence of a horizontal connection. Since a horizontal connection can happen anywhere in the row while a vertical connection can only happen at the top of the column, the entire row of the piece that was just placed must be checked for a connection. To do this, the function starts from column index 0 and checks if the first 4 pieces moving right are the same player pieces. If they are, a horizontal connection has been found. If they aren't, the function increments the column index by 1, and reruns the test. This continues until a connection is found, or until column index becomes greater than 3 since at this point, a horizontal connection is no longer possible.

Example:

Step 1: first 4 pieces from the left are checked

_ _ _ _ _ _ _

Step 2: increment column index by 1, recheck

_ _ _ _ _ _ _

(Continues until a match is found or column index is greater than 3. The check for when column index is equal to 3 is the last possible check for a horizontal connection.)

If a horizontal connection is found, 1 is returned, and the program prints who won the game. If one isn't, the function then checks for the first diagonal connection.

Diagonal connection (bottom left to top right direction)

If a horizontal connection is found, the function checks for a diagonal connection from the bottom left to top right direction with respect to the piece just placed. To do this, the function first moves down and left from the piece just placed until it reaches an edge of the gameboard.

Example:

A piece was just placed in the space highlighted yellow:

_ _ _ _ _ _ _

_ _ _ _ _ _ _

_ _ _ _ _ _ _

_ _ _ _ _ _ _

_ _ _ _ _ _ _

_ _ _ _ _ _ _

The function then traverses down and left, reaching the position highlighted green before beginning its search

_ _ _ _ _ _ _

_ _ _ _ _ _ _

_ _ _ _ _ _ _

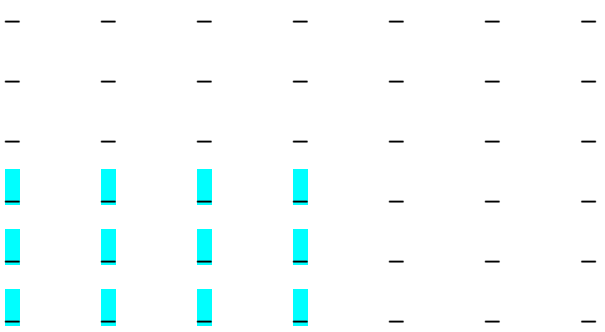
_ _ _ _ _ _ _

_ _ _ _ _ _ _

_ _ _ _ _ _ _

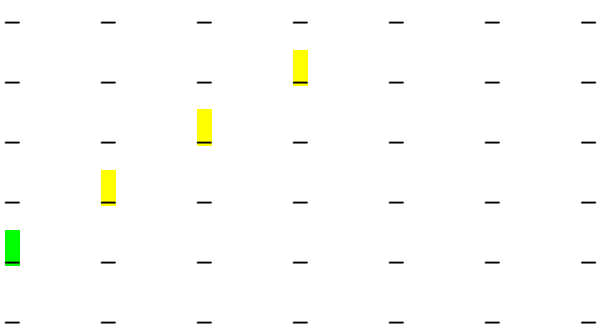
This allows the function to check for every possible connection with respect to the piece just placed in this particular direction. Using this approach, we can see that if the search position is outside of certain range, a diagonal connection becomes impossible:

If the search position is outside of the area highlighted blue, this particular diagonal connection is impossible.



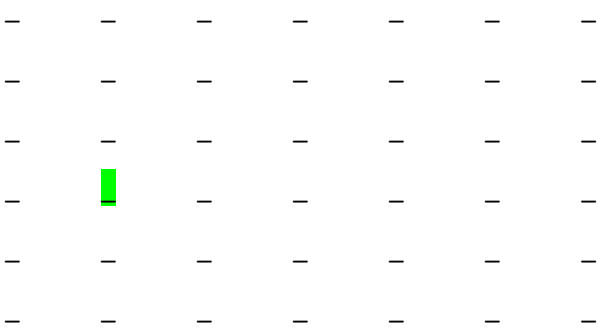
Like the way we checked for a horizontal match, we start at the search position and check the first 4 pieces of the line to see if they are the same player piece, but instead of moving just horizontally, we move up and to the right.

The first four pieces in the line (including the search position highlighted green) are checked for a connection.

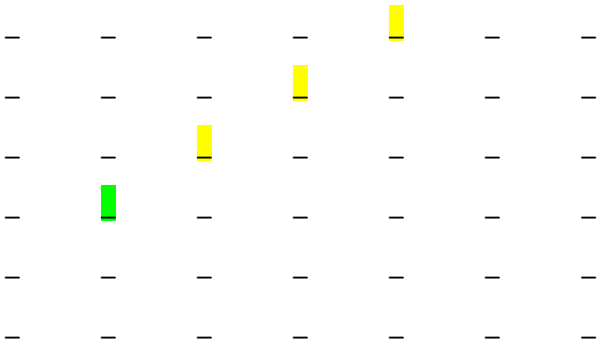


If a connection is found, 1 is returned, and the program prints who won the game. If a connection isn't found, we decrement the row index and increment the column index, moving the search position 1 space up and 1 space to the right:

If a connection isn't found, the search index is moved 1 space to the right and 1 space upwards:



The function then runs the connection check again to see if the next 4 pieces are the same:



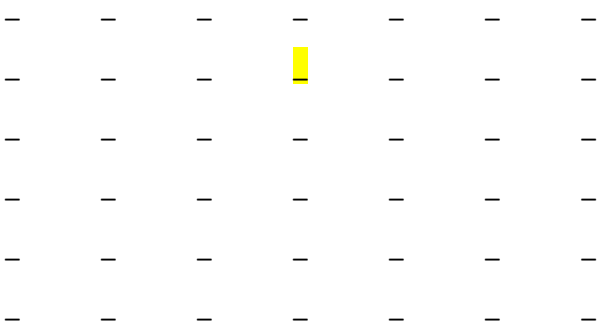
This continues until a connection has been found, or until the search position goes outside of the area for possible diagonal connections to occur. If a connection isn't found, the function checks for the second diagonal condition.

Diagonal Connection (bottom right to top left direction)

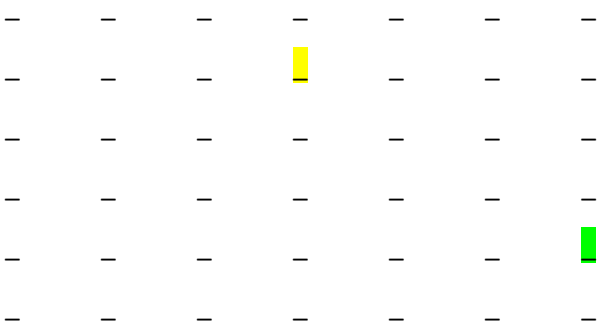
This is the last condition checked by the function. Logically, it works the same as the first diagonal condition, but instead of searching from the bottom left of the board and moving up and to the right, it starts searching from the bottom right of the board and moves up and to the left.

Example:

A piece was just placed in the space highlighted yellow:

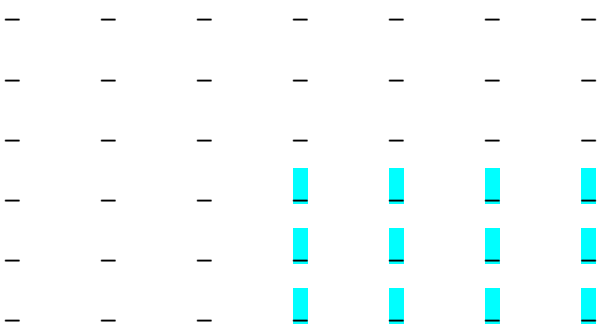


The function moves down and to the right until it reaches an edge of the board to begin its search:

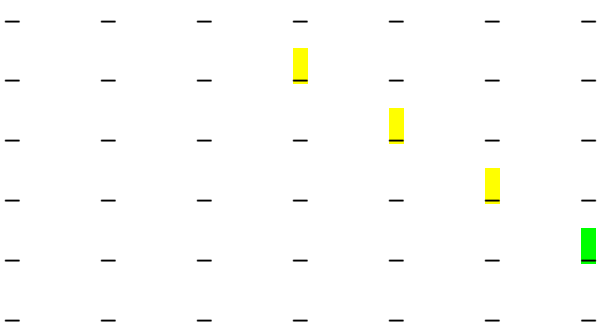


Just like with the first diagonal condition, if the search position is outside of a certain range, a diagonal connection is impossible:

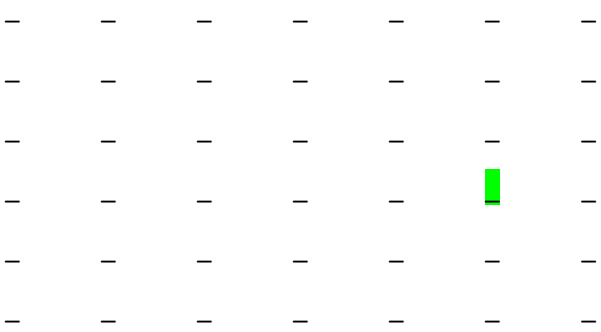
If the search position is outside of the area highlighted blue, this particular diagonal connection is impossible:



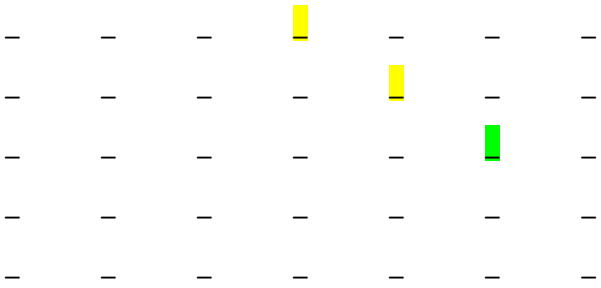
The function checks to see if the first 4 pieces in a line from the search position are the same:



If they are the same, a connection has been found. If they aren't the function decrements both the row index and column index of the search position, moving it up 1 and to the left 1:



The function then reruns the connection check:



Much like the first diagonal condition, this continues until a match is found or the search position goes outside the area for possible diagonal connections to occur. If a connection hasn't been found after this condition has been checked, 4 pieces haven't been connected yet. The program then allows the computer to place a piece on the board or checks for if a draw has occurred, depending on whose turn it currently is.