CSE 220: Systems Fundamentals I

Homework #3

Fall 2016

Assignment Due: November 4, 2016 by 11:59 pm

Assignment Overview

The focus of this homework assignment is on reading data from files, memory organization, and multi-dimensional arrays in memory. This assignment also reinforces MIPS function calling and register conventions.

In this homework you will be implementing Bombsweeper, a version of every computer scientist's favorite game - Minesweeper. If you are unfamiliar with the game, give it a try here. The Help menu at that website will give you the full rules of the game and how to play it.

To implement the game we will use a basic display screen which operates similarly to VT100. The standard VT100 terminal uses ANSI escape codes to specify the foreground and background colors of each position of the terminal. In this assignment, you will use the ANSI characters with and a special designed font to display the Bombsweeper game in a 10 x10 display. To display the game, the tool uses the idea of Memory Mapped I/O (MMIO), which is described a little later.

In order to complete this assignment you will have to get familiar with displaying information in the MARS MMIO Minesweeper display, which is a 2D array data structure.

Please read the assignment completely before implementing all the functions.

You **MUST** implement all the functions in the assignment as defined. It is OK to implement additional helper functions of your own in the `hw3.asm` file.

⚠ **You MUST follow the efficient MIPS calling and register conventions. Do not brute-force save registers! You WILL lose points.**

⚠ Do **NOT** rely on changes you make in your main files! We will test your functions with our own testing mains. Functions will not be called in the same order and will be called independent of each other.

⚠ 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.

ℹ 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.

ℹ 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.

Getting started

Download `hw3.zip` from Piazza in the homework section of Resources. This file contains `hw3.asm` and `main.asm`, which you need for the assignment. At the top of your `hw3.asm` program in comments put your name and SBU ID number.

```
# Homework #3
# name: MY_NAME
# sbuid: MY_SBU_ID
```

How to test your functions

To test your functions, simply open the provided `hw3.asm` and `main.asm` in MARS. Next, just assemble `main.asm` and run the file. MARS will take the contents of the file referenced with the `.include` at the end of the file and add the contents of your file to the main file before assembling it. Once the contents have been substituted into the file, MARS will then assemble it as normal.

`main.asm` provides a wrapper for your functions to enable actual game play. You may modify this file, or create your own files to test your functions independently.

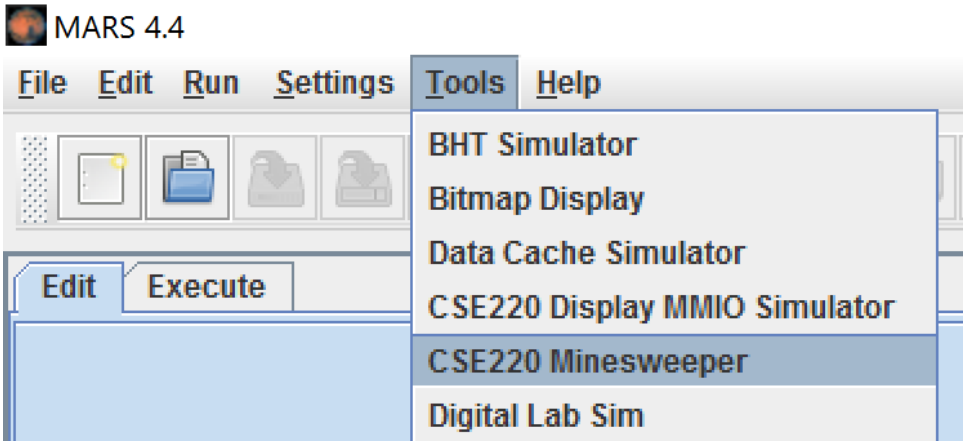
⚠ Your assignment will not be graded using the provided main.

Any modifications to `main.asm` will not be graded. You will only submit your `hw3.asm` file via Sparky. Make sure that all code require for implementing your functions (`.text` and `.data`) are included in the `hw3.asm` file!

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

The Bombsweeper Display in MARS

In the Tools menu, there is an option called **CSE220 Minesweeper**. If your version of MARS does not have this tool, you need to download the version of MARS for this semester from Piazza.



To use the tool, you must connect it to your assembled program. When you are ready to test your code, assemble the main.asm, open the Minesweeper tool and then press the button “Connect to MIPS". When you run your code you will see real-time changes in the display when you change the MMIO values.

This tool simulates displaying data on the screen similar to a terminal windows (like sparky). A section of main memory is mapped directly to each cell in the screen. This technique is a form of Memory Mapped I/O (MMIO). Each cell of the display is specified by a half-word (2 bytes) of information. The lower byte contains the ASCII character to be displayed at the cell position. The upper byte contains the background and foreground color information for the cell. The MMIO region in MARS begins at address `0xffff0000`. You can select this region in the MARS simulator from the drop down in the data segment sub-window. The simulator will treat the bytes starting at `0xffff0000` to `0xffff00c7` as the values of a 10-column-by-10-row window. What this means is that the simulator will attempt to interpret the values stored at these memory addresses as ASCII characters and colors to print to the display.

Display colors

FOREGROUND/BACKGROUND VALUE	ATTRIBUTE_VALUE = 0	ATTRIBUTE_VALUE = 1
0	Black	Dark Gray
1	Red	Bright Red
2	Green	Bright Green
3	Brown	Yellow
4	Blue	Bright Blue
5	Magenta	Bright Magenta
6	Cyan	Bright Cyan
7	Gray	White

Colors on the display are controlled by specific values stored in the upper byte. The Foreground color is

the color of the character which is displayed. The Background color is the background color of the entire cell. Below is a chart of the colors available and their corresponding values. The **ATTRIBUTE_VALUE**, also known as the **BOLD BIT**, will set the depth of the color.

To store all of the color information into a single byte, the following format is used. The example shown in the table sets the Background color to Green and the Foreground color to Bright Magenta. The hexadecimal value for the byte is also shown.

	Bit position and meaning							
Field Description	Background Bold Bit	Background Color			Foreground Bold Bit	Foreground Color		
Bit Position	7	6	5	4	3	2	1	0
Bit Value	0	0	1	0	1	1	0	1
Hexadecimal Value	2				D			

The default cell state will be black background, white foreground, and ASCII char NULL (‘\0‘).

Display font

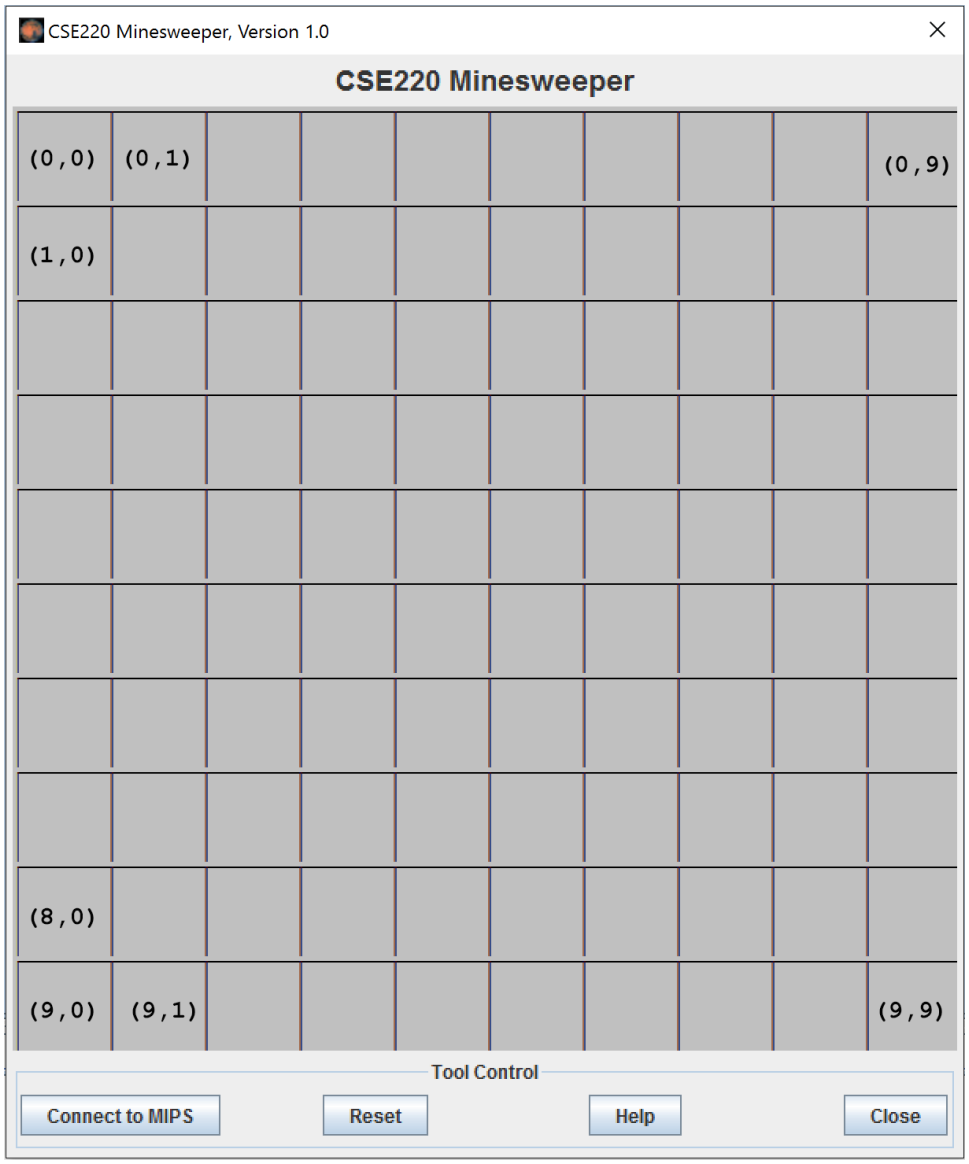
The font in the Bombsweeper display is a custom font. The table illustrates the ASCII character mappings to symbols used for the game. All printable characters not listed in the table map to an ‘X’.

ASCII Char	Symbol
‘0’	
‘1’	
‘2’	
‘3’	
‘4’	
‘5’	
‘6’	
‘7’	
‘8	
‘b’ or ‘B’	
‘e’ or ‘E’	
‘f’ or ‘F’	
All others	
‘\0’	

⚠ Note: The ASCII character ’0’ and its symbol should never be displayed during game operation. However, this character/symbol was included in the display font to assist you in debugging your logic.

Display Layout

As mentioned above, the region of video memory which is mapped between the addresses `0xffff0000` to `0xffff00c7` describes a 10-column-by-10-row console. Since each location on this display takes up two bytes (one for the colors and one for an ASCII character), this means that each row actually takes up 10 columns $\times \frac{2\,\text{bytes}}{\text{column}}$ which leaves us with $\frac{20\,\text{bytes}}{\text{row}}$. So the total continuous region of memory consists of 200 bytes.



In attempt to visualize what is actually happening, lets look at some address values and some translations of that address to (row, col) coordinates. The base case of bytes [0:1] (`0xffff0000` & `0xffff0001`).

Byte 0 is the 8-bit value of the ASCII character to display on the screen. Byte 1 is the 8-bit format for the coloring information for the location at which the character is being displayed. Bytes [0:1] make up the value to display at location $(0, 0)$ on the console.

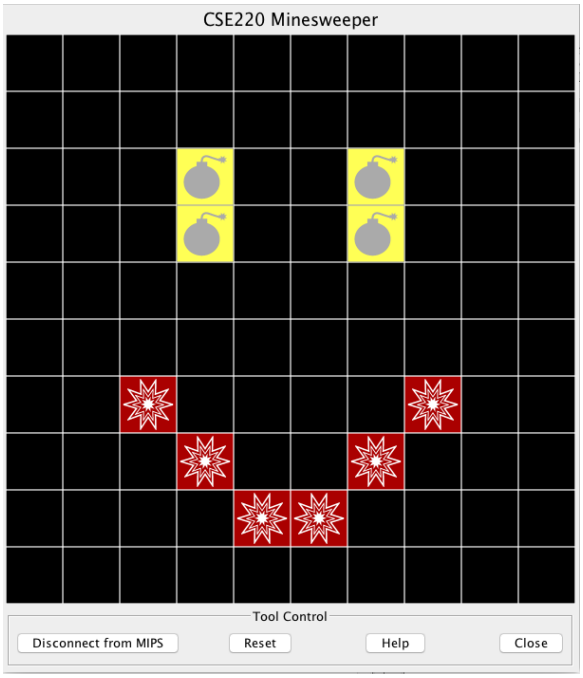
The next two bytes [2:3] (`0xffff0002` & `0xffff0003`) make up the value to display at location $(0, 1)$.

Bytes [4:5] (`0xffff0004` & `0xffff0005`) make up the value to display at $(0, 2)$. Etc.

The memory mapped I/O region of memory is stored in Row-Major Order. "Row-major order" means that each entry per row is placed into memory consecutively one after another, followed by the next row, and so on.

Part I: Basic Display Functionality

Before starting to implement the functions for Bombsweeper you will implement a function that will get you familiar with the MMIO display and 2D arrays. Begin by creating a function that will display a smiley face (because doing CSE 220 homework is fun, just like winning minesweeper!)




```
/*
This function fills the MMIO region to draw a smiley face.
*/
void smiley(void)
```

This function will reset each cell in the display to the default colors (background: Black, foreground: White) and NULL ASCII character. Then it will set particular cells of the display to show a smiley face.

The cells of the smiley face eyes will have a background color of Yellow and foreground color of Grey. The ASCII character of the eyes will be set to 'b' for a bomb. The background color for the mouth will be Red (not Bright Red), the foreground will be white, and ASCII character 'e' for an exploded bomb.

The coordinates of the eyes are: (2,3), (3,3), (2,6), and (3,6). The coordinates for the smile are (6,2), (7,3), (8,4), (8,5), (7,6), and (6,7).

Make sure you clear everything in the MMIO before you draw your smiley face otherwise you can end up with some “greens” in your smile (garbage in memory).

Part II: Reading from Files

In this version of minesweeper you will be reading the content of the game from a file instead of randomly generating a map based on the first click of the mouse.

ℹ `main.asm` takes the filename as a argument to the program. The filename must be in the directory relative to the MARS jar file, not where your main.asm and hw2.asm files are stored.

In this part you will create functions to open files, close files, and read data from the input files into an array in memory. To assist with reading and writing files, MARS has defined additional system calls.

Service	Code in $v0	Arguments	Results
open file	13	$a0 = address of null-terminated filename string $a1 = flags $a2 = mode	$v0 contains file descriptor (negative if error). See note below table
read from file	14	$a0 = file descriptor $a1 = address of input buffer $a2 = maximum num of characters to read	$v0 contains num of characters read (0 if end-of-file, negative if error). See note below table
close file	16	$a0 = file descriptor	

ℹ Service 13: MARS implements three flag values: 0 for read-only, 1 for write-only with create, and 9 for write-only with create and append. It ignores mode. The returned file descriptor will be negative if the operation failed.

The underlying file I/O implementation uses java.io.FileInputStream.read() to read and java.io.FileOutputStream.write() to write.

MARS maintains file descriptors internally and allocates them starting with 3. File descriptors 0, 1 and 2 are always open for: reading from standard input, writing to standard output, and writing to standard error, respectively (new in release 4.3).

There is an example of using these syscalls here.

Create the following function, `open_file`. The function is provided the name of the file to open as an argument. The function must perform the system call to open the file for reading only and return the file descriptor value returned by the system call.

```
/*
This function performs the system call to open a file and return the file
descriptor of the file.
```

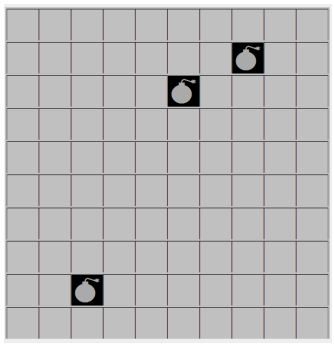
```
@param filename: Starting address of the string containing the filename.
@return The file descriptor (integer) of the open file.
*/
public static int open_file(char[] filename)
```

Next create the function, `close_file`. The function is provided the file descriptor to a file as an argument. The function should perform the system call to close the file.

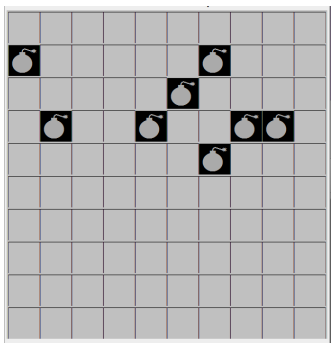
```
/*
This function goes and closes the file specified by the file descriptor.

@param fd: File descriptor of the file that needs to be closed.
*/
public static void close_file(int fd)
```

We will use these two functions in the main program to open and close the map file for the game.

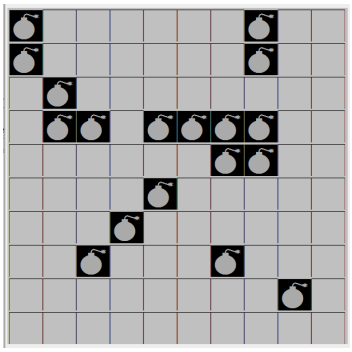
Each map file will load a single game with the coordinates of each bomb in the 10 x 10 board. Each line of the input file will specify the row and col coordinates of a bomb. The coordinates will be separated by a space. Refer to the sample map files included in the assignment zipfile, which are depicted in the below screenshots.



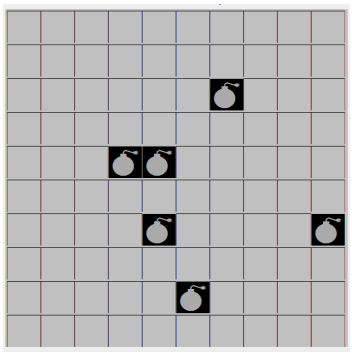
`map1.txt`



`map2.txt`



`map3.txt`



`map4.txt`

A valid map file must meet the following conditions:

- There must be at least one bomb position specified.
- There can be at most 99 bombs specified.
- The position of each bomb must be valid: (0,0) through (9,9).
- Repeated coordinate pairs in the file are ignored (e.g., if (2,7) appears twice, the second instance is ignored).

Since each cell in the MMIO is specified by two bytes – one for the colors and one for an ASCII character – there is no room in the MMIO to store the game information. A array of 100 bytes is needed to store the game information. We will refer to this array as `cells array`. This array will be passed to your functions. For each cell of the board, the following information is stored in a single byte of the `cells array`:

bit 7: Always set to 0

bit 6: Has the cell has been revealed by player? (0: no, 1:yes)

bit 5: Does the cell contain a bomb? (0: no, 1:yes)

bit 4: Has a flag been set on the cell? (0: no, 1:yes)

bit 0-3: Number of bombs that are adjacent to the cell (0-8)

Create the function `load_map` to read the bomb positions from the map file and initialize the `cells array` to the start state of a game. If any error is detected in the map file, the function will return -1.

An error is any invalid character, or invalid value. VALID characters are the numerical characters 0-9, and the whitespace characters: space, tab (\t), carriage return (\r) or newline (\n). The function should be able to handle any amount of extra whitespace characters throughout the file. Coordinates come in pairs (row, col), and appear in the range (0,0) through (9,9). Therefore if the file contains an odd number of integer coordinate values, or a coordinate value less than 0 or greater than 9, the input file is invalid.

The coordinates in each map file are specified as (row, col). For example, `map1.txt` contains the following lines:

```
1 7
8 2
2 5
```

This file contains the positions of 3 bombs at (1,7), (8,2), and (2,5).

```
/*
This function loads the minesweeper game from the specified file into the array.

@param fd: File descriptor
@param array: The "cells array", which stores the state of the game.
@return 0 if the end of file (EOF) was reached, -1 if the file contains
invalid data. An invalid file format means that it contains invalid
input (invalid character or cell position) or an invalid number of bombs.
*/
public static int load_map(int fd, byte[] array)
```

When loading the map, each cell should start off as hidden (bit 6 set to 0) and no flag set (bit 4 set to 0). Given the position of the bombs in the map file, set the bomb bit (set bit 5 to 1) for the corresponding `cells array` position. Once all bombs are set, calculate and set the number of bombs positioned adjacent to each cell (bits 0-3) in each entry of the `cells array`.

⚠ DO NOT assume that the memory allocated for the game is empty! You must clear all bytes before storing any information.

Also, to be able to play the game, you must keep track of the cell that the player is operating on. Therefore, you need to initialize the two global variables `cursor_row` and `cursor_col` in your `.data section` to store the row and col position of the player's cursor. Initialize row and col to position $(0, 0)$, the top-left corner, in `load_map`.

ℹ You may write helper functions to assist you with any of the above functions. For example, it may be useful to write a function to check an adjacent cell for a bomb.

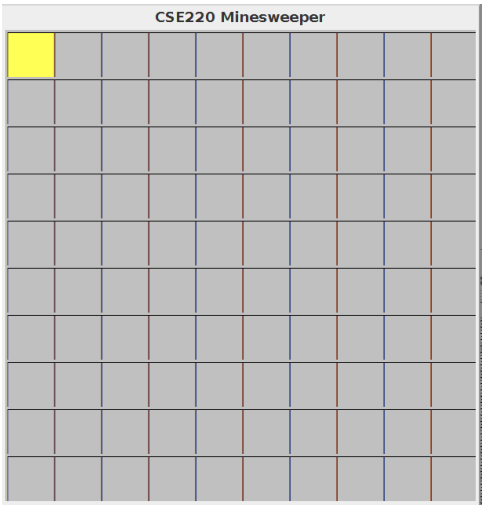
Part III: Displaying Game Information

The `load_map` function loaded the initial state of a game into the `cells array`. However, none of this information is displayed to the player in the window. To display the information to the player, the cells of the MMIO display must be set.

Create the function `init_display`, which will initialize the game display to the starting state.

```
/*
This function initalizes the ASCII char and foreground and
background colors of the MMIO to the the starting game state.

Hidden cells will have a grey foreground and background and an
ASCII character of '\0'.

Set and display the cursor on the screen. The cell at which the cursor is
positioned has a background color of yellow. The foreground and ASCII
character of the cell are unmodified.
*/
public static void init_display();
```

Once the initial state of the game is loaded, the display board will display hidden grey squares and the position of the cursor. Recall that the (row, col) position of the cursor is managed internally by your code, not the main file.



When playing the game, we need to be able to change the state of a particular cell of the board based on actions, such as moving the cursor around or setting the flag on a cell. To assist with this task, create the function `set_cell`. This function will set a specified MMIO cell (row, col) to have the ASCII character, foreground color and background color specified by the arguments. The function returns -1 if the row, col, FG color, or BG color are invalid values. Return 0 if the arguments are valid. Your code does not need to check if the ASCII character is invalid.

```
/*
This function changes the content and colors of the cell (row,col) in the
MMIO to the specified contents.

@param row: The row index
@param col: The column index
@param ch: The character to be displayed
@param FG: The new foreground color
@param BG: The new background color

@return -1 if any argument (except ch) is invalid. Valid values for the
parameters are 0<=FG<=15, 0<=BG<=15, 0<=row<10, and 0<=col<10. Return 0
if all parameters are valid.

Invalid input DOES NOT make any changes to the MMIO display.
*/
public static int set_cell(int row, int col, char ch, byte FG, byte BG);
```

ℹ Note that this functions takes 5 arguments. As there are only 4 argument registers, you will need to push the fifth argument onto the stack prior to calling the function. `set_cell` will read the argument from the stack. It is the the responsibility of the calling function to remove the argument from the stack once `set_cell` returns. It is CONVENTION that the function which places items on the stack is responsible to remove said items.

This function will be used to visually modify the MMIO to display the following cell states:

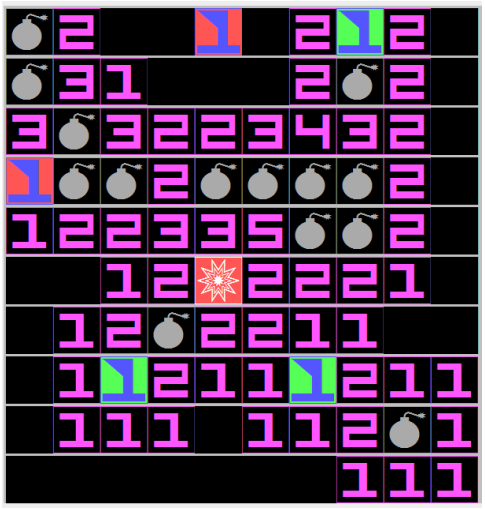
State	ASCII	BG	FG
Bomb	'b' or 'B'	Black	Grey
Exploded Bomb	'e' or 'E'	Bright Red	White
Numbers	ASCII digit	Black	Bright Magenta
Flag	'f' or 'F'	Grey	Bright Blue
Hidden, Empty Cell	'\0'	Black	White

Refer to the Display Colors chart for the ASCII values of each color.

ℹ Note that the `set_cell` function can be used to modify the MMIO cell to contain any ASCII character and color combination. The above combinations are for usage in the game. DO NOT hard-code them into the `set_cell` function. The functions `perform_action` and `reveal_map` defined later will pass the above specified colors to `set_cell`.

In real Minesweeper when you win the game, a winning screen is displayed. In our Bombsweeper game we will display a smiley face by calling the `smiley` function you wrote earlier.

When you lose the game (reveal a cell with a bomb), the full map will be revealed to show the bomb positions. In the cell which ended the game, display the Exploded Bomb (at the position of the cursor). All other cells are revealed, thereby revealing all digits and bomb positions, except cells which were flagged. If a cell is flagged and the cell had a bomb (correctly identified bomb position), change the BG color to Bright Green. If the cell is flagged but the cell did not have a bomb (incorrectly identified bomb position), change the BG to Bright Red. Shown is an example using `map3.txt`.



Create a single function which will reveal the game, win or lose, or do nothing if the game is still on-going.

```
/*
Modifies the MMIO region to display the exploding bomb and map if
game is lost, the smiley face if game is won, or does nothing if the
game is still on going.

@param game_status (1: won, 0: on-going, -1: lost)
@param array: The "cells array", which stores the state of the game.
*/
public static void reveal_map(int game_status, byte[] array)
```

ℹ `reveal_map` MUST call functions `smiley` and `set_cell`.

ℹ Note: The cursor will disappear when you win or lose the game. Upon a win, the smiley face will overwrite the cursor. When you lose, the exploded bomb will be at the position of the cursor. The cursor's current position will become irrelevant, as upon the next load of the game the cursor will be positioned back at $(0, 0)$.

Part IV: Gameplay

Now that the game state can be properly displayed in the MMIO display, we need to enable the player to interact with it. For this you will need to use the row and column position of the cursor you initialized earlier in your `.data` section. The cursor will be visible on the display with a yellow background.

The player can perform any of the following actions to play the game:

- **'f' / 'F':** the flag is TOGGLED on the cell
- **'r' / 'R':** the cell is revealed
- **'w' / 'W':** the cursor is moved up 1 row
- **'a' / 'A':** the cursor is moved 1 column left
- **'s' / 'S':** the cursor is moved down 1 row
- **'d' / 'D':** the cursor is moved 1 column right

Create the function `perform_action`, which will move the cursor or alter the cell at the position of the cursor both in the `cell array` and in the MMIO display.

All the actions indicated by the player are be performed based on the cursor's **current** cell.

Note, the cursor may not move beyond the edges of the board. This means there is no wrapping around

from left to right or top to bottom. Attempting to move outside the boundaries of the board is an error. Do not move the cursor and return -1

Other error-handling is given in the function comments below.

This function will call `search_cells` which is defined in Part V. For now, create an empty function for `search_cells` and call the empty function in `perform_action`.

```
/*
This function checks the cell at the cursor position and applies the game's
logic. If the action tries to move the cursor to an invalid position or
tries to place a flag on a revealed cell, do no action and return -1.

If a player tries to flag a cell which is already flagged, the flag is removed
(toggled).

If a player tries to reveal a cell with a flag, the flag is removed and the
cell is revealed.

If a player tries to flag a revealed cell, do nothing and return -1.

If a player tries to reveal a cell which is already revealed, do nothing and
return -1.

When the desired action is r/R to reveal a cell, this function must call
the search_cells function on the cursor's location. The search_cells
function is described in Part V.

@param array: The "cells array", which stores the state of the game.
@param action: The ASCII code of the action to be performed by the player:
f/F/r/R/w/W/a/A/s/S/d/D as described a few paragraphs back.
@return 0 for a valid move and -1 for an invalid move
*/
public static int perform_action(byte[] array, char action);
```

`perform_action` MUST call function `set_cell` and `search_cells`.

After the player performs an action, the state of the game must be checked.

There are three states the game can be in:

- Win: To win the game, the player must have correctly flagged all the bombs and no additional cells.

- Lose: To lose the game, the player must have revealed a cell with a bomb.
- On-going: All other situations.

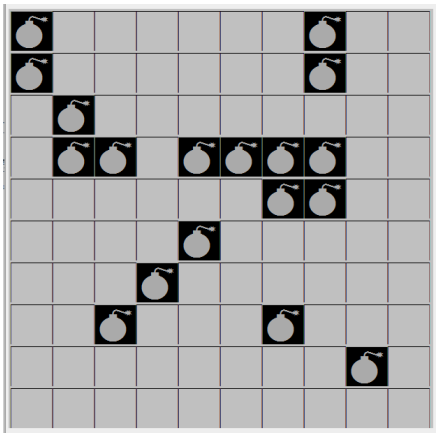
Create the function `game_status` which will check the current state of the game.

```
/*
Checks the status of the array argument to return the state of the game.

@param array: the "cells array" which stores the state of the game.
@return 0 if game is ongoing, 1 if game won and -1 if game is lost.
*/
public static int game_status(byte[] array);
```

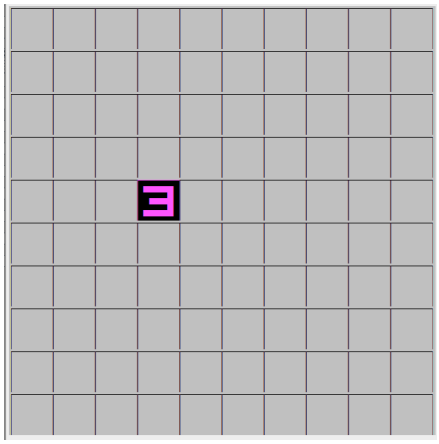
Part V: Advanced Gameplay

In real Minesweeper, when you reveal a cell that has no adjacent bombs, the game performs a search for nearby cells that are either empty or that contain a number indicating the how many mines are nearby. It continues searching outwards, revealing cells as it goes, but does not reveal cells containing mines. The search also will not attempt to go beyond the boundary of the board.

Consider `map3.txt`. Once the game is loaded, if the player reveals cell (4,3), only the single cell is uncovered. Since the cell has adjacent bombs, only this cell is revealed.

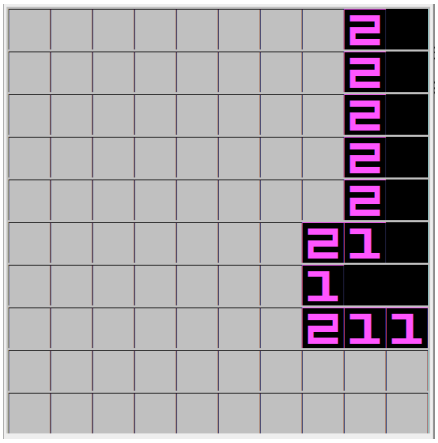


bomb positions in map3.txt



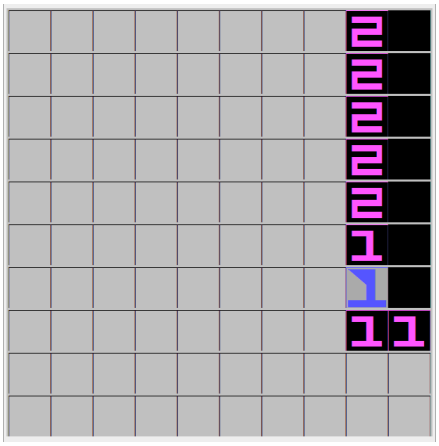
reveal (4,3)

If the player instead reveals cell (3,9), a set of cells are revealed. All cells without any adjacent bombs are revealed. In addition to these cells, any cells adjacent to them with an adjacent bomb is revealed.



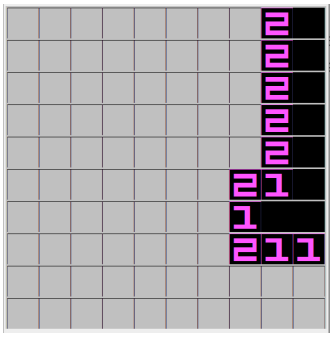
reveal (3,9)

If a flag was previously placed on a cell (whether a bomb exists in the cell or not), it is not revealed and any neighboring cells are not revealed either.

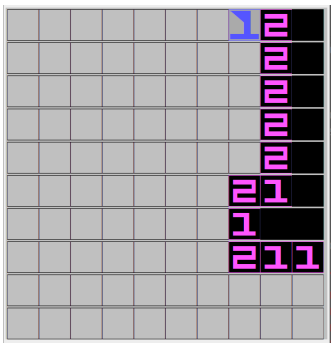


flag @ (6,8), reveal (3,9)

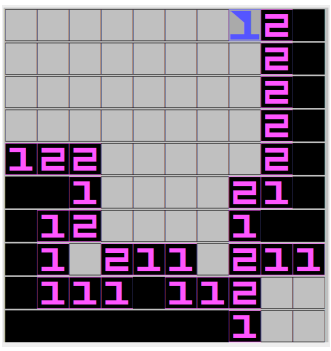
The player reveals (3,9) and the places a flag on (0,7). Then the player reveals (9,0), `search_cells` will uncover the additional cells. Flags can then be placed on (7,2) and (7,6).



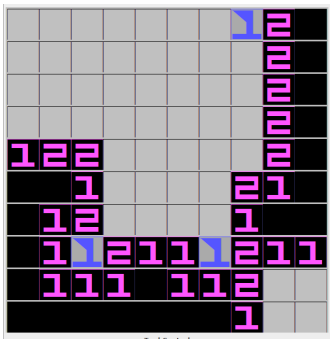
reveal (3,9)



flag (0,7)



reveal (9,0)



flag (7,2) & (7,6)

To implement this functionality, create the function `search_cells`. It is called by `perform_action` when the player reveals a hidden cell that has a bomb count of 0.

```
/*
This function reveals the cells adjacent to the given cell index and the
neighbors of those, etc., until it reaches a cell adjacent to a bomb or
edge of the board.
@param array: The "cells array", which stores the state of the game.
@param row: Row index where the search for empty cells begins.
@param col: Column index where the search for empty cells begins.
*/
public static void search_cells(byte[] array, int row, int col)
```

This function will use the stack to track all adjacent cells which could be revealed. See the pseudocode below for the algorithm you should implement. `$fp` is the **frame pointer**, a register we can use in concert with `$sp` to manage the stack. In this algorithm it is used to help us keep track of what cells of the board we still need to process and possibly reveal during the search. In the pseudocode, `isFlag` returns `true` if a cell has been flagged by the player, `isHidden` returns `true` if a cell is hidden, `reveal` reveals a cell, and `getNumber` returns the count of bombs (0 through 8) near a cell.

```
fp = sp;
sp.push(row);
sp.push(col);
while (sp != fp) {
    int row = s.pop();
    int col = s.pop();
    if (!cell[row][col].isFlag())
        cell[row][col].reveal()
    if (cell[row][col].getNumber() == 0){
        if (row + 1 < 10 && cell[row + 1][col].isHidden() && !cell[row + 1][col].isFlag()){
            sp.push(row + 1);
            sp.push(col);
        }
        if (row + 1 < 10 && cell[row][col + 1].isHidden() && !cell[row][col + 1].isFlag()) {
            sp.push(row);
            sp.push(col + 1);
        }

        if (row - 1 >= 0 && cell[row - 1][col].isHidden() && !cell[row - 1][col].isFlag()){
            sp.push(row - 1);
            sp.push(col);
        }
        if (row - 1 >= 0 && cell[row][col - 1].isHidden() && !cell[row][col - 1].isFlag(){
            sp.push(row);
```

```
            sp.push(col - 1);
        }
        if (row - 1 >= 0 && col - 1 >= 0) && cell[row - 1][col - 1].isHidden()
            && !cell[row - 1][col - 1].isFlag()){
            sp.push(row - 1);
            sp.push(col - 1);
        }
        if (row - 1 >= 0 && col + 1 < 10 and cell[row - 1][col + 1].isHidden()
            && !cell[row - 1][col + 1].isFlag()){
            sp.push(row - 1);
            sp.push(col + 1);
        }
        if (row + 1 < 10 && col - 1 >= 0 && cell[row + 1][col - 1].isHidden()
            && !cell[row + 1][col - 1].isFlag(){
            sp.push(row + 1);
            sp.push(col - 1);
        }
        if (row + 1 < 10 && col + 1 < 10 && cell[row + 1][col + 1].isHidden()
            && !cell[row + 1][col + 1].isFlag()){
            sp.push(row + 1);
            sp.push(col + 1);
        }
    }
}
```

Congratulations! You have created a working version of ~~Minesweeper~~ Bombsweeper in MIPS assembly.

Hand-in Instructions

See Sparky Submission Instructions on Piazza for hand-in instructions.

❗ There is no tolerance for homework submission via email. Work must be submitted through Sparky. Please do not wait until the last minute to submit your homework. If you are struggling, stop by office hours for additional help.