CMPUT 229 - Computer Organization and Architecture I

Lab 2: Magic Square

Creator: Xander Fair

DISCLAIMER: This page contains math equations and images. If you cannot see either of them on this page, try loading the page on a different web browser.

Introduction

In this lab you will validate a solution to a magic square puzzle. Your solution will require you to write code that handles two-dimensional arrays.

Background

What is a magic square? A magic square is a number puzzle where the goal is to fill in an $n \times n$ grid such that each row, column, and the two main diagonals add up to the same sum. A magic square is *completed* if all cells in the grid have a number. Below is a simple example of a valid magic square where $n = 3$.

6	1	8
7	5	3
2	9	4

$6 + 1 + 8 = 15$ $\quad 6 + 7 + 2 = 15$ $\quad 6 + 5 + 4 = 15$

$7 + 5 + 3 = 15$ $\quad 1 + 5 + 9 = 15$ $\quad 2 + 5 + 8 = 15$

$2 + 9 + 4 = 15$ $\quad 8 + 3 + 4 = 15$

We can see that each row, column, and diagonal sums to 15.

Assignment

Write a RISC-V program that checks **only if the columns** of a completed magic square are valid. Only the columns are checked to avoid writing repetitive code segments — checking rows and diagonals requires similar code to checking columns. You must write the function `magicSquare` with the following specifications:

```
magicSquare:
```

```
This function checks the validity of columns of a magic square.

Arguments:
    a0: pointer to a 2D NxN array of the input magic square where each item is a four-byte uns
    a1: N
    a2: The target sum to check against. That is, what each column would need to sum to for th
        to be considered valid.

Return:
    a0: 1 if all columns are valid. 0 if at least one column is not valid.
    a1: Number of valid columns
```

You can make any other helper functions as long as they do not share a name with any labels already used in the `common.s` file.

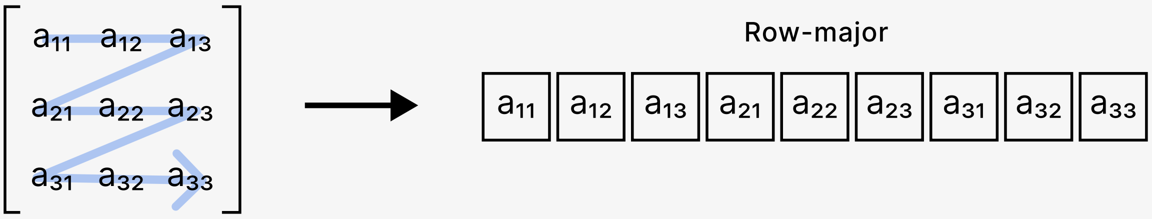
Magic Square Validator in RISC-V

The provided file `magicSquare.s` contains the label for the `magicSquare` function. Your solution **must** be written using this file. Do not modify the `magicSquare` label, because this label is required by the `common.s` file to properly call your `magicSquare` function.

The directive `.include "common.s"` in `magicSquare.s` causes RARS to insert the code from the file `common.s` before the start of any code in the solution. The code in `common.s` allocates space in memory, parses the numbers from the test file into a 2D array of integers, and then calls the `magicSquare` function written as a solution to the lab. You are encouraged to read the code in the `common.s` file to understand how the whole program works.

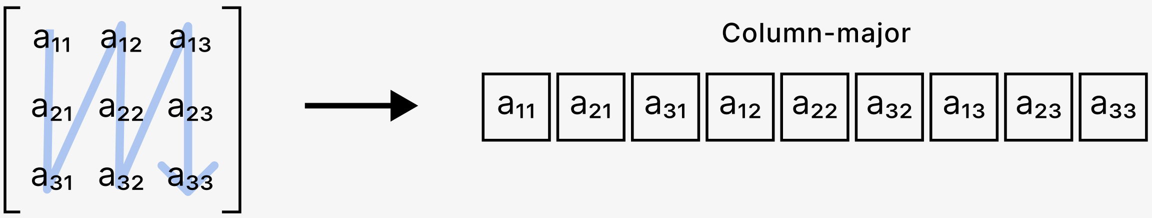
2D Arrays:

In memory, a 2D $n \times n$ array is stored as a linear array of $n \cdot n$ elements. There are two ways to store a 2D array in memory: row-major order or column-major order. For row-major order, each row is contiguous in memory.



a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}
Row-major
a_{11} a_{12} a_{13} a_{21} a_{22} a_{23} a_{31} a_{32} a_{33}

For column-major order, each column is contiguous in memory.



a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}
Column-major
a_{11} a_{21} a_{31} a_{12} a_{22} a_{32} a_{13} a_{23} a_{33}

For this lab, the magic square is stored in **row-major** order, and each element is a single word. For example, in the C language, you might access a row and column with `nums[row][col]`, which is equivalent to `nums[row*N + col]` or `*(nums + row*N + col)`, assuming that there are N elements per row and that each element occupies one byte. In C, this is equivalent to using an array of `char`. In this lab you have to use a modified version of these relations because each entry is a word, not a single byte.

Loops:

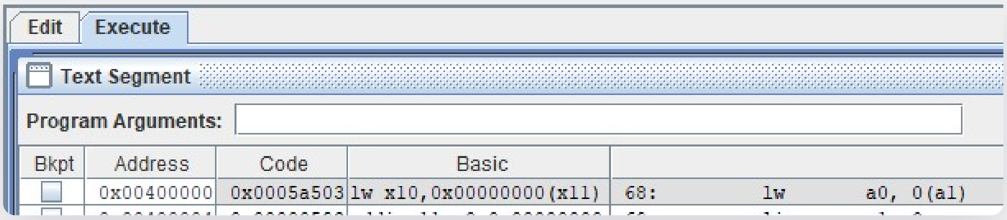
There are many different ways to implement loops in RISC-V. Several examples can be found here: Do-While, While, For.

Testing Your Lab

When debugging, it can be very helpful to step through the assembly code instruction by instruction. You can use `ebreak` statements to pause the execution program execution at a specified instruction and examine the state of the registers and memory. From that point you can then execute instructions one by one.

The folder `Code/Tests/` contains sample test cases. The `*.txt` files contain test inputs. The corresponding `*.out` file contains the correct output. The provided test cases are not extensive. Additional testing is required to ensure that the solution is correct.

To perfom a test, enter the complete file path of the corresponding test input into the *Program Arguments* box in RARS. The file path cannot have any spaces. Do not put quotations areound the filename. Provide a full path to the file. For example `/home/user/cmput229/test.txt` is a valid path, while `/home/user/cmput 229/this is a test.txt` is not.



Program Arguments:

Bkpt	Address	Code	Basic	
□	0x00400000	0x0005a503	lw x10,0x00000000(x11)	68: lw a0, 0(a1)

Once you have entered the path to the test case, you can run the program. If the program arguments field does not appear in your RARS, you need to change your settings. Go to `Settings -> Program Arguments Provided To Program` (second option) and make sure that this opiton is checked.

You can test your lab from a terminal with `rars LABFILE pa TESTFILE` where `LABFILE` is the path to `magicSquare.s` and `TESTFILE` is the path to the file that contains the test case you want to test. Depending on the version of RARS and your operating system RARS may print extra copyright lines. Ignore these lines.

Check My Lab

Link to CheckMyLab

This lab is supported by CheckMyLab. To get started, navigate to the Magic Square Validator lab in CheckMyLab found in the dashboard. From there, you can upload test cases in the *Test cases* table (see below). Your test cases will be shared with the entire class. You can upload your `magicSquare.s` file in the

My solutions table, which will then be tested against all other valid test cases.

Test Case Format

Test cases are plain text files ending with .txt that must be in the following format:

```
S [Where S is the target sum]
[a square grid, where the numbers are separated by spaces]
```

For example:

```
15
6 1 8
7 5 3
2 9 4
```

Assumptions and Notes

- Do not edit any part of the `common.s` file. Only the functions written by you will be graded. A solution that requires any edits to `common.s` to work is likely to be incorrect.
- Do not use any of the labels that are already used in `common.s`
- The size of a magic square is less than or equal to 50×50.
- All entries in the magic square are present.
- Ensure that there is a return instruction at the end of your `magicSquare` function to return execution to `common.s` .

Resources

- Slides used for in-class introduction of the lab (.pptx) (.pdf)
- Marksheet used to mark the lab (.txt)

Marking Guide

- 30% for correctly determining the validity of magic squares (correct `a0` return value for `magicSquare`).
- 50% for correctly determining the number of valid columns (correct `a1` return value for `magicSquare`)
- 20% for code cleanliness, readability, and comments

Submission

There is a single file to be submitted for this lab: `magicSquare.s` which should contain the code for `magicSquare` and all supporting functions.

- **Do not** add a `main` label to this file.

- **Do not** modify the line `.include "common.s"`.
- **Keep** the file `magicSquare.s` in the `Code` folder of the git repository.