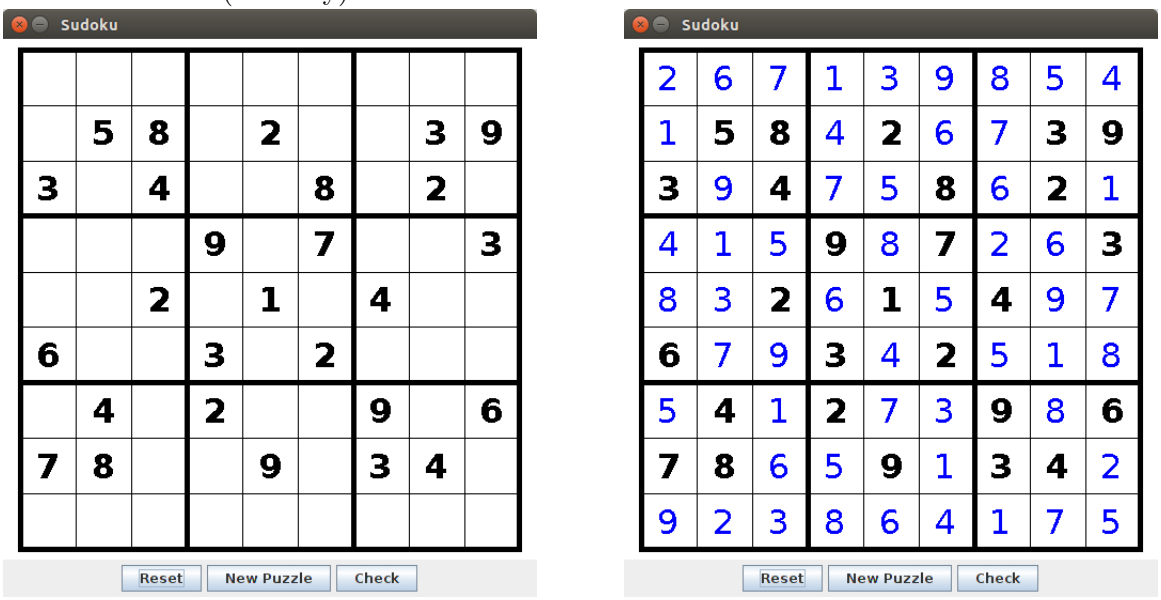
Project - Sudoku

CS 0447 — Computer Organization & Assembly Language

See the Canvas of this course for the due date

The purpose of this project is for you to practice writing backtracking with recursion in assembly language. The main goal of your program is to solve a Sudoku puzzle. The Sudoku (Mars Tool) that we are going to use for this project is shown below. This tool can be found in `sudoku.zip` located in the CourseWeb under this project. Extract all files to your `[..]/mars4_5/mars/tools` directory. If you extract all files to the right directory, when you run the MARS program, you should see "Sudoku (Memory) V0.1" under the "Tools" menu.



Introduction to the Sudoku Puzzle (from Wikipedia)

Sudoku (originally called **Number Place**), is a logic-based, combinatorial number-placement puzzle. The objective is to fill a 9x9 grid with digits so that each column, each row, and each of the nine 3x3 subgrids that compose the grid (also called "boxes", "blocks", "regions", or "subsquares") contains all of the digits from 1 to 9. The puzzle setter provides a partially completed grid, which for a whell-posed puzzle has a unique solution.

Completed games are always a type of Latin square with an additional constraint on the contents of individual regions. For example, the same single integer may not appear twice in the same row, column, or any of the nine 3x3 subregions of the 9x9 playing board.

Visit `https://en.wikipedia.org/wiki/Sudoku` for more information about Sudoku.

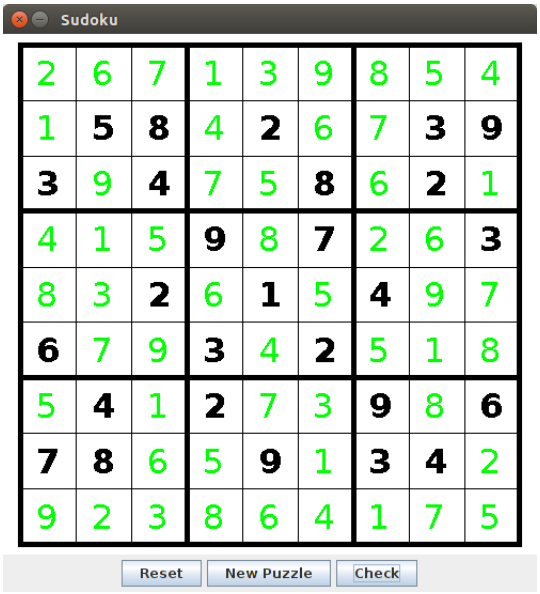
Introduction to the Sudoku (Mars Tool)

Sudoku (Mars Tool) will fill the main memory byte-by-byte starting at the address 0xFFFF8000 with puzzle. If a square is blank, the associated memory location will be filled with 0. For example, the content of the main memory starting at the address 0xFFFF8000 associated with the puzzle shown above is as follows:

Address	Content	row	column
0xFFFF8000	0	0	0
0xFFFF8001	0	0	1
0xFFFF8002	0	0	2
0xFFFF8003	0	0	3
0xFFFF8004	0	0	4
0xFFFF8005	0	0	5
0xFFFF8006	0	0	6
0xFFFF8007	0	0	7
0xFFFF8008	0	0	8
0xFFFF8009	0	1	0
0xFFFF800A	5	1	1
0xFFFF800B	8	1	2
0xFFFF800C	0	1	3
⋮	⋮	⋮	⋮

IMPORTANT: Since the Sudoku (Mars Tool) needs to update memory contents, after you successfully assemble your program, you must press either the "Reset" or "New Puzzle" button before your **run** your program. This will allow the Sudoku (Mars Tool) to write data into your memory. The "Reset" button will clear the puzzle without changing the puzzle and update contents of the main memory. Similar to the "Reset" button, the "New Puzzle" button will create a new puzzle and update contents of the main memory.

Note that each puzzle generated by the Sudoku (Mars Tool) will have exactly one solution. Your program **MAY NOT** modify contents of the memory that contain the original digits. In the above example, you are not allowed to modify contents at memory locations 0xFFFF8009, 0xFFFF800A, 0xFFFF800C, and so on. The "Check" button can be used to verify your solution. If a square contains an invalid digit, the digit will turn red. Otherwise, it will turn green. If a square of the original puzzle that contain a digit has been modified, it will turn red as well. The figure below shows the result of pressing the "Check" button when the puzzle is being solved correctly.



What to Do?

For this project, write a MIPS assembly program named `sudokuSolver.asm` to solve a Sudoku puzzle. Your program must perform the following:

1. Your program must display the puzzle on the console screen of MARS. For example, for the puzzle shown on the first page, your console screen should display the following:

   ```
   000000000
   058020039
   304008020
   000907003
   002010400
   600302000
   040200906
   780090340
   000000000
   ```

   This step simply check that your program can read a given puzzle correctly. This part will be your partial credit in case your program cannot solve a puzzle.

2. Your program must solve the Sudoku puzzle by changing contents of main memory starting at the address 0xFFFF8000. If a data stored in a memory location is not zero, do not modify the content of that memory location since it is a part of the puzzle. If a data stored in a memory location is zero, you must change it to a non-zero value (1 - 9). Again, make sure you follow all rules of Sudoku.

Once the puzzle is solved, your program should simply terminate. After your program is terminated, you should click the "Check" button to verify that the puzzle is solved correctly.

Recursion and Backtracking for Sudoku

An idea of using recursion and backtracking to solve a Sudoku puzzle is to associate a recursive call with a cell in Sudoku. For example, suppose your `_solveSudoku` function looks like the following:

```
boolean _solveSudoku(int row, int column)
{
    :
}
```

When `_solveSudoku(r, c)` is called, this call is responsible for filling the cell at row r column c. Once it finds a number that has no conflict, it should make a recursive call to `_solveSudoku(r, c + 1)` so that the next call will take care of row r, column c + 1.

Now, consider a recursive call `_solveSudoku(r, c)`. At this call, if the cell at row r column c is available, this call has 9 possible numbers (1 - 9) to put into this row r column c. This call will pick the first number that does not have any conflict (row, column, and 3 by 3) and put it into the cell. **Note** that this number (choice) may or may not lead to the correct solution. If the select (no conflict) number does not lead to the solution, this call have to pick the next no conflict number. However, if this call runs out of a choice of numbers. This mean that the choice picked by the **previous** call does not work. So, this call have to send a signal to previous call (a return value) to tell the previous call to pick a new number. But importantly, this call should set the row r column c back to its original value (0) before returning back to the previous call.

Note that some cells may already have numbers. If that is the case, the recursive calls responsible to those cells do not have to do anything. Simply call the next one.

Suppose the first row is the row number 0 and the first column is the column number 0, below shows a pseudo code that solves a Sudoku puzzle using backtracking and recursion.

```
boolean _solveSudoku(r, c)
{
    boolean p;

    if(r == 8 and c == 9)
        return true;

    if(c == 9)
    {
        r = r + 1;
        c = 0;
    }

    if(data at row r column c is not 0)
        return _solveSudoku(r, c + 1);
    else
    {
        for i = 1 to 9
        {
            if(i has no conflict)
```

```
            {
                put i into the cell at row r column c;
                p = _solveSudoku(r, c + 1);
                if(p)
                    return true;
            }
        }

        put 0 back to the cell at row r column c

        return false;
    }
}
```

Requirements

1. Your program **MUST** use backtracking and recursion to solve puzzles.
2. Your program must contain the function named `_solveSodoku`. This will be your recursive function.
3. You can create as many helper function as you wish. For simplicity, at least you should have the following three helper functions:
   - `_checkRow`: This function checks whether a given number is already in a give row.
   - `_checkColumn`: This function checks whether a given number is already in a given column.
   - `_checkSubgrid`: This function checks whether a given number is already in a subgrid where a given row and a given column is located.
4. All functions in your program must follow all calling conventions discussed in class.

Submission

The due date of this project is on the Canvas. Late submissions will not be accepted. You should submit the file `sudokuSolver.asm` to Canvas under this project.