EE 352 Lab 2
Spring 2010 Nazarian

Name: __________________________ Score:_______

Assigned: Thursday, February 11

Due: Sunday, February 21 (Digital submission at 11:59pm)

1 Introduction

In this lab you will write a program to find a pathway through a maze using a simple (brute-force) recursive (depth-first) search algorithm.

2 What you will learn

This lab is intended to give you practice at implementing subroutines in assembly and using the stack to save and restore parameters and/or register values.

3 Background Information and Notes

The maze is represented by a 2D array of 1's and 0's. 1's indicate a wall and thus an invalid location from which you need to "backtrack". 0's indicate free space from which you can continue searching. The basic algorithm is as follows: try moving to a neighboring position in one direction and if that location is valid, continue searching from that point (i.e. recursively search again). If the location is invalid, backtrack (exit from the recursive call) to the last valid location and try another direction. If all directions are unsuccessful, then backtrack again. The entire C code is provided below and can be entered, compiled, and run if it helps you understand the algorithm. However, your job is to code the algorithm in assembly, taking care to store necessary values on the stack to ensure each recursive call to "search" remembers its location.

The "maze" and "visit" arrays: The "**maze**" array provides the maze layout. We will assume a 5x5 maze for this lab, though it should be noted that it is very straightforward to make your algorithm parameterized for the size. Each entry in the maze array will be a byte with a value of only '0' or '1'. A '0' indicates a free space while a '1' indicates a wall. **Important: It is implied that walls are around the outer perimeter!** The code skeleton provided gives one example maze that you can use for a first test, but you should try other variants to make sure your code works. When we grade your lab, we will try a few of our own mazes. The "**visit**" array matches the maze in terms of size and keeps a record of where you have visited in the past. A '0' indicates an unvisited location while a '1' indicated you have previously visited that position.

(0,0) **Start**	(0,1)	(0,2)	(0,3)	(0,4)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4) **Exit**

Figure 1 - Layout of a 5x5 Maze showing ordinals (row,col)

The "search" routine: The search routine is a recursive routine with parameters r and c (row and column) that indicate the current location in the maze. The routine makes a few initial checks (whether you are in a valid location, visited that location previously, or are at the exit and thus done) then continues searching first in the east direction, then south, west, and north

Helper routines

a) `bool invalid_location(int r, int c)` - Checks if the current location is out of bounds or a wall, returning a '1' if so.

b) `bool visited(int r, int c)` – Checks if the current location has been visited already

c) `bool atexit(int r, int c)` – Checks if the current location is the exit (goal)

d) `void print_location(int r, int c)` – Prints the current location in the format: (r,c)

4 Procedure and Program Requirements

The requirements that your program must meet are as follows:

a. Your program shall implement the subroutines and recursive search routine exactly as shown in the C language implementation below. You may add other subroutines if you so desire (though it really should not be necessary) but you must implement the routines shown as specified.

b. Your program must use the stack to save and restore r and c as necessary.

c. Every routine should pass arguments using the argument registers starting with $a0 for the first argument, $a1 as the second argument, etc. Also, return values should be placed in $v0 for each routine that returns a value.

d. Your program shall work for all valid 5x5 mazes (i.e. not just the maze shown in the example)

e. Your program shall output a message indicating a path DOES or DOES NOT exist. If a path DOES exist, print out the path from start to finish in **REVERSE** order (i.e. finish back to start). Note: This is already correctly implemented by the C language version, so you really just have to translate that code.

f. Comment your code enough for the TA/Instructor to understand your approach/intent to solving the problem.

g. Try to organize your code in a coherent fashion (no spaghetti code with jumps and branches all over) and make it readable using helpful labels, good formatting/alignment, etc.

5 Programming and Submission Guidelines

1) A program skeleton is provided. Use it as a baseline.

2) You may use any pseudo-instructions you desire. "li – load immediate", "la – load address", and branch pseudo-instructions will be helpful.

3) Each call to search requires its own values of r and c. Thus, you will need to save these values to the stack before the recursive call to search and restore them after search returns. However, when you store values on the stack you may create your own stack frame organization (i.e. you do not have to use the stack frame organization showed in lecture.)

4) Debugging will be the key in this lab. Feel free to add print statements inside your search or other helper routines to help you figure out what is going on. However, please remove them for the final submission. Another helpful technique will be setting breakpoints. It may be helpful to set a breakpoint at the first line of the "search" routine. Each time you hit the run button, the program will run to the beginning of the next call of "search". You should be able to look at $a0 and $a1 as your row/column values to see where search is at.

5) Submit your source file by going to the Lab2 assignment on Blackboard (Assignments..Labs..Lab2) and then attach and submit your ee352_lab2_search.s file. Note: You must "Submit" and not just "Save" when you perform the submission.

6 Review

None

C Implementation and sample program execution.

```
#include <stdio.h>
#define SIZE 5
char maze[SIZE][SIZE] = {   {0,0,0,0,0},
                            {1,0,1,1,1},
                            {1,0,0,0,1},
                            {0,0,1,0,1},
                            {1,0,0,0,0} };

char visit[SIZE][SIZE] = {    {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} };
void main() {
  int r=0, c=0;
  if(search(r,c) == 0)
    printf("Path through maze DOES NOT EXIST!!\n");
  printf("\n");
}
char search(int r, int c){
  if(invalid_location(r,c))   return 0;
  if(visited(r,c))     return 0;
  if(atexit(r,c))
    { print_location(r,c); return 1; }
  if(search(r,c+1))                         // Search east
    { print_location(r,c); return 1; }
  if(search(r+1,c))                         // Search south
    { print_location(r,c); return 1; }
  if(search(r,c-1))                         // Search west
    { print_location(r,c); return 1; }
  if(search(r-1,c))                         // Search north
    { print_location(r,c); return 1; }
  return 0;
}
char invalid_location(int r, int c) {
  if(r < 0 || c < 0 || r > SIZE-1 || c > SIZE-1) return 1;
  return maze[r][c];
}
char visited(int r, int c) {
  char status;
  status = visit[r][c];
  visit[r][c] = 1;
  return status;
}
char atexit(int r, int c){
  if(r != SIZE-1 || c != SIZE-1) return 0;
  printf("Path through maze EXISTS!!\n");
  return 1;
}
void print_location(int r, int c){
  printf("(%d,%d)", r,c);
}
```

Sample Output:

```
Path through maze EXISTS!!
(4,4)(4,3)(3,3)(2,3)(2,2)(2,1)(1,1)(0,1)(0,0)
```

A sample assembly code skeleton is provided on Blackboard. Please use it to begin.

Another Example

Here is the call stack for an example 3x3 maze:

```
0 0 1
1 0 0
1 1 0
```

1. Main will perform initialization and call search(0,0)
2. Search(0,0) should perform the necessary checks, find that everything is valid and call search(0,1) [i.e. to the east]
3. Search(0,1) should perform the necessary checks, find that everything is valid and call search(0,2) [i.e. to the east]
4. Search(0,2) should perform the necessary checks, find that it is a wall (invalid) and return back to search(0,1)
5. Search(0,1) should next search in a new direction by calling search (1,1) [i.e. to the south]
6. Search(1,1) should perform the necessary checks, find that everything is valid and call search(1,2) [i.e. to the east]
7. Search(1,2) should perform the necessary checks, find that everything is valid and call search(1,3) [i.e. to the east]
8. Search(1,3) should perform the necessary checks, find that it is out- of-bounds (invalid) and return back to search(1,2)
9. Search(1,2) should next search in a new direction by calling search (2,2) [i.e. to the south]
10. Search(2,2) should perform the necessary checks, find that it is at the exit, print the message & location and return to search(1,2)
11. Search(1,2) should see that search(2,2) returned true, print its location, and return to search(1,1)
12. Search(1,1) should see that search(1,2) returned true, print its location, and return to search(0,1)
13. Search(0,1) should see that search(1,1) returned true, print its location, and return to search(0,0)
14. Search(0,0) should see that search(0,1) returned true, print its location, and return to main
15. Main will terminate the program

7 Lab Report

Online Submission Only.

8 Grading Rubric

Name: ________________________________ Score: _________

Req. / Guideline	Mult	Score	4 (Excellent)	3 (Good)	2 (Poor)	1 (Deficient)	(0) Failure
			Works correctly all the time	**Works usually but fails in 1-2 cases**	**Fails in several cases but not always**	**Does not work**	**Not implemente d**
Req. a	1						
Req. b	2						
Req. c	1						
Req. d	5						
Req. e	2						
Req. f	2		Well-commented with helpful insight	Some comments with good insight	Some comments but with little insight	Few comments of little insight	No comments
Req. g	2		Well-organized and readable with good labels and formatting	Well-organized with acceptable labels and formatting	Hard to read or poorly organized but formatted well	Hard to read or poorly organized and with poor formatting	Extermely poor organization and formatting

Hard Copy			5 pts.				
Late			-10 per day				
TOTAL							