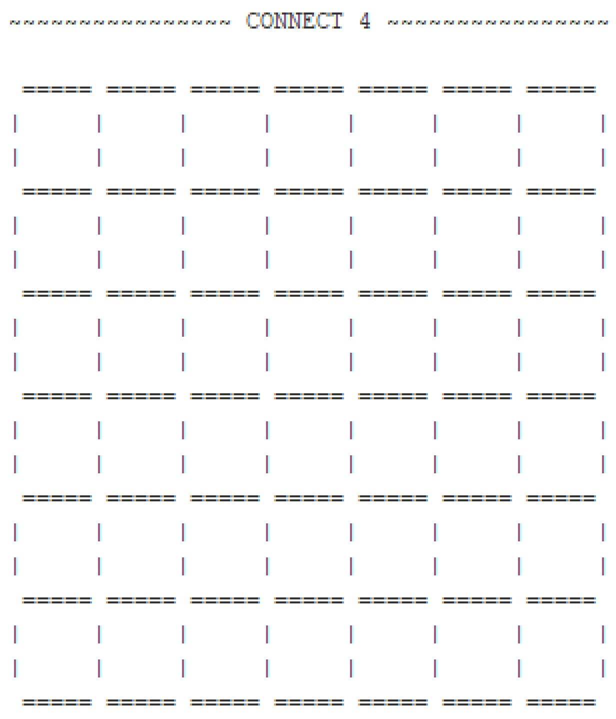
Connect 4:
A Virtual Text-Based Version

Introduction:

Connect 4 has been around since 1979, and has been a well-loved family game throughout the recent few decades. It is a very simple game to learn but can require a good amount of strategy depending on who you go up against. The rules are simple: drop your designated game piece down one of the 7 available columns, and try to get 4 of your pieces in a row, whether it be vertically, horizontally, or diagonally. The first to connect 4 in a row, is the winner.



```
~~~~~~~~~~~~~~~~ CONNECT 4 ~~~~~~~~~~~~~~~~

 ===== ===== ===== ===== ===== ===== =====
|     |     |     |     |     |     |     |
|     |     |     |     |     |     |     |
 ===== ===== ===== ===== ===== ===== =====
|     |     |     |     |     |     |     |
|     |     |     |     |     |     |     |
 ===== ===== ===== ===== ===== ===== =====
|     |     |     |     |     |     |     |
|     |     |     |     |     |     |     |
 ===== ===== ===== ===== ===== ===== =====
|     |     |     |     |     |     |     |
|     |     |     |     |     |     |     |
 ===== ===== ===== ===== ===== ===== =====
|     |     |     |     |     |     |     |
|     |     |     |     |     |     |     |
 ===== ===== ===== ===== ===== ===== =====
|     |     |     |     |     |     |     |
|     |     |     |     |     |     |     |
 ===== ===== ===== ===== ===== ===== =====
```

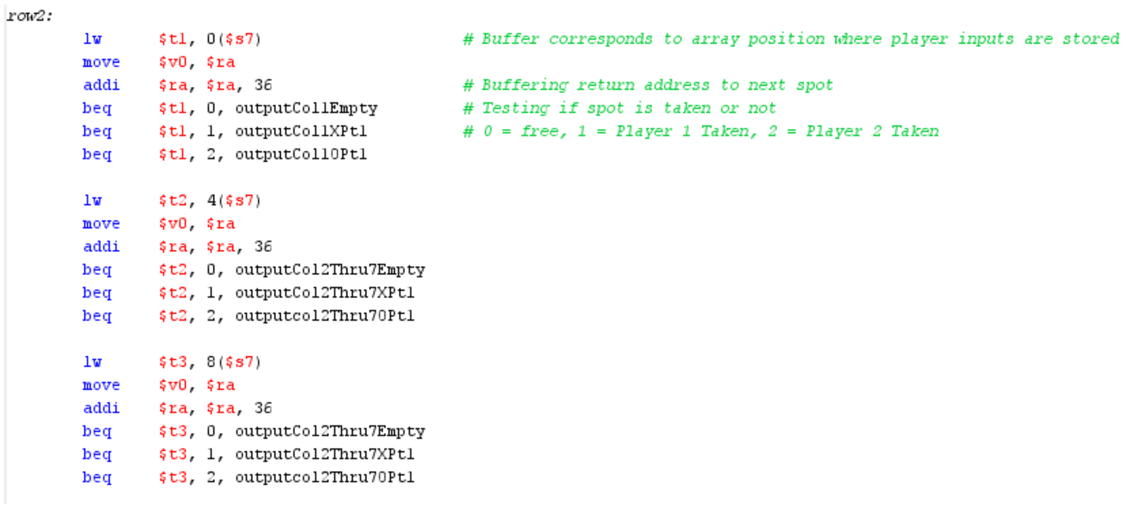
What I have created:

I took the concepts of this game, and programmed a simple text-based version. All of the rules and functionalities are the same, even gravity is simulated. The game is played the same exact way, with Player 1 making the first move. First to connect 4 in a row is deemed the winner. Validations are included to prevent unexpected behaviors from happening. Everything is user friendly, however the "winner" system is merit-based. Creating an autonomous "winner check system" that checks for EVERY possible winning move would be too laborious, repetitive, and resource costly. So, I decided to leave the winner declaration to the players themselves.

In-depth:

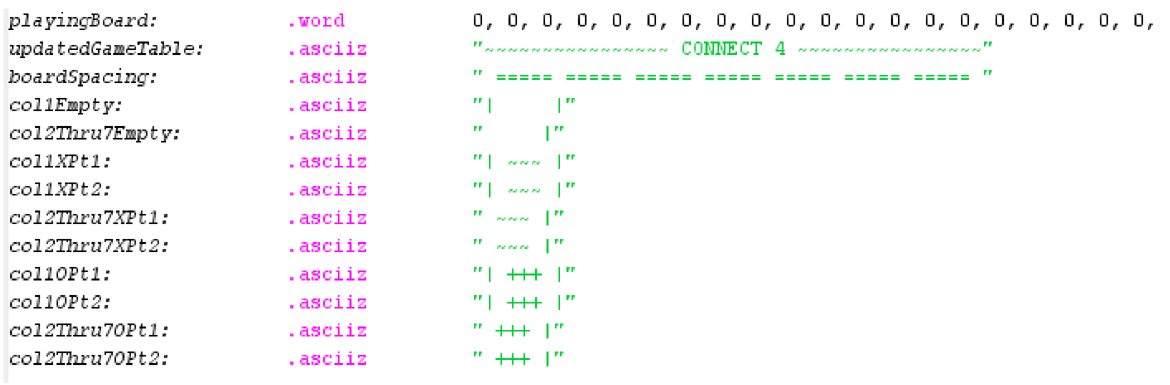
Real-Time Game Board Rendering:

The way this program functions, is by using “templates” in order to draw the game board after every move a player makes. Having whole premade board templates would mean I would have to supply nearly 109.4 **Quintillion** different board possibilities to accommodate all combinations of empty, player 1 taken, and player 2 taken spaces (based on the math of: 3^{42}, please correct me if I’m wrong). Well, I didn’t feel like spending time doing that, so I made “board element” templates, and created a “copy-paste” type algorithm.



```
row2:
        lw      $t1, 0($s7)                     # Buffer corresponds to array position where player inputs are stored
        move    $v0, $ra
        addi    $ra, $ra, 36                    # Buffering return address to next spot
        beq     $t1, 0, outputCol1Empty         # Testing if spot is taken or not
        beq     $t1, 1, outputCol1XPt1          # 0 = free, 1 = Player 1 Taken, 2 = Player 2 Taken
        beq     $t1, 2, outputCol1OPt1

        lw      $t2, 4($s7)
        move    $v0, $ra
        addi    $ra, $ra, 36
        beq     $t2, 0, outputCol2Thru7Empty
        beq     $t2, 1, outputCol2Thru7XPt1
        beq     $t2, 2, outputcol2Thru7OPt1

        lw      $t3, 8($s7)
        move    $v0, $ra
        addi    $ra, $ra, 36
        beq     $t3, 0, outputCol2Thru7Empty
        beq     $t3, 1, outputCol2Thru7XPt1
        beq     $t3, 2, outputcol2Thru7OPt1
```

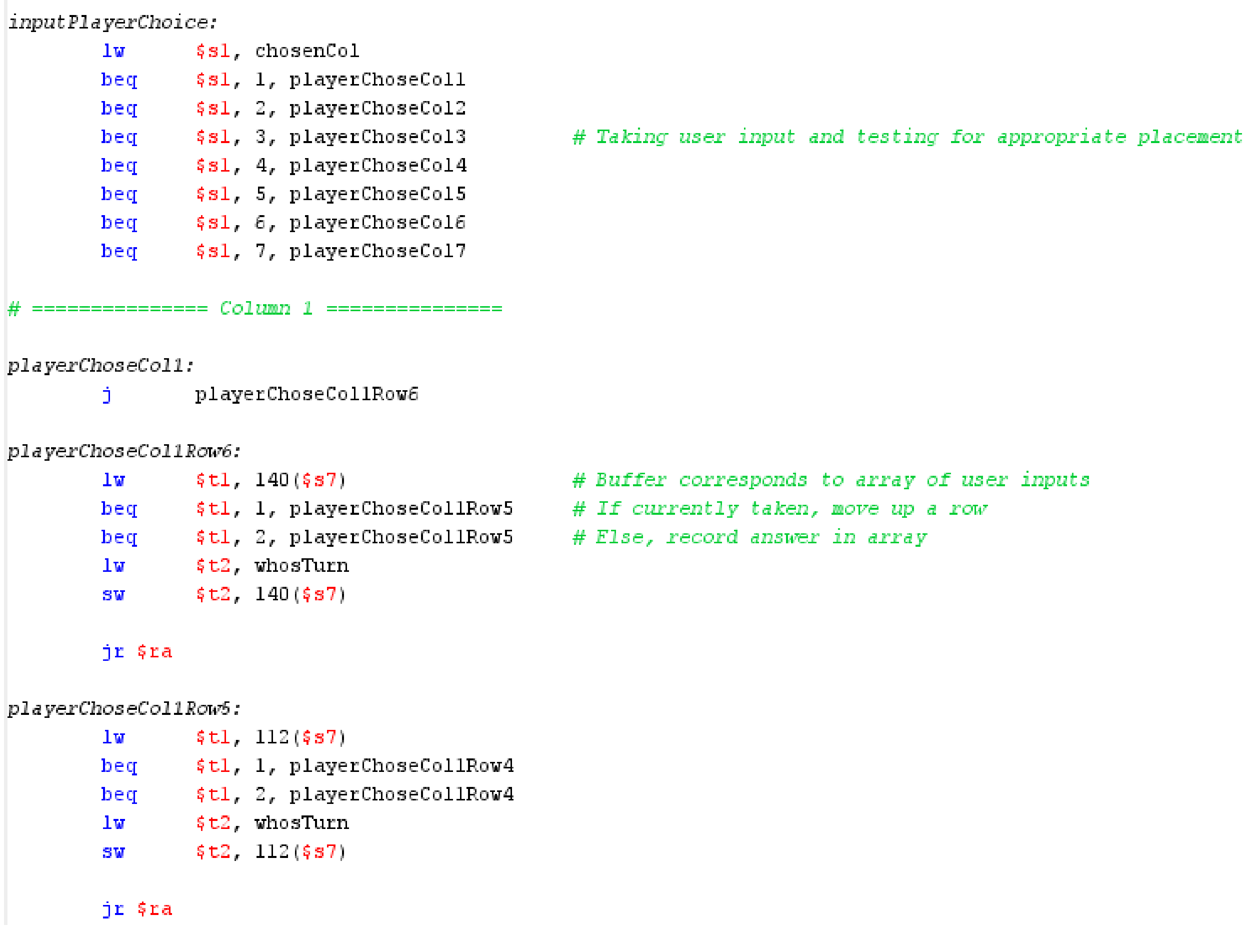
This algorithm looks into an array of integers, ranging between 0 and 2, picks out a predetermined spot, and tests the value to see whether it should place: an empty space, a space marked by Player 1’s game piece, or a space marked by Player 2’s game piece. The picture below is every copy-paste element that makes up the entire game board, including the array that stores the player’s moves.



```
playingBoard:           .word           0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
updatedGameTable:       .asciiz         "~~~~~~~~~~~~~~~~~ CONNECT 4 ~~~~~~~~~~~~~~~~~"
boardSpacing:           .asciiz         " ===== ===== ===== ===== ===== ===== ===== "
col1Empty:              .asciiz         "|      |"
col2Thru7Empty:         .asciiz         "      |"
col1XPt1:               .asciiz         "| ~~~ |"
col1XPt2:               .asciiz         "| ~~~ |"
col2Thru7XPt1:          .asciiz         " ~~~ |"
col2Thru7XPt2:          .asciiz         " ~~~ |"
col1OPt1:               .asciiz         "| +++ |"
col1OPt2:               .asciiz         "| +++ |"
col2Thru7OPt1:          .asciiz         " +++ |"
col2Thru7OPt2:          .asciiz         " +++ |"
```

Validation:

Now, the validation is just a whole bunch more copy-pasting, with minor tweaks. In accordance to how this game is played in real-life, game pieces cannot overlap, and they must fall as far down to the bottom of the column as possible. So, not only does validation have to check for existing game pieces, it must check for empty spaces where the game piece can fall down further. This is the simulated gravity I mentioned previously.



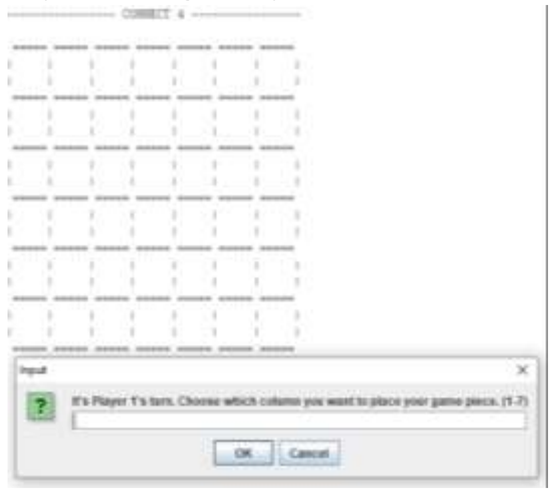
```
inputPlayerChoice:
	lw	$s1, chosenCol
	beq	$s1, 1, playerChoseCol1
	beq	$s1, 2, playerChoseCol2
	beq	$s1, 3, playerChoseCol3		# Taking user input and testing for appropriate placement
	beq	$s1, 4, playerChoseCol4
	beq	$s1, 5, playerChoseCol5
	beq	$s1, 6, playerChoseCol6
	beq	$s1, 7, playerChoseCol7

# =============== Column 1 ===============

playerChoseCol1:
	j	playerChoseCol1Row6

playerChoseCol1Row6:
	lw	$t1, 140($s7)			# Buffer corresponds to array of user inputs
	beq	$t1, 1, playerChoseCol1Row5	# If currently taken, move up a row
	beq	$t1, 2, playerChoseCol1Row5	# Else, record answer in array
	lw	$t2, whosTurn
	sw	$t2, 140($s7)

	jr $ra

playerChoseCol1Row5:
	lw	$t1, 112($s7)
	beq	$t1, 1, playerChoseCol1Row4
	beq	$t1, 2, playerChoseCol1Row4
	lw	$t2, whosTurn
	sw	$t2, 112($s7)

	jr $ra
```

Efficiency:

In efforts to maximize resource saving, there is an initial test before the player's choice actually gets tested spot by spot. This test is simply checking which column the player wishes to drop a game piece, so only that column is tested, which saves ~ 85.7% of resources used when checking every column needlessly. To save a little bit more, once a column is selected for testing, it starts with the very bottom row, and tests its way up towards the top, since the bottom rows are going to be filled more often than the top rows. So, as soon as the bottom most free spot of the column is found, that spot gets written over with the current player's move, marking it as theirs for future move testing. There is definitely room for more improvements in resource saving, but the project would have to be restructured to implement a good portion of other possible ideas.

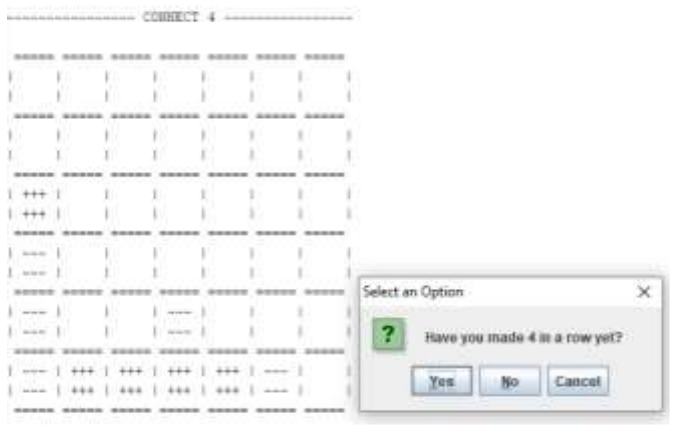
Gameplay:

Everything is straightforward (at least I hope so). You compile and run the program, being prompted with a couple of messages, a blank gameboard is drawn, and Player 1 being ready to make the first move. This continues back and forth between Player 1 and Player 2 until someone achieves in connecting 4 of their game pieces in a row. Thus, prompting the proper time in declaring a winner.


CONNECT 4

Input

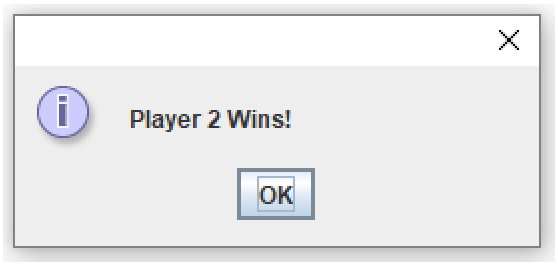
It's Player 1's turn. Choose which column you want to place your game piece. (1-7)

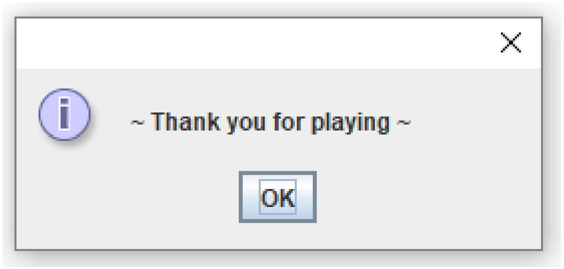

CONNECT 4

Select an Option

Have you made 4 in a row yet?

Once a winner has been declared, the game congratulates the victor, thanks those who played, and exits gracefully.


Player 2 Wins!


~ Thank you for playing ~

If the losing player wants a rematch, all that needs to be done, is reset or re-compile/run the program.

Program and Documentation by:

Christopher tePoele – 12/2/2018