“Beauty is more important in computing than anywhere else in technology because software is so complicated. Beauty is the ultimate defense against complexity. ... The geniuses of the computer field, on the other hand, are the people with the keenest aesthetic senses, the ones who are capable of creating beauty. Beauty is decisive at every level: the most important interfaces, the most important programming languages, the winning algorithms are the beautiful ones.”


“The road to wisdom? Well, it’s plain and simple to express: Err and err and err again, but less and less and less.”

Piet Hein

This is a solo lab!

Learning Objectives

1. More practice writing MIPS code with conditionals, array refs, and calling conventions

2. To use function calls and recursion in MIPS assembly

Work that needs to be handed in (via github)

- p1.s: implement the dfs function. Run on EWS with:
  QtSpimbot -file p1_main.s p1.s
- p2.s: implement the rule1 function. Run on EWS with:
  QtSpim -file p2_main.s p2_helpers.s p2.s

Additional Information

For Lab 8 we are providing “main” files (e.g., p1_main.s) and files for you to implement your functions (e.g., p1.s), just like in Lab 7. We will need to load both of these files into QtSpim to test your code.

We will only be grading the p1.s and p2.s files, and we will do so with our own copy of p* main.s and p2_helpers.s, so make sure that your code works correctly with an original copy of p* main.s and the solutions provided for lab 7 for p2 helpers.s. We provide you with dfs.c and sudoku.c that contain C implementations of all the functions that you need to implement for this Lab.

Guidelines

- Guidelines are the same as Lab 7.
- You’ll find the assignment so much easier if you try to understand the C code you’re translating before starting.
- Just as in Lab 7, follow all calling and register-saving conventions you’ve learned. It’s even more important in this MP.
- Don’t try to change the algorithms at this point, just write the code in MIPS as closely to the provided C code as possible.
- Remember to test your code using the testcases provided!
Problem 1: DFS [25 points]

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 p1.s. There will be several tree array testcases for you in p1_main.s.

```c
int dfs(int* tree, int i, int input) {
    if (i >= 127) {
        return -1;
    }
    if (input == tree[i]) {
        return 0;
    }

    int ret = dfs(tree, 2 * i, input);
    if (ret >= 0) {
        return ret + 1;
    }
    ret = dfs(tree, 2 * i + 1, input);
    if (ret >= 0) {
        return ret + 1;
    }
    return ret;
}
```
Problem 2: Rule 1 [75 points]

The first rule of CS 233 has nothing to do with Fight Club. Instead this rule goes a significant way toward solving many Sudoku puzzles. We want you to translate it to MIPS assembly. Three things that you need to know:

1. We’ll release solutions to Lab 7 on Wednesday morning after the Lab submission late period ends. Until then, feel free to use your solutions from Lab 7 by copying them into p2.helpers.s (Once the solutions are released, you can copy those over instead.)
2. Be sure your function still calls the versions of has_single_bit_set and get_square_begin that we provide (i.e., don’t implement these inside your function) and observes all calling conventions.
3. Your code will likely be shorter if you use callee saved variables.

Also, we’ve provided a collection of test cases (some of these are commented out in p2.main.s). The first three individually test each of the major inner loops with a board that is missing one row, one column, or a single square in a larger square. We strongly suggest that you implement and test these loops one at a time. We also supplied a board that requires multiple passes of rule1 to solve.

We’ve included the rule1 code on the next page, but it is the same code from Lab 7.
bool rule1(unsigned short board[GRID_SQUARED][GRID_SQUARED]) {
    bool changed = false;
    for (int i = 0; i < GRID_SQUARED; ++i) {
        for (int j = 0; j < GRID_SQUARED; ++j) {
            unsigned value = board[i][j];
            if (has_single_bit_set(value)) {
                for (int k = 0; k < GRID_SQUARED; ++k) {
                    // eliminate from row
                    if (k != j) {
                        if (board[i][k] & value) {
                            board[i][k] &= ~value;
                            changed = true;
                        }
                    }
                    // eliminate from column
                    if (k != i) {
                        if (board[k][j] & value) {
                            board[k][j] &= ~value;
                            changed = true;
                        }
                    }
                }
                // eliminate from square
                int ii = get_square_begin(i);
                int jj = get_square_begin(j);
                for (int k = ii; k < ii + GRIDSIZE; ++k) {
                    for (int l = jj; l < jj + GRIDSIZE; ++l) {
                        if ((k == i) && (l == j)) {
                            continue;
                        }
                        if (board[k][l] & value) {
                            board[k][l] &= ~value;
                            changed = true;
                        }
                    }
                }
            }
        }
    }
    return changed;
}