CS1022 Assignment
Sudoku

19 February 2018

This is an <u>individual</u> assignment and will contribute 20% of your final mark for CS1022.

You must submit your solutions using Blackboard no later than **23:59 on Friday, 9th March, 2018**. Late submissions without a satisfactory explanation will receive zero marks. You will be asked to demonstrate your solution during the scheduled lab on **Friday, 9th March, 2018**. (Your demonstration will be worth up to 5% of your mark for the assignment.) You should also expect someone grading your program to execute it. It will be assumed that programs that do not assemble without errors do not work.

Submit your `.s` assembly language source files and your report **in PDF format** as attachments to the Assignment in Blackboard.

Your `.s` assembly language source files must be suitably commented.

Solutions will be checked for plagiarism.

Introduction

Sudoku is a popular number puzzle game. You begin with a partially completed 9×9 grid of digits in the range $1 \ldots 9$, as illustrated below. You must complete the grid subject to:

(a) each digit appearing exactly once in each row

(b) each digit appearing exactly once in each column

(c) each digit appearing exactly once in each 3×3 sub-grid

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

1 [10%] Getting and Setting Digits

Applying the principles of subroutine interface design presented in lectures, design and implement ARM Assembly Language subroutines to perform the following operations on a Sudoku grid:

(a) Get the value of a digit in the given row and column

```
byte getSquare(word row, word column)
```

(b) Set the value of a digit in the given row and column

```
void setSquare(word row, word column, byte value)
```

Test your subroutines thoroughly using the template project and sample Sudoku grid provided for the assignment. The sample Sudoku grid is represented as a 9×9 grid of byte-size values in memory. The value 0 (zero) represents a blank (unfilled) square in the grid.

2 [25%] Validating Solutions

Design and implement an ARM Assembly Language subroutine (or subroutines) to determine whether a Sudoku grid represents a valid solution (or valid partial solution), based on the rules of Sudoku. A partial solution is valid if it satisfies the rules of Sudoku, ignoring any remaining blank squares in the grid. Your subroutine should return a true/false (one/zero) boolean result.

```
bool isValid(word row, word column)
```

Your subroutine MUST make use of your subroutine from Part 1 to retrieve the digit from a given grid square.

Test your subroutine thoroughly using the template project and sample Sudoku grid provided for the assignment. Modify some of the squares in the sample grid to test your solution for both valid and invalid solutions.

3 [25%] Solving a Sudoku Puzzle

We can adopt a "brute force" approach to finding a solution to a Sudoku puzzle by iterating through the digits 1 ... 9 in the current blank square and, if a digit is valid, moving on to the next square and repeating.

The above approach can be conveniently implemented using recursion, as illustrated in the pseudo-code below. We recursively try to solve a Sudoko grid, beginning with a partially complete, valid grid.

At some point we might realize that none of the digits 1 ... 9 are valid in the current square, because of some incorrect choice we made in an earlier square. This requires us to "backtrack" and try a different value in a previous square. Again, recursion is ideally suited to implementing a "backtracking" algorithm such as this, where each recursive invocation of the subroutine represents the state for a single square (i.e. which of the digits 1 … 9) we are currently trying in that square.

Translate the pseudo-code below into an ARM Assembly Language subroutine. You MUST make use of your subroutines from Parts 1 and 2 above. Test your implementation thoroughly using the provided grid and other grids.

4 [15%] The Extra Mile

Improve or extend the Sudoku solver in a manner of your choosing. You might decide to implement a subroutine to display the Sudoku grid on the console. Alternatively, you might decide to improve the performance of the solver and evaluate the performance experimentally. The choice is yours but you must document what you do in your report.

[20%] Documentation

Document your approach to each part of the assignment. Your documentation should be in the form of a typed document **in PDF format**. This document must be submitted on Blackboard with your three assembly language `.s` source files. Documents in a format other than PDF may receive zero marks.

Your documentation should make use of examples, diagrams and pseudo-code where appropriate. **It must not take the form of a "narrative" for your assembly language program ("I moved the value from R5 into R4. Then I added R4 to R3 and stored the result in R0"). Such narratives will receive very low marks.** Instead, try to describe how your program works at a higher, conceptual level.

Your documentation must also include a detailed description of your methodology for testing each subroutine that you have written. This should include a tabulated list of the inputs used to test your subroutine and the corresponding results, along with an explanation. You should give careful consideration to choosing inputs that test different parts of your subroutines.

Note that marks will be awarded for both the content and presentation of your documentation.

```
bool sudoku(address grid, word row, word col)
{
    bool result = false;
    word nxtcol;
    word nxtrow;

    // Precompute next row and col
    nxtcol = col + 1;
    nxtrow = row;
    if (nxtcol > 8) {
        nxtcol = 0;
        nxtrow++;
    }

    if (getSquare(grid, row, col) != 0) {
        // a pre-filled square
        if (row == 8 && col == 8) {
            // last square -- success!!
            return true;
        }
        else {
            // nothing to do here - just move on to the next square
            result = sudoku(grid, nxtrow, nxtcol);
        }
    }
    else {
        // a blank square - try filling it with 1 ... 9
        for (byte try = 1; try <= 9 && !result; try++) {
            setSquare(grid, row, col, try);
            if (isValid(grid, row, col)) {
                // putting the value here works so far ...
                if (row == 8 && col == 8) {
                    // ... last square -- success!!
                    result = true;
                } else {
                    // ... move on to the next square
                    result = sudoku(grid, nxtrow, nxtcol);
                }
            }
        }

        if (!result) {
            // made an earlier mistake - backtrack by setting
                        // the current square back to zero/blank
            setSquare(grid, row, col, 0);
        }
    }

    return result;
}
```