Project Report

Dots & Boxes Game Coded in Assembly

By:

Cameron Gai - chg200002

Christopher Chaiban - csc210003

Edwin Feng - exf210015

Sayed Rahman - smr210001

Table of Contents

Abstract

This project report is about the “Dots & Boxes” game developed in multiple assembly language modules where the user competes against a computer (AI). Initially, the program displays a blank game board with an 8x6 grid of “+” ASCII characters. The user is then prompted to input the row and column they wish to place their line within the grid. The program prevents the user from inputting values that are not blank spaces (i.e., a dot or already existing line). After the user inputs their values, the board is then displayed twice, the first being the change the user added and the second being the change the computer created. The computer’s move is created randomly and goes through the same error check as the user. When a box is created, a point will be given to the respective player along with a letter declaring whose box it is (‘P’ for user and ‘C’ for computer). Additionally, following pre-established rules, when a player creates a box, they are granted an extra turn. This continues until there are no longer any available spaces and more lines can be placed. The winner is determined by which player created more boxes (or tie if they both have the same amount). The program is split into 4 different files with 3 modules responsible for board and gameplay management and 1 main file. Here is a video demonstration of the project in action.

Challenges

Some challenges that the group faced were a lack of registers due to some poor programming practices, the use of multiple .asm files, creating a sturdy means of displaying the board that translated to other aspects of the program down the line, and box recognition.

- Lack of Registers
    - MIPS provides a limited amount of registers for use, and due to some poor programming practices, many of the temporary registers were used as permanent registers, constantly used by one part of the program. This led to some difficulties in coding as the available registers steadily decreased as the project went on. To combat this problem, we started to shift a lot of the values into .word memory spaces. This can especially be seen in the newer parts of the program where no more registers are being taken up by a function permanently in order to store a value across multiple calls, and everything is stored in appropriate memory spaces.

- Multiple .asm files
    - We had no prior experience creating programs and working with multiple .asm files. We solved this problem by coding in a single large file, which itself proved to be a challenge at times, and splitting the file up into multiple smaller files once the overall structure is formed.

- Board Foundation
    - The game board is probably the most important aspect when creating the “Dots & Boxes” game. It is the only thing the user sees and interacts with, therefore making this the highest priority. The board had to have a strong foundation because most of the game logic had to interact with the board. If the board were sloppy and too complicated to code/interact with, it would make the entire process nigh impossible. The solution was to brainstorm effective ways to create and display the board, which led us to our current product. Line placement, box

scoring, turn count, game looping, and error checking were made much easier due to a strong foundation.

- Box Recognition
    - The most difficult challenge is perhaps detecting when a box is formed and giving the correct player a point and an extra turn. This problem was solved by splitting up the entire game into 6 separate scenarios based on where the input is placed. This is then duplicated for the A.I moves. Meaning that any move made on the board is one of 12 cases and handled separately based on their location. Each case is then given specific offsets they will check for inputs, and if all spots are occupied by an input, a box is formed, and a point is given to the player that made the move.

- User Input
    - We needed a way to stop the user or the A.I. from making illegal moves, as that can give the opposite player an extra turn or go out of the board's memory address and cause an error. We made an algorithm that checks if the 2 positions entered are both odd or even, as all odd and even spaces are either invalid spaces or +. We also check if the entered values go out of bounds of our board size. In order to prevent players from entering repeated values, we created a 221 large .space memory space and stored every user and A.I. moves in it. So each time a move is made, it gets checked against the move history, and if there is a match, the move is invalid. If no move is found then the new move is declared valid and entered in the move history.
- History/Overwriting
    - Throughout the project, aspects of our program would work by itself but in conjunction with another part, it would lead to overwriting of previous moves. Lines already placed by the user could be placed by the computer and vice versa, boxes already created could be overwritten and unintentionally grant another point. The only way to properly prevent this would be having an accurate history that allowed for the error check to recognize that the user/computer is attempting an invalid input. Without this, the program runs without any major bugs, but the

game itself wouldn't be valid. Through quality control and making sure that the user and computer have unique registers so they don't accidentally override scores/moves, this problem is mostly resolved.

Learning Experience

I learned a lot about the importance of good registry and data management. Compared to the smaller solo assignments I've done in mips it was much more important to have good organization because when someone else tries to work on the code they can't get a hang of what is going on.

I also learned the surface of playing sounds on mips, I couldn't really do anything complicated but I did figure out how to play a few sounds. I played a neutral tone on every move and a high pitched good chime on a point score.

Another very important thing I learned is the importance of stacks when calling subroutines from inside subroutines. We learned about it in class but it's much easier to internalize it when you actually have to program it yourself

A crucial lesson was also the importance of a second opinion. There were a couple of parts where we were all stumped on something and had to look over the same screen line by line and put our heads together to come up with a solution. The most notable time was the algorithm to check for complete boxes

My personal favorite part of the code is the input validation. It took me a very long time to figure out how to correctly convert a string to an int, especially when it was a two digit integer. We needed to convert from a string because if we just asked for an int it would crash if you typed "2a" or anything other than strictly an integer value.

Algorithms & Techniques

This program uses loops and branching for a majority of its execution. The initialization and displaying of the board are created through a loop that creates the rows and columns of the board, using branching to end rows/columns. Branching is an important technique that we used for error checking. This prevented the user/computer from entering any invalid moves and prompted them to reenter them.

One of the big techniques that were applied in this project was storing ASCII values in a large freed memory space. This was applied to both displaying and storing the board and user input validation. Since .space frees up bytes of continuous memory space, it was sort of like a 1D array that we had to visualize and convert to a 2D array for displaying the board. By treating the board as a long 1D array also simplified the box detection, instead of a complex algorithm finding all | and - that would form a box, a switch case that utilized the fact that every space on the board relative to another can be reached by using the correct offset. This made the box detection a simple 6-case switch statement that usually only needs to check 1 irrelevant position before correcting to the correct side.

Another technique used is treating 2 spaces of the memory space as a single block. Since all inputs consist of 2 numbers, the user history space will always contain pairs of 2 numbers.

Treating a pair of inputs as a block lets the program to only check every other number for matching ASCII values, meaning if the first user input value does not match with the first ASCII value of the pair, the program skips the next value and checks the one after it, saving time and resources. Another technique used in the process is converting user-inputted integers into ASCII values. By adding 48 to the imputed integer, it is converted to its ASCII equivalent.

An extra feature we decided to implement was sound. A sound byte would be played, signifying the player to input their value. Through some digging, MIPS has a feature to play sounds using Syscall 31. Using $a0-3, you can change different aspects of the sound being played. $a0 controls the pitch, $a1 changes the duration, $a2 chooses the instrument, and $a3 adjusts the volume. After choosing the sound we want, we use Syscall and choose where the sound plays in the game loop.