CSE 220: Systems Fundamentals I

Homework #4

Fall 2017

Assignment Due: December 1, 2017 by 11:59 pm via Sparky

⚠ **READ THE WHOLE DOCUMENT TWICE BEFORE STARTING!**

⚠ DO **NOT** COPY/SHARE CODE! We will check your assignments against this semester and previous semesters!

ℹ Download the Stony Brook version of MARS posted on Piazza. **DO NOT USE** the MARS available on the official webpage. The Stony Brook version has a reduced instruction set, added tools, and additional system calls you will need to complete the homework assignments.

⚠ You personally must implement the assignment in MIPS Assembly language by yourself. You may not use a code generator or other tools that write any MIPS code for you. You must manually write all MIPS Assembly code you submit as part of the assignment. You may also not write a code generator in MIPS Assembly that generates MIPS Assembly.

⚠ All test cases MUST execute in 100,000 instructions or less. Efficiency is an important aspect of programming.

⚠ Any excess output from your program (debugging notes, etc) may impact your grading. Do not leave erroneous printouts in your code!

⚠ Do not submit a file with the functions/labels `main` or `_start` defined. You are also not permitted to start your label names with two underscores (`__`). You will obtain a ZERO for the assignment if you do this.

Assignment Overview

The focus of this homework assignment is on memory organization and working with multidimensional arrays in memory. This assignment also reinforces MIPS function calling and register conventions.

In this homework you will be implementing the functions to play the game, 2048. If you have not played the game, give it a try.

To implement the game, we will be using a 2D array to track the state of the game. The original 2048 game is played in a 4x4 grid. However, we will generalize the functions to support a board of any dimension. To succeed in this assignment, you should become familiar with loading values from and storing values to 2D arrays of arbitrary dimensions, as well as performing address arithmetic on 2D arrays containing objects of arbitrary size.

You **MUST** implement all of the functions in the assignment as defined. It is OK to implement additional helper functions of your own in `hw4.asm`.

⚠ **You MUST follow the MIPS calling and register conventions. If you do not, you WILL lose points.**

⚠ Do not submit a file with the functions/labels `main` or `_start` defined. You will obtain a ZERO for the assignment if you do this.

🛈 If you are having difficulties implementing these functions, write out the pseudocode or implement the functions in a higher-level language first. Once you understand the algorithm and what steps to perform, then translate the logic to MIPS.

🛈 When writing your program, try to comment as much as possible. Try to stay consistent with your formatting. It is much easier for your TA and the professor to help you if we can figure out what your code does quickly.

Understanding the Game Board

Each cell on the 2048 game board holds power-of-2 integers greater than 2^1. A cell with value X can be merged with another (adjacent) cell with same value X (X is a power of 2) to form a single cell with value 2X. The board in memory will be a 2D array of 16-bit (2-bytes, or half-word) two's complement integers. Recall that 2D arrays can be stored in two ways: row-major order or column-major order. Our game board will be stored in **row-major order**. 2D arrays are implemented as an array of 1D arrays in memory. Therefore, the rightmost cell of a row is followed by the left-most cell of the next row. The 2D array will also order the rows such that the top-left cell of the board is the first element. Thus, memory addresses will increase as we progress from the leftmost cell to the rightmost cell and as we progress from the top row to the bottom row of the game board.

Notation and Terminology

The board will have `n` rows and `m` columns with `n` and `m` ≥ 2.

Each cell of the board is 2 bytes, or one half-word, in memory. The cell holds a two's complement integer. If the cell value is -1, the cell is empty. Any other value in the cell which is not a power of

2 greater than 2^1 is invalid.

Throughout this document we will refer to cells of the board in (row, col) format. Position `(0,0)` of the board is the top-left corner. Position `(n-1, m-1)` is the bottom-right corner of a board.

For example, the 48 cells in a board with 6 rows and 8 columns would be indexed as in the figure below.

Top of Board

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)

Bottom of Board

Getting Started

From the Resources section of Piazza download the files `hw4.asm` which you need for the assignment. At the top of your `hw4.asm` program in comments put your name and SBU ID number.

```
# Homework #4
# name: MY_NAME
# sbuid: MY_SBU_ID
```

How to Test Your Functions

To perform **basic** tests on your functions, we suggest you write your own main functions **in separate files** which call each function independently. Remember to include your `hw4.asm` file in each of your main files using the line:

```
.include "hw4.asm"
```

⚠ Your assignment will not be graded using any of your mains!

Your main files will not be graded. No code that you include inside of the main files should be submitted. You will only submit your `hw4.asm` file via Sparky. Make sure that all code required for implementing your functions (`.text`) are included in the `hw4.asm` file!

To make sure that your code is self-contained, try assembling your `hw4.asm` file by itself in MARS. If you get any errors (such as a missing label), this means that you need to refactor (reorganize) your code, possibly by moving labels you inadvertently defined in your mains to `hw4.asm`.

All functions implemented in the assignment must be placed in `hw4.asm` and follow the standard MIPS register conventions for functions taught in lecture. If you write your own helper functions, these also must be included in `hw4.asm`.

⚠ There is no need to create any variables or memory storage for this homework in your `hw4.asm` file. Therefore you **MUST NOT** declare a `.data` section in your `hw4.asm` file. Your functions should be implemented using ONLY of the information provided in the arguments. (Your main files will need to declare .data sections in order to call your functions with inputs.)

⚠ Make sure to initialize your registers within your own functions! Never assume registers or memory will hold any particular values!

⚠ Each function will be tested independently and therefore should not rely on any information other than the state of the arguments provided to your function. Do not write any functions that expect certain labels to be available in the main file.

Part I: Cell and Board

This version of the 2048 game will conclude when a cell has a value greater than or equal to 2048. Since each cell is stored as a half-word in memory, there is a limit on the largest possible value that can be represented, namely $2^{15} - 1$. We will be using the sign bit (msb) to represent an empty cell, i.e. an empty cell is -1.

Board with 3 rows and 6 columns

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)

Board with 4 rows and 4 columns

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

2048 Visualizer in Mars

There should be an option located in the Tools menu called `2048 Display Board`. This tool simulates the concept of video memory where data stored directly to an address in memory is reflected in a visual manner. The simulator will treat the bytes starting at `0xffff0000` to a maximum of `0xffff007f` as the values of a rectangular text console. The row and column sizes for which the board is reading/displaying is denoted by the drop-down options at the top of the tool. Each half-word of the memory is mapped to a cell of the board based on the rows and columns specified.

What this means is that the simulator will attempt to translate the values stored at these memory addresses as values to display. This technique is a form of Memory Mapped I/O (MMIO). You can read more about this here: http://en.wikipedia.org/wiki/Memory-mapped_I/O.

⚠ The maximum address that the board will try to translate and display is defined by `0xffff0000 + (row*column*2) - 2`. Since the board can only display at most a 8x8 board, the starting address for the last cell is `0xffff007e`.

⚠ DO NOT hardcode the address of the visualizer in your functions. All functions should assume the address of the board is provided as the argument to the function. To test with the 2048 Visualizer, pass the address `0xffff0000` as the board address to your functions.

⚠ NOTE: There is **NO** maximum board size for this assignment. Your code should be capable of handling any row and column ≥ 2. The visual tool can only display a maximum of 8x8.

How to use the 2048 Visualizer

To begin using the `2048 Display Board`, launch the tool from the Tools Menu. Once the `2048 Display Board` Tool appears, assemble your testing program. Select the size of the board you wish to test from the drop down menus of the tool. Before pressing the run button, you must connect the visualizer tool to MARS by pressing the `Connect to MIPS` button on the tool. When your program stores any data to the addresses in the board range (address `0xffff0000` to `0xffff007e`) the stored value will display in the cell of the board.

Note, the tool can be used to test boards with any dimension from 2-8, in row-major order. Upon selecting a new size, the board will be reset visually, but the values stored in memory remain unchanged. To explicitly zero the memory press the `reset` button at the bottom of the tool.

⚠ Changing the row and columns will cause the board to resize. If the board is cut off, simply pick a corner of the tool and drag the window to make it larger or smaller.

Working with the game board in memory

We will need to read and write the values of an individual cell of the 2D array that represents the board. In order to access a cell on the board, we must use the general formula for working with 2D arrays in memory. Generalizing is not only good practice, but necessary in a situation like this assignment where we will be working with boards of arbitrary dimensions.

Suppose you would like to access `obj` in the 2D array `obj_arr`. The dimensions of `obj_arr` is `n` rows by `m` cols, and the location of `obj` is at `(i, j)` in the 2D array, where `i` is in the range [0, `n`-1], and `j` is in the range [0, `m`-1]. The address of `obj` is then given by:

```
obj_arr[i][j] = base_address + (row_size * i) + (size_of(obj) * j)
```

where

```
row_size = n_cols * size_of(obj)
```

Note that `obj_arr[i][j]` will be the computed address of where the desired object starts in memory, and not the object itself. Also, consider why the size of a row is `n_cols * size_of(obj)`.

ℹ Why does this make sense? How would this calculation vary if it were column-major order instead? What is the size of an `obj` for us?

⚠ For all functions, you may assume the `board` references the correct space in memory when `num_rows` and `num_cols` are valid.

a. `int clear_board(cell[][] board, int num_rows, int num_cols)`

   Never assume that memory is in the state you want it to be in! The part of memory our board occupies may have previously been used by another program and thus have garbage data in it. To ensure a clean, appropriate state, we will have to explicitly set it.

   This function will clear the board, setting all the cells of the board to empty cells (-1).

   Arguments:

   - `board`: The starting address of a 2D array holding the state of the game board.
   - `num_rows`: The number of rows in the board.
   - `num_cols`: The number of columns in the board.
   - *returns*: 0 for success, -1 for error.

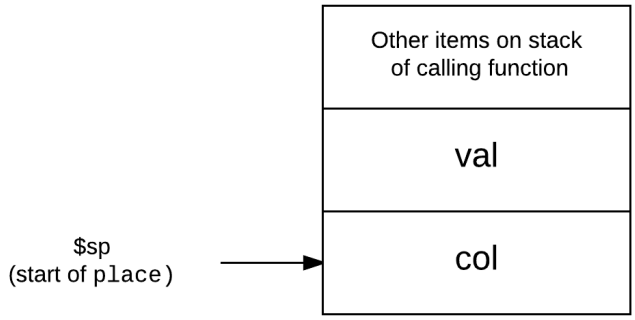
   Return -1 for error:

- `num_rows` or `num_cols` is less than 2.

To place a new value into the board, implement the following function:

b. `int place(int[][] board, int n_rows, int n_cols, int row, int col, int val)`

Since the function has more than 4 arguments, the remaining arguments must be placed on the stack. The caller function will place the arguments `val` and `col` on the stack prior to calling `place`. At the start of the `place` function, the stack pointer and arguments will be set as follows:



This function takes in a 2D array `n_rows` by `n_cols` in size, calculates the address of a particular cell given by (`row`, `col`), then stores the given `value` into the appropriate fields of the two-byte cell object in memory.

Arguments:

- `board`: The starting address of a 2D array holding the state of the game board.
- `n_rows`: The number of rows in the board.
- `n_cols`: The number of columns in the board.
- `row`: The row number of the cell being set.
- `col`: The column number of the cell being set.
- `value`: The number being placed into the cell.
- *returns*: 0 for success, -1 for error

Return -1 for error in any of the following cases:

- `n_rows` or `n_cols` is less than 2

- `row` is outside the range of [0, `n_rows` -1]
- `col` is outside the range [0, `n_cols` -1]
- `value` is **not** -1 **AND not** a power of 2 ≥ 2.

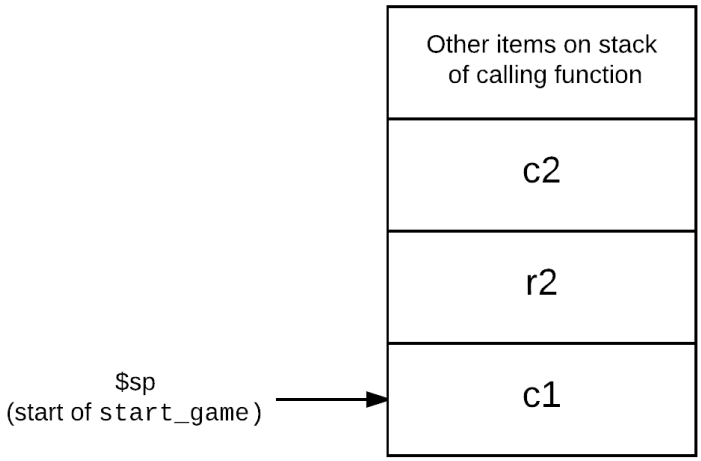
⚠ All functions **MUST** validate `num_rows` and `num_cols` to be ≥ 2.

🛈 Consider writing a get_cell function as a helper function, similar to place. It is not a reuqirement of the assignment, however it may be useful in your implementation.

c. `int start_game(cell[][] board, int num_rows, int num_cols, int r1, int c1, int r2, int c2)`

This function will start a game by calling `clear_board` first and then placing two starting values (the starting value is 2), one at `(r1,c1)` and another at `(r2,c2)`.

Since the function has more than 4 arguments, the remaining arguments must be placed on the stack. The caller function will place the arguments `c1`, `r2` and `c2` on the stack prior to calling `start_game`. At the start of the `start_game` function, the stack pointer and arguments will be set as follows:



Remember, it is the responsibility of the function that PLACES arguments on the stack before calling an inner function to also REMOVE those arguments when that inner function returns. The inner function, `start_game` will read the arguments, but it MUST NOT shrink the stack to remove them.

Arguments:

- `board`: The starting address of a 2D array holding the state of the game board.
- `num_rows`: The number of rows in the board.

- `num_cols`: The number of columns in the board.
- `r1`: Row number for first starting value.
- `c1`: Column number for first starting value.
- `r2`: Row number for second starting value.
- `c2`: Column number for second starting value.
- *returns*: 0 for success, -1 for error.

Return -1 for error if:

- `num_rows` or `num_cols` is less than 2.
- `r1` or `r2` is outside the range [0, `num_rows` -1].
- `c1` or `c2` is outside the range [0, `num_cols` -1].

⚠ `start_game` **MUST** call `clear_board` and `place`.

Part II: Gameplay

d. `int merge_row(cell[][] board, int num_rows, int num_cols, int row, int direction)`

e. `int merge_col(cell[][] board, int num_rows, int num_cols, int col, int direction)`

Each function will iterate over pairs of cells in the specified `row` or `col`, attempting to merge the cells together. In order for a cell to merge with another, both must have the same (non-empty) value and they must also be adjacent. `direction` specifies in which direction to merge as shown in the table below.

direction	merge_row	merge_col
0	left-to-right	bottom-to-top
1	right-to-left	top-to-bottom

If a successful merge has occurred, the cell which holds the merged value contains twice the original. The other cell becomes an empty cell (-1).

Arguments:

- `board`: The starting address of a 2D array holding the state of the game board.
- `num_rows`: The number of rows in the board.

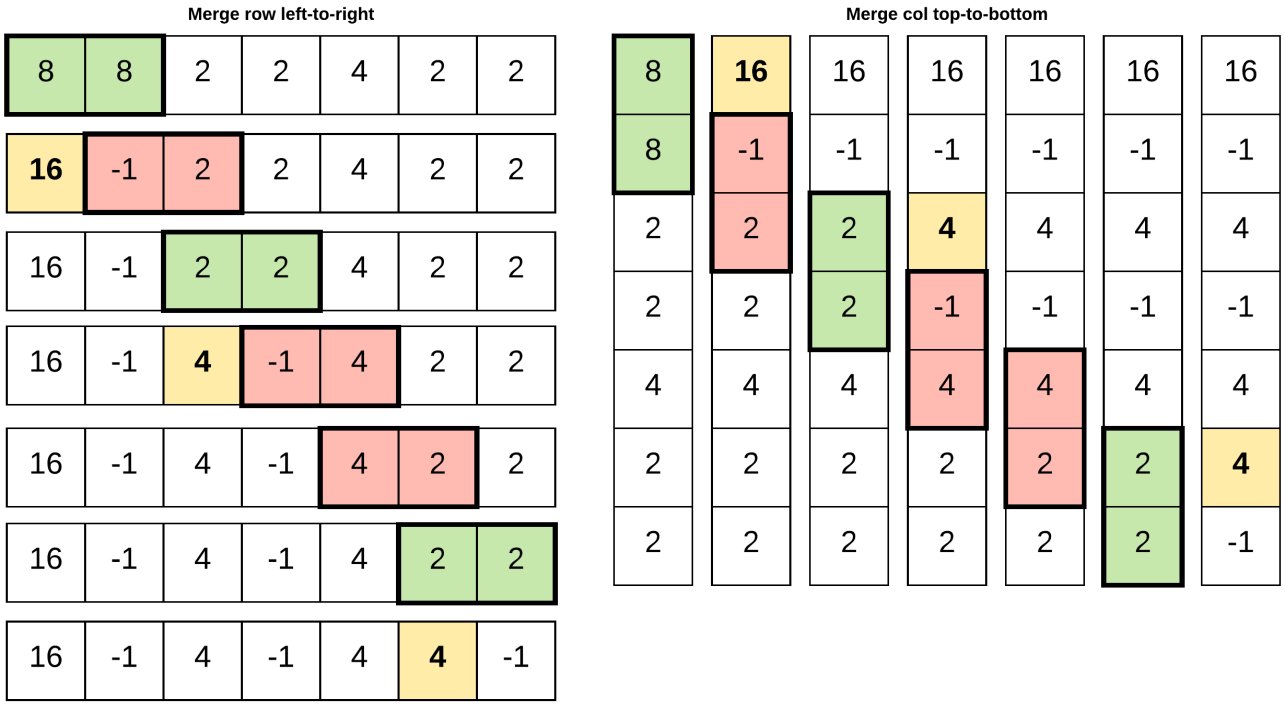
- `num_cols`: The number of columns in the board.
- `row` or `col`: The row or column whose cells should be merged.
- *returns*: Number of cells with non-empty values in the `row` or `col` after merge, -1 on error.

Return -1 for error in any of the following cases:

- `row` or `col` is invalid (i.e., outside board range or negative).
- `num_rows` or `num_cols` is less than 2.
- `direction` is invalid, eg. not 0 or 1.

The figure below illustrates two examples of merging. The bold box shows the pair of cells being compared. Green denotes values to merge. Red denotes values which cannot be merged. Yellow denotes the modified value.

For the left example shown, the row is merged left-to-right. We begin at the left-most cell and compare it with its neighbor. If they have the same value, they are merged and the sum is placed in the left-hand cell of the pair. The right-hand cell of the pair is replaced with -1. We repeat the process starting with each cell in the row (except the last cell).



f. `int shift_row(cell[][] board, int num_rows, int num_cols, int row int direction)`

g. `int shift_col(cell[][] board, int num_rows, int num_cols, int col, int direction)`

These functions shift the cells of a specified `row` or `col` as far as possible in the appropriate direction. The shifting will start and end as specified in the table below.

Function	**`direction`**	Start	End
`shift_row`	0 (left)	Col #`1`	Col #`num_cols-1`
`shift_row`	1 (right)	Col #`num_cols-2`	Col #`0`
`shift_col`	0 (up)	Row #`1`	Row #`num_rows-1`
`shift_col`	1 (down)	Row #`num_rows-2`	Row #`0`

A cell is shifted only if it is not empty. If a cell cannot be shifted any further, the function attempts to shift the next (non-empty) cell in the row/col. This is repeated until all the cells in the row/col have been attempted to be shifted.

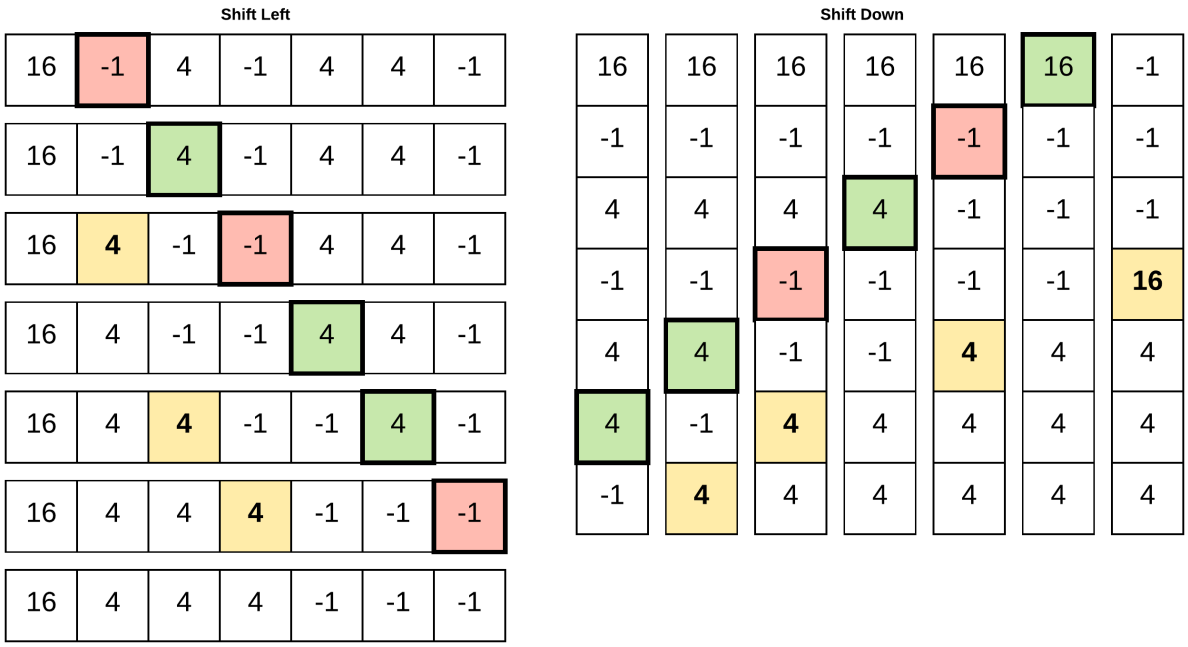
Arguments:

- `board`: The starting address of a 2D array holding the state of the game board.
- `num_rows`: The number of rows in the board.
- `num_cols`: The number of columns in the board.
- `row/col`: The row/col that should be shifted.
- *returns*: Number of cells shifted, -1 on error.

Return -1 for error in any of the following cases:

- `row` or `col` is invalid (i.e. outside board range or negative).
- `num_rows` or `num_cols` is less than 2.
- `direction` is invalid, eg. not 0 or 1.

The figure illustrates two different shifts. The left-hand figure shows a Left Shift, whereas the right-hand figure shows a Down Shift.



h. `int check_state(cell[][] board, int num_rows, int num_cols)`

This function checks the current state of the game. If any cell on the board has a value $\geq$ 2048, the game has been won. If all cells are full and no adjacent cells can be merged (no diagonals), the game has been lost.

Arguments:

- `board`: The starting address of a 2D array holding the state of the game board.
- `num_rows`: The number of rows in the board.
- `num_cols`: The number of columns in the board.
- *returns*: 1 if the game has been won, -1 if the game has been lost, 0 otherwise.

⚠ You may assume `num_rows` and `num_cols` are valid values passed to `check_state`.

i. `(int, int) user_move(cell[][] board, int num_rows, int num_cols, char dir)`

This function simulates a user's move. `dir` can be one of the following values: `'L'`, `'R'`, `'U'` or `'D'`. This function will call the previous game-play functions in the following order:

```
for each row or col:
    call shift_row or shift_col
    call merge_row or merge_col
    call shift_row or shift_col

call check_state after loop
```

The loop will iterate over all the rows or columns based on the direction:

Direction	`dir`	Iterate Over
Left	L	Rows
Right	R	Rows
Up	U	Columns
Down	D	Columns

This algorithm will first merge all the adjacent cells together and then shift them as close together as possible in the specified direction. This is considered to be one move. At the end of the move, the state of the board is checked to see if the user won the game.

Arguments:

- `board`: The starting address of a 2D array holding the state of the game board.
- `num_rows`: The number of rows in the board.
- `num_cols`: The number of columns in the board.
- *returns*: `(0,x)`, where `x` is the return value of `check_state` or `(-1,-1)` if any of the functions had an error or `dir` is an invalid character.

You can test your functions by writing a main program that runs the game as it should be played by calling the functions you implemented above. This is additional work that is **NOT** required and you won't necessarily be guided through this. Get creative!

⚠ Remember: **DO NOT** submit a file with the functions/labels `main` or `_start` defined. You are also not permitted to start your label names with two underscores (`__`). You will obtain a ZERO for the assignment if you do this. Additionally, **DO NOT** submit any main file that plays the game using the functions you implemented. Only submit your `hw4.asm` file which contains the ***implementation*** of the functions defined above.

Hand-in instructions

Do not add any miscellaneous printouts, as this will probably make the grading script give you a zero. Please print out the text **EXACTLY** as it is displayed in the provided examples.

When writing your program try to comment as much as possible. Try to stay consistent with your formatting. It is much easier for your TA and the professor to help you if we can figure out what your code does quickly.

See Sparky Submission Instructions on Piazza for hand-in instructions. There is no tolerance for homework submission via email. They must be submitted through Sparky. Please do not wait until the last minute to submit your homework. If you are having trouble submitting, stop by office hours prior to the deadline for assistance.