Connect 4 Writeup - Lyell Read.

This Writeup is about the program connect4.asm, written by Lyell Read, included in this archive.

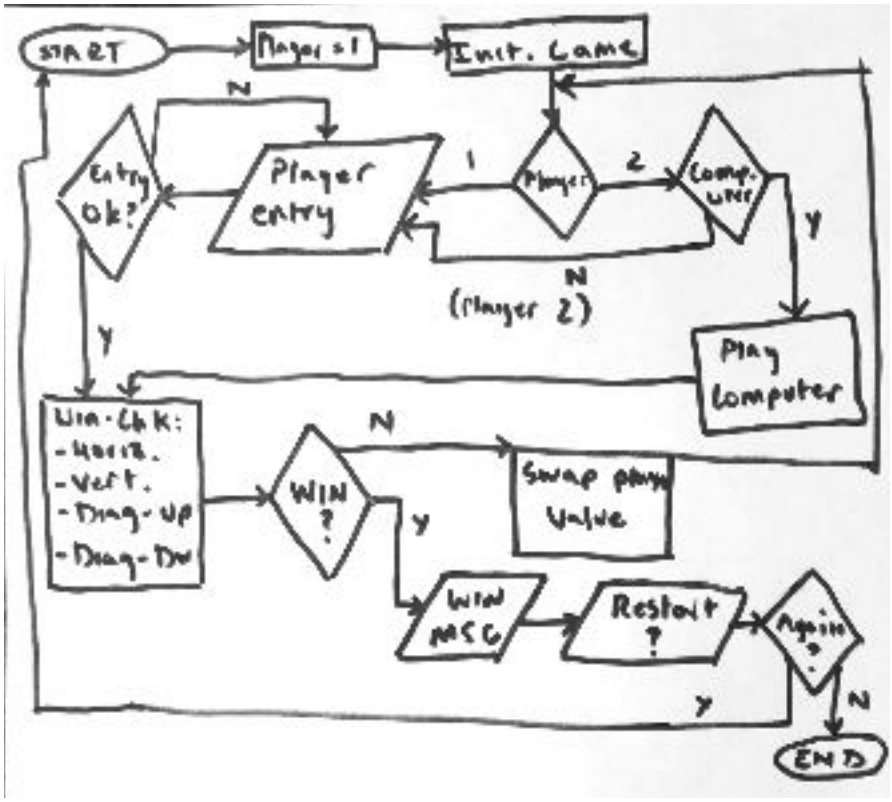
Planning

Planning this project took place in two stages. First, I contemplated the flow of the program and put together a very detailed (albeit quite messy) diagram that is featured below. It includes the points I questioned my ability to complete (in green pen) and included some variables I knew I would need to use. The second stage was working out (before I started the program) how I would clear up all those green pen marks - the things I was not sure I could complete. One by one, I found solutions to these issues, some more obvious than others. For example, on one end of the spectrum, I was unsure how to color text, until I read the chapter on the Irvine Library. Kip Irvine includes the SetTextColor macro that is used extensively throughout. On the opposite end of that spectrum, I had larger problems to solve - the largest (and the center of the game) is the win checking algorithm. Though there might be more elegant ways to do this, I chose to use a 'brute-force' check and iterate over all the possible win conditions, to check for a victory. At first, I was unsure of my ability to complete this, but through much hard work, I was able to develop a solution.

Game Flow

The game flow for this project required that the program be restartable (my own requirement), and that affected the way in which I wrote it. The program starts with an initialization stage - this is where variables that control the game are set, as opposed to being set in the data segment. This program uses many, many (i.e. about 10) macros to operate. These range from smaller things, like printing a pipe character ('|' - mprintcharpipe), to crucial parts of the program that were macro-ized for simplicity, such as the macro mwincheck which calculates if a win has happened. Below is a detailed diagram of how this game moves through its paces, and delivers a game of Connect 4:

Diagram 2.1



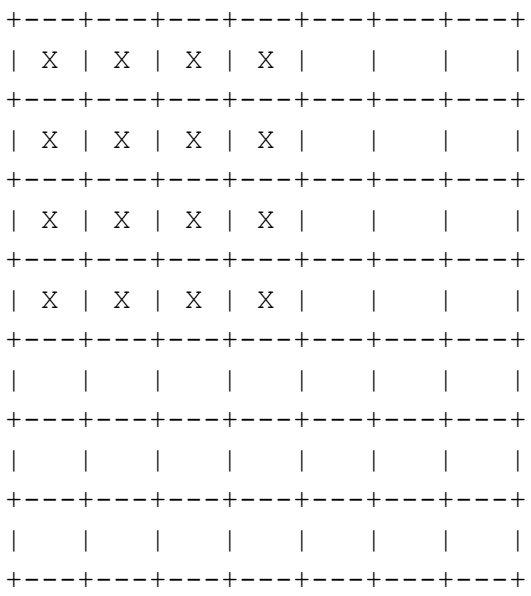
The Important Parts

Win Checking Algorithm

The Win Checking Algorithm is the beating heart of this program - it manages determining wins. It works off of a very simple principle - brute force. I'll examine more in depth how this program determines if a Diagonal Down win has occurred, but the program also does the checks for Diagonal Up, Vertical (Up), Horizontal (Across).

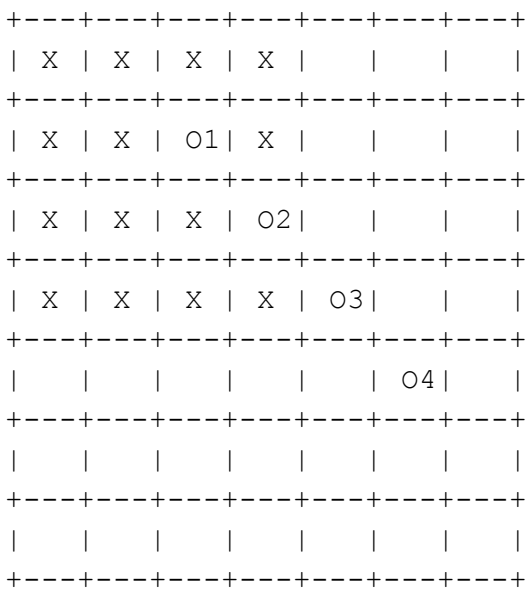
The parent process, in this case, check_diagonal_down is mainly an iterator that generates the pairs: (0..3, 0..3) or a 4 * 4 square in the upper left of the grid. Let's look at that:

Diagram 3.1:



If you identified that these are all the locations where a Diagonal Down win case can start, you are correct. An example one is shown below with the character 'O':

Diagram 3.2



Therefore, this iterator has the job of calling the child function, load_diagonal_down, with each pair with an 'X' in diagram 3.1. load_diagonal_down has a very simple but crucial feature of its own to perform. It has to load the set of values that form a 4- long play from the passed point (x, y) into a 4- WORD array called check_array. These values, for a diagonal down pattern relative to starting point (x, y) would be: (x, y), (x+1, y+1), (x+2, y+2), (x+3, y+3). This is done inside a loop that generates the values 0,1,2,3 (see the values added to x and y) which also correspond

to the indices of where these values 'belong' in check_array. For the above example, our check_array would be some list including the elements {O1, O2, O3, O4}. (note: check_array might be organized by virtue of how it is made, but by all means it does not need to be - {1,1,1,1} == {1,1,1,1} and that is the only concerning case - the win case).

After load_diagonal_down has compiled the set of numbers that correspond with the diagonal down pattern into check_array, that array is checked to see if it is all one number. If it is, a win result is passed back, and caught by the main procedure.

Main Procedure

The main procedure manages the way that the game flows. It controls the player's turns, computer's turn (if enabled). It also manages calling to print, and enacting change if a win or cat game is made. It is messy, and for that I am sorry. It works well, though ;).

Grid Print Out

This grid print out took remarkably quite a bit of time to put together across many iterations, but in its final form I daresay it looks pretty great :). It features colored printout of the blue grid with the red and yellow tiles, which is reminiscent of the game that I played as a kid (european, travel version).

This print_grid procedure leverages one important child procedure - print_value. print_value is called every time that there is a non-frame (i.e. playable space) spot to be printed. It ensures that 1's are colored red, 2's are colored yellow, and empty spaces (0's) are printed black, or 'empty'. This forms the grid that is shown:

Diagram 4.1

A Couple Notes

In-Game this program makes a huge effort to keep interactions with the user to one line, about 2 characters below the grid print out. This line is for everything from displaying win notice, to printing errors, to seeking input. The line is written to by placing the cursor (using Irvine's GotoXY Macro) at the certain offset from the top and side, and then printing text. It is cleared by overwriting it with spaces, because Irvine's ClrScr (Clear Screen) Macro is very slow and noticeably impacts program progression.

The game checks for invalid input (of course), and empty input is treated as a 0. Also (and of course as well), the CPU player (when active) must ensure that it does not play above the top of the table. It does this by continually selecting a random number on 0..6 until it can play in a column.

Debugging Learnings

Through debugging this program (which took many, many hours), I mainly three things. First, nested loops have to be carefully jumped out of, because even though you can see both a 'push' and a 'pop' opcode, jumping skips the pop, messing up the stack frame when it goes to return. Second, I learned that I need to be very careful when reading the textbook

documentation for the Irvine library macros - in one call alone, I misread 'eax' for 'edx' and spent 30 minutes solving the error, obviously kind of frustrated at the end. Lastly, I learned to be very careful about when to use and when not to use the <name> PROC USES <reg's> declaration format. The USES, pushes those registers to the stack before the process starts, and pops them at its termination. This is an awesome feature but with great implications if you intend to return some values in registers from a process. This, of course I learned the hard way.

A Note to Mr. Kalathas

I'll be brief, but I want to thank you for offering such a creative assignment. I had a good time writing and debugging this game, and I value the fact that you are willing to step outside the curriculum in the name of learning hands-on. Thank you!

-- Lyell Read