Name: __________________________  Partner’s Name: __________________________

Learning Objectives

1. Practice calling conventions
2. To manage pointers and data structures in ARM assembly
3. To use function calls and recursion in ARM assembly

1 Lab Setup

1. Use illinoisNet_Guest to connect Raspberry Pi to the internet. Make sure you sign in for illinoisNet_Guest by opening the browser and visiting [illinois.edu](https://illinois.edu).
2. Clone your repository CS233 repository.
3. Run (all in one line) to add our repository as a remote

   ```
   git remote add honors_release \
   https://github-dev.cs.illinois.edu/cs233-fa19/_honors_release.git
   ```

   Run the following (separate commands) to get the files for this lab

   ```
   git fetch honors_release
   git merge --allow-unrelated-histories honors_release/RaspberryPiLab2
   ```

   You only need to do this step once. Once you make a commit with merged lab, it will be available in your repository and simply cloning it would be enough.

4. Commit and push your work frequently. If it gives you a missing git configuration error then configure git without the [--global](https://git-scm.com/docs/git-config) option. This is important because the hardware will be shared. We don’t want you to modify any global settings.

5. Once you complete your lab, commit and push your final version. Then delete the files from Raspberry Pi. This is important because the hardware will be shared.

6. You have the option of either working in pairs or individually but we strongly encourage pairs. If you work in pairs, please make sure to fill the partners.txt file.

7. You will have to do this lab during TA Office Hours but please read the full lab manual before coming to lab.
2 Assignments

2.1 Problem 1: DFS

In this problem, your job is to write an assembly program to traverse a complete binary tree using a Depth-first search algorithm. If you do not remember what Depth-first search or a binary tree is, you should refresh yourself before you continue reading this assignment. This task can be accomplished using recursion over an array. If you let element $i$ in an array represent a node on a binary tree, then its two child nodes are elements $2i$ and $2i + 1$ and its parent node is element $i/2$. By recursively checking the child nodes, it is possible to explore every node in the binary tree.

For this lab, your goal will be to return which depth a given input is stored at. Remember that the depth of the root node of a binary tree is 0. If the input does not appear anywhere in the tree, then return $-1$. Please note that with this implementation, we must use an array that begins at index 1. As such, our test cases will all have starting values of $i$ that are 1 or greater. A sample DFS algorithm can be found in `dfs.c`. The assembly code should be written in `dfs.s`. There will be several tree array testcases for you in `dfs_main.s`. We have also provided a Makefile so you can compile `dfs` with the following command:

```
make dfs // compiles without debug option
make dfs-debug // compiles with debug options
```

```c
int dfs(int target, int i, int* tree) {
    if (i > 127) {
        return -1;
    }
    if (target == tree[i]) {
        return 0;
    }
    int ret = dfs(target, 2 * i, tree);
    if (ret >= 0) {
        return ret + 1;
    }
    ret = dfs(target, 2 * i + 1, tree);
    if (ret >= 0) {
        return ret + 1;
    }
    return ret;
}
```
2.2 Problem 2: IslandFill

IslandFill is an algorithm that takes a 2-d array of ‘s and #’s and counts how many "islands" of #’s there are. The ‘.’ characters can be thought of as the ocean, and the ‘#’ characters as land. For our implementation, we will take a 2-d array of chars and replace any ‘#’ characters with different characters to indicate which island it is. The first island will be all A’s, the second island will be B’s, and so on. You label islands starting from the top left and moving across each column, going to a new row when you reach the end of the previous one. For instance, this is an example of what Island will do given the following input:

```
# # _ _ _ _ _ _ _ _ _ _ _ _ 
_ # # _ _ _ _ _ _ _ _ _ # # _
_ _ _ _ _ _ _ _ _ _ _ _ # _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ # _ _ _ _ _ # _ _ _
_ _ _ # # # # _ _ _ _ _ _ _ _
_ _ _ _ # # _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ C _ _ _ _ _ D _ _ _
_ _ _ _ _ _ _ C C C C _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _
```

Land that’s connected diagonally is still part of the same island, and land can’t be connected from one edge of the map to the other to form an island. You can store the ASCII value for a character with an mov r0, #'A' instruction. This number can be incremented like any other to create other characters. For instance, adding 1 to the ASCII value for ‘A’ would give ‘B’. IslandFill uses a function called floodFill for its operation. C version of both islandfill and floodfill are provided to you. Add compilation targets for islandFill in your MakeFile so that it is easy for you to compile it. Use dfs make commands as an example.
struct Puzzle {
    int NUM_ROWS;
    int NUM_COLS;
    char** board;
};

char floodfill (struct Puzzle* puzzle, char marker, int row, int col) {
    if (row < 0 || col < 0) {
        return marker;
    }
    if (row >= puzzle->NUM_ROWS || col >= puzzle->NUM_COLS) {
        return marker;
    }
    char ** board = puzzle->board;
    if (board[row][col] != '#') {
        return marker;
    }
    board[row][col] = marker;
    floodfill(puzzle, marker, row + 1, col + 1);
    floodfill(puzzle, marker, row + 1, col + 0);
    floodfill(puzzle, marker, row + 1, col - 1);
    floodfill(puzzle, marker, row, col + 1);
    floodfill(puzzle, marker, row, col - 1);
    floodfill(puzzle, marker, row - 1, col + 1);
    floodfill(puzzle, marker, row - 1, col + 0);
    floodfill(puzzle, marker, row - 1, col - 1);
    return marker + 1;
}

void islandfill(struct Puzzle* puzzle) {
    char marker = 'A';
    for (int i = 0; i < puzzle->NUM_ROWS; i++) {
        for (int j = 0; j < puzzle->NUM_COLS; j++) {
            marker = floodfill(puzzle, marker, i, j);
        }
    }
}
2.3 Problem 3: Floyd-Warshall Algorithm

The Floyd-Warshall algorithm is an algorithm that, given a directed graph, finds the shortest distance between each pair of vertices. For instance, consider the following graph:

Each vertex has been numbered 0, 1, and 2. The numbers next to each edge are the weight of that edge. Given that graph, the transition matrix, which gives the weights of the edges between each vertex, would look like this:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9999</td>
<td>5</td>
<td>9999</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>9999</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>9999</td>
<td>9999</td>
</tr>
</tbody>
</table>

The row of the transition matrix is the starting vertex, and the column is the end vertex. For instance, there is an edge from vertex 2 to vertex 0 of weight 2. Any edge that is not on the graph is given "infinite" weight, or 9999 in our case. Given these weights, it is possible to calculate the shortest distance from one vertex to another. For the given graph, the shortest distances would look like the following:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>

The distance is given by the sum of weights. For instance, the shortest distance from vertex 1 to vertex 0 is not the 5 weight edge from vertex 1 to vertex 0, but rather the 1 weight edge to vertex 2, and then the 2 weight edge from vertex 2 to vertex 0. This gives a total distance of 3.
The Floyd-Warshall algorithm can obtain these shortest distances. First, it initializes the distance from a vertex to itself as 0. Then, it takes whatever edges exist in the graph and initializes the shortest distances as those distances. For instance, in the example above, the shortest distance between vertices 1 and 0 would be initialized as 5. Then, the algorithm iteratively goes through each vertex and finds intermediate vertices that would result in shorter distances if taken. For instance, during this step, passing through intermediate vertex 2 would be found to result in a shorter distance between vertices 1 and 0. Your task is to convert this code to MIPS. Add compilation targets for floydwarshall in your MakeFile so that it is easy for you to compile it. Use dfs make commands as an example. The C code is given below:

```c
void floydwarshall (int graph[6][6], int shortest_distance[6][6]) {
    for (int i = 0; i < 6; ++i) {
        for (int j = 0; j < 6; ++j) {
            if (i == j) {
                shortest_distance[i][j] = 0;
            } else {
                shortest_distance[i][j] = graph[i][j];
            }
        }
    }
    for (int k = 0; k < 6; k++) {
        for (int i = 0; i < 6; i++) {
            for (int j = 0; j < 6; j++) {
                if (shortest_distance[i][k] + shortest_distance[k][j] < shortest_distance[i][j]) {
                    shortest_distance[i][j] = shortest_distance[i][k] + shortest_distance[k][j];
                }
            }
        }
    }
}
```

Note that negative edge weights can exist. Because the Floyd-Warshall algorithm fails when there are negative cycles, you do not need to worry about those cases.
3 Additional Resources

1. GNU ARM Assembler Quick Reference

2. The gnu Assembler
   (https://web.eecs.umich.edu/prabal/teaching/resources/eecs373/Assembler.pdf)

3. Debugging Assembly Code with gdb
   (http://web.cecs.pdx.edu/apt/cs491/gdb.pdf)

4. Introduction to ARM Assembly Basics
   (https://azeria-labs.com/writing-arm-assembly-part-1/)

5. ARM Quick Reference Sheet
   (http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001l/QRC0001_UAL.pdf)

6. Procedure Call Standard for the ARM® Architecture

   (https://static.docs.arm.com/ddi0403/eb/DDI0403E_B_armv7m_arm.pdf).