CS 152 Lab 1: RISC-V Assembly

Due: 11:59 PM, Feb. 5

Overview

For this lab you will write a sudoku solver for 4x4 mini-sudokus in RISC-V assembly. This lab will require you to write a recursive procedure, which includes implementing stack management.

Mini-Sudoku

Mini-Sudoku is a simpler form of the sudoku puzzle, where the board consists only of 16 numbers (instead of the original 81). The following figure shows an example mini-sudoku puzzle.

	2		3
1	3	4	
2			4
3		2	

The grid is divided into four sub-grids (according to the darker lines).
The objective is to fill in the blank cells such that the resulting table has the following properties:

1. All cells have either 1, 2, 3, or 4 in them
2. No numbers should be repeated in the same row, column, or sub-grid

The answer for the above example would look like this. A well-formed sudoku puzzle has one, unique answer.

4	2	1	3
1	3	4	2
2	1	3	4
3	4	2	1

Backtracking Algorithm

You are free to choose whatever algorithm you are comfortable with.

The most straightforward, tractable algorithm for solving sudoku is the backtracking algorithm, which is a depth-first search algorithm into all possible solutions.

The backtracking algorithm tries filling in an empty cell with a number, and if the addition of a single number does not cause the grid to be invalid, it moves on to the next empty cell. If the new number causes the grid to be invalid, it tries a different number. If all possible numbers have been tried and nothing works, it “backtracks” to a previous cell, and tries numbers it has not yet tried there. There are many resources on the web which describes the backtracking algorithm, including the wikipedia page for sudoku solving algorithms:
(https://en.wikipedia.org/wiki/Sudoku_solving_algorithms#Backtracking)

In pseudocode, the backtracking algorithm for a mini-sudoku looks like the following. This algorithm assumes all 16 numbers are stored in a 16-element array (“set”), and that empty cells are marked with the value “0”. The “solve” function can be called with “solve(0)” to initiate.

```
boolean solve(int index) {
        if ( index >= 16 ) return true;
        if ( set[index] > 0 ) return solve(index+1);
        else {
                for ( n = 1 to 4 ) {
                        set[index] = n;
                        if ( check(index) and solve(index+1) ) return true;
                }
                set[index] = 0; // returns the value to 0 to mark it as empty
                return false; // no solution
        }
}
```

An important component of this algorithm is the “check” function, which checks if the addition of a number at “index” results in a violation or not. For a 4x4 mini-sudoku, this can be checked using the following pseudocode.

```
Let row = index/4;
Let col = index%4;
Let blk = ((index/8)*8)+(col/2)*2; // index of one of the 4 sub-grids
for ( c = 0 to 3 ) {
        rowval = set[row*4+c];
        colval = set[c*4+col];
        blockval = set[blk+(c/2)*4+(c%2)];
        // check if rowval, colval, blockval are unique
}
```

Hint: A register-efficient way of checking each value is unique, is to keep a bit mask “mask”, which is initialized to zero. For rowval, uniqueness can be checked by checking if (mask&(1<<rowval)) is zero,

for colval, (mask&(1<<(colval+8))), and for blockval, (mask&(1<<(blockval+16))). “mask” has to be updated after each check to set the appropriate location to 1, e.g., mask |= (1<<rowval);
Hint2: While the bit mask approach is register-efficient, you don’t have to build it in a register efficient way. Simply maintaining three, 4-byte arrays somewhere in the stack works perfectly well.

RISC-V Emulator

It is suggested to use the provided lightweight command-line RV32I ISA emulator for this lab. The emulator and its instructions can be found here: https://github.com/sangwoojun/rv32emulator

The use of this emulator is simply for your convenience, and you are welcome to use other, more elaborate emulators such as Spike(https://github.com/riscv/riscv-isa-sim), or Ripes(https://github.com/mortbopet/Ripes).

Skeleton Code

The provided skeleton code, “sudoku.s” consists of a function “solve”, which takes the address of the input data as argument (via a0), and some supporting code around it which repeatedly calls “solve” with different inputs, as well as checks its solution against the provided answers.
Your job is to modify “solve”, as well as add more functions as necessary, to modify the memory space provided via a0 to have the correct answer.

Important: Please only modify the parts denoted as the parts to modify, and do not assume things outside this area to remain the same during evaluation. Things that may change include, but not limited to:

- The values of the inputs and answers
- labels of data such as ‘inputs’, ‘answers’, etc
- Use of registers outside of the “solve” function call, adhering to the RISC-V conventions. For example more s registers may be used, etc.
- MMIO addresses

When executed via the provided emulator, you will see (among others), three lines of output that looks like the following:

[System output]: 0xa
[System output]: 0xa
[System output]: 0xa

This is output from the checker code emitted to the screen via emulated MMIO, and each number is the number of values that are different from the solution. With correct answers, this value should be 0x0 for all three outputs.

Submission and Grading

Please submit one assembly file, “sudoku.s” via canvas.

Your submission will be checked using a set of 25 randomly generated sudoku puzzles with varying difficulty, including the three given in the skeleton code. All questions will have one unique answer. The same set will be used on all submissions. Each correct answer is worth 1 point.

This lab will correspond to 25% of the lab grades.