**Reversi**



Using MIPS Assembly Language with MARS

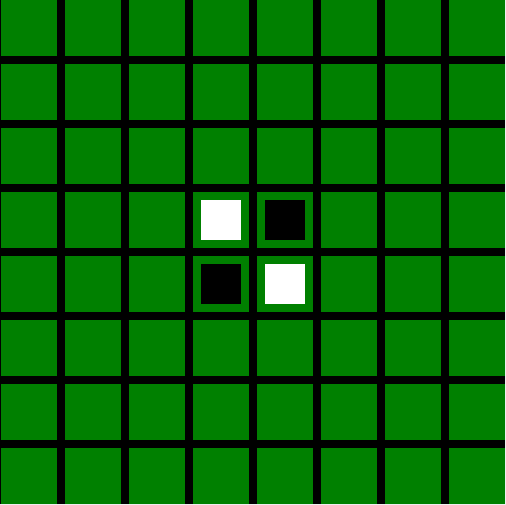
Gary Chen

Project for UTD CS 3340

**Project Description**

The purpose of this project is to create a Reversi (or Orthello) game using the MARS MIPS assembly language. This particular Reversi game utilizes the Bitmap Display tool built in MARS to display the graphical representation of the game while players type in inputs to the console to engage gameplay.

**Challenges**

****The first challenge faced in this project was setting up a graphical display for the game. The team began by researching different games that are also based on MIPS assembly language and had graphical display. I took a looked at a Snake game and implemented its Bitmap Display setup into our Reversi game. I set the unit dimension in pixels to 8x8 and the display to be 512x512 pixels. Colors are filled into the background and spaces using looping function.

With the graphical display resolved, the next challenge was to sync the dynamic display with player inputs. Our solution was to have an 8x8 matrix that represents the spaces of the Reversi game board. At first I wanted to input the RGB hexadecimal values of the corresponding color into the matrix elements. However, rather than inputting the actual RGB codes, I decided to use our own integer codes instead, since integers are much easier to work with. The codes are as follows:

Figure : Reversi Bitmap Display

These numbers will be filled into the matrix as shown in figure 2.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |
| **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |
| **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |
| **0** | **0** | **0** | **2** | **1** | **0** | **0** | **0** |
| **0** | **0** | **0** | **2** | **1** | **0** | **0** | **0** |
| **0** | **0** | **2** | **1** | **1** | **0** | **0** | **0** |
| **0** | **2** | **2** | **2** | **0** | **0** | **0** | **0** |
| **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |

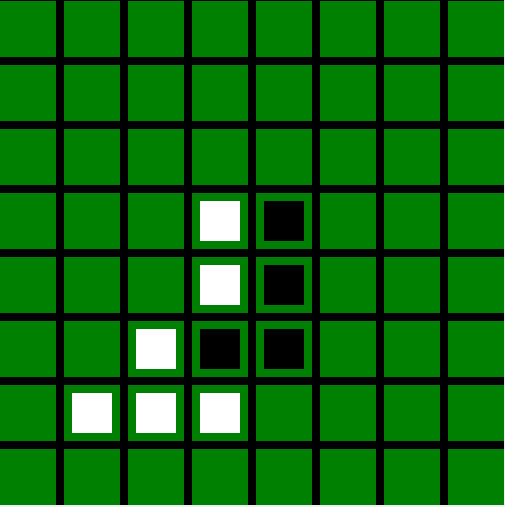


Figure 2: Reversi graphical display with the corresponding matrix color code

Resolving the second challenge greatly helped solved our next challenge which is to decide how the user should enter an input. Given the nature of the matrix element access in most coding languages, the user would simply type in integers in the range of 0 to 7 when prompted for both row and column coordinates, since the matrix, which acts like an array of arrays, start from 0 to 7 for both rows and columns. To access the starting pixel coordinate (PC) to use the coloring loop, I simply apply an algebraic equation.

where

The equation is as such, because P specifies the top left pixel of a game piece, which also is the starting point a nested loop for coloring in the game piece into the display. Utilizing this method and simplified the program by reducing the amount of functions needed to complete a task. The program will check the matrix constantly in order to provide the right color for the graphic display. Specific details will be discussed later.

**Knowledge Gained from Project**

Due to the Challenging nature of the project, I have solidified much of what I learned in class and applied them smoothly into the project. One of the most important aspect of the code our team developed was to call functions efficiently and effectively. I kept our functions to a minimum to avoid coding mess and cluttering.

Aside from applying knowledge gained from class, I have also gained new knowledges in the MARS program. I learned how to use the Bitmap Display built-in tool as well has how to attain address to each pixel to fill it in with color. I have also learned how to setup a matrix in MIPS as well as how to access each element in the matrix. Finally I learned how to call MIPS to show GUI dialog boxes to display messages.

**Algorithms and Techniques**

As explained in Challenges section, having users entering integers to access a matrix element greatly simplifies the program as only very arguments are required when calling a function. This also makes the functions universal, which means the can be used to complete a tasks for both player 1, player 2, or the AI. There are only 4 arguments needed to be passed to other functions throughout the code. The two main arguments are the row and column coordinates specified by the user. The other two are the row and column increments that is to be used to traverse the matrix horizontally, laterally, and diagonally. A pseudocode of how the code works in general is as follows:

Prompt user for row

Prompt user for column

Call ‘Check’ function (arguments: row and column coordinates)

Check if space is empty

Check if piece of opposite color is in any of the adjacent spaces

If at least 1 condition true, continue; else return with error message and reprompt

Check adjacent directions for piece of same color at end (arguments: row and column coordinates, row and column increment)

If a direction is true, flip pieces; else continue to next condition

If at least one of the conditions is true, fill in the player specified coordinate with the player’s piece; else return with error and reprompt

**Important factors**

Algorithm to access the Bitmap Display as well as functions to access matrix elements address and pixel address.

Algorithm for multiplayer gameplay.

Algorithm for the AI for singleplayer gameplay.

Algorithm for conditional checking that adheres to the Reversi rules.

**Suggestions**

Although the code works perfectly, in the end the code is still amateurishly written. I started the project as soon it was announced, so I didn’t have all the knowledge I needed at the time. As a result, the program ended up using and reusing a lot of temporary registers, which made debugging confusing at times. If I ever tried to write this game again, I will be sure to save valuable data into unique memory and reserve space in memory for other functions.