CSE 220: Systems Fundamentals I Spring 2018

Homework #3

Assignment Due: May 4th, 2018 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 10,000 instructions or less. Efficiency is an important aspect of programming.

⚠ Any excess output from your program (debugging notes, etc) WILL 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.

Introduction

The focus of this homework is to re-enforce the MIPS register conventions, function calling, memory organization, and multi-dimensional arrays.

In this assignment, you will write functions to create a simple text editor similar to vi. If you are unfamiliar with the editor, follow this tutorial and play with the editor to edit a file on Sparky.

To create the editor, we will use a basic display 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. To display the data, the VT100 uses Memory Mapped I/O (MMIO).

⚠ For this assignment, unless otherwise stated, you do not have to validate any arguments passed to the functions you implement.

Part 1 - String Functions

Before we can create the text editor we need to write basic string functions to use. In this assignment, we will define whitespace to be any of the following characters:

- null character (`'\0'`)
- newline character (`'\n'`)
- space character (`' '`)

A whitespace character can match with any other whitespace character, i.e. when explicitly stated as whitespace, a null should be treated as though it were equivalent to a newline.

(a) `int is_whitespace(char c)`

This function checks if a character, `c`, is a whitespace character or not. It returns 1 if `c` is a whitespace character, otherwise it returns 0.

Function parameter and return value summary:

- `c` : Character to check.
- *returns*: 1 if `c` is a whitespace character, otherwise returns 0.

(b) `int cmp_whitespace(char c1, char c2)`

This function checks if the characters `c1` and `c2` are both whitespace characters. If they are both whitespace characters then the function returns 1, otherwise it returns 0.

⚠ This function **MUST** call `is_whitespace`.

Function parameter and return value summary:

- `c1` : First character.
- `c2` : Second character.
- *returns*: If both `c1` and `c2` are whitespace characters return 1 otherwise return 0.

(c) `void strcpy(String src, String dest, int n)`

This function copies `n` bytes from a source string, `src`, into a destination string, `dest`. If the `src` address is lower than or equal to `dest`, then this function doesn't do anything.

Function parameter and return value summary:

- `src` : The address the string is copied from.
- `dest` : The address the string is copied to.
- `n` : Number of bytes to copy from `src` to `dest`.

(d) `int strlen(String s)`

This function takes the address of a string, `s`, as an argument and calculates the length in bytes. The function will calculate the length of the string until a whitespace character is encountered.

⚠ This function **MUST** call `is_whitespace`.

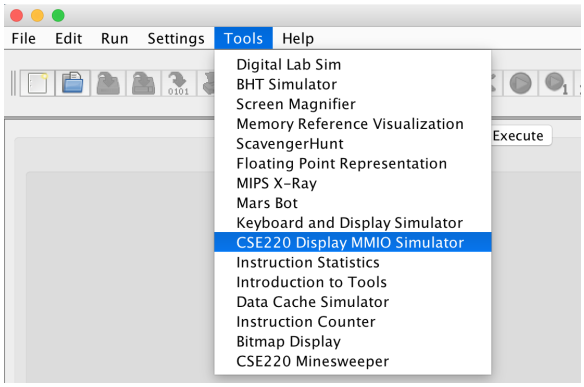
Function parameter and return value summary:

- `s` : String to calculate the length of.
- *returns*: length of the string until the whitespace character.

VT100 Display

The VT100 Display is a basic display. To display the data, the VT100 uses Memory Mapped I/O (MMIO) and ANSI escape codes to specify the foreground and background colors of each cell of the terminal.

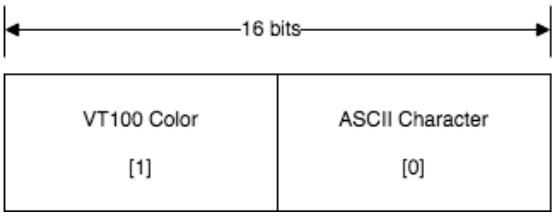
For this assignment, we will use a tool built into Mars called the **CSE220 Display MMIO Simulator** which located in the tools menu called `CSE220 Display MMIO Simulator`.



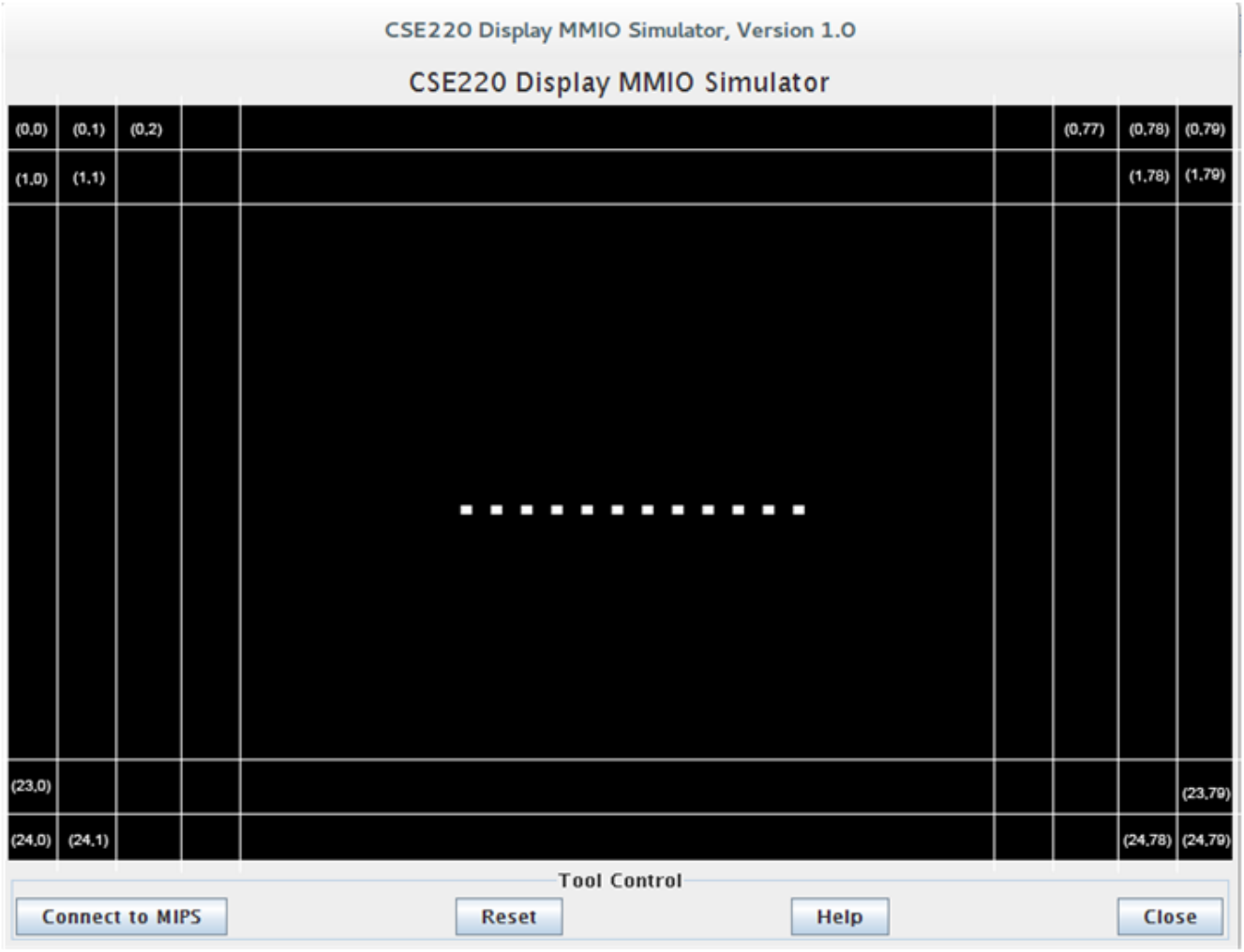
This tool simulates video memory which is mapped directly to an address in memory, aka Memory Mapped I/O (MMIO). The simulator will treat the bytes starting at `0xffff0000` to `0xffff0fa0` as the values of a 25 row by 80 column console. The simulator will attempt to translate the values stored at these memory addresses as values to print to its display. Each cell of the simulator is mapped to a half word (2 bytes) in memory. The upper byte stores the color (described later) and the lower byte stores the ASCII character displayed. Each row consists of 80 columns and each column is 2 bytes in size, therefore, each row is 160 bytes. The total continuous region of memory for the MMIO is 4000 bytes, as there are 25 rows.

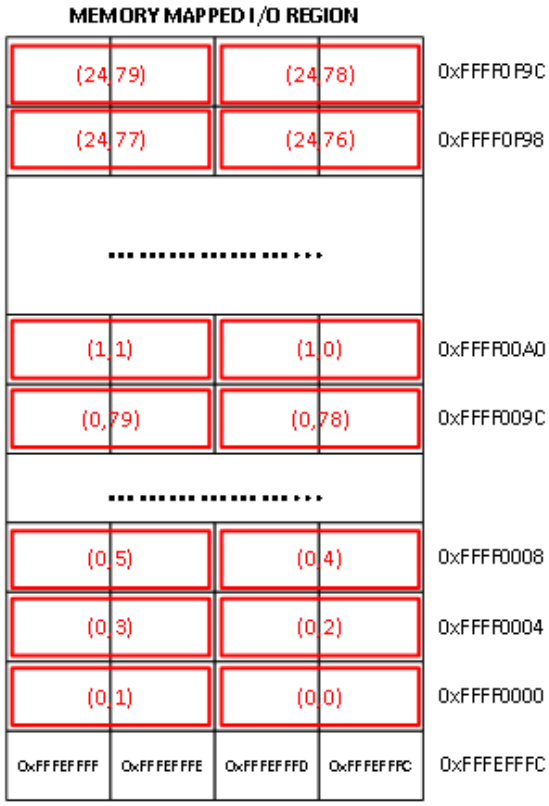
Bytes [0:1] (addresses `0xffff0000` & `0xffff0001`) contain the data for display at location `(0,0)` on the simulator. Byte 0 is the 8-bit value of the ASCII character to display on the screen. Byte 1 is the 8-bit format for the coloring information for the location at which the character is being displayed. 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 MMIO cell has the following layout:



The MMIO region of memory is stored in Row-Major Order. The following figure illustrates this mapping.


CSE220 Display MMIO Simulator, Version 1.0
CSE220 Display MMIO Simulator
(0,0)
(0,1)
(0,2)
(0,77)
(0,78)
(0,79)
(1,0)
(1,1)
(1,78)
(1,79)
(23,0)
(23,79)
(24,0)
(24,1)
(24,78)
(24,79)
Tool Control
Connect to MIPS
Reset
Help
Close

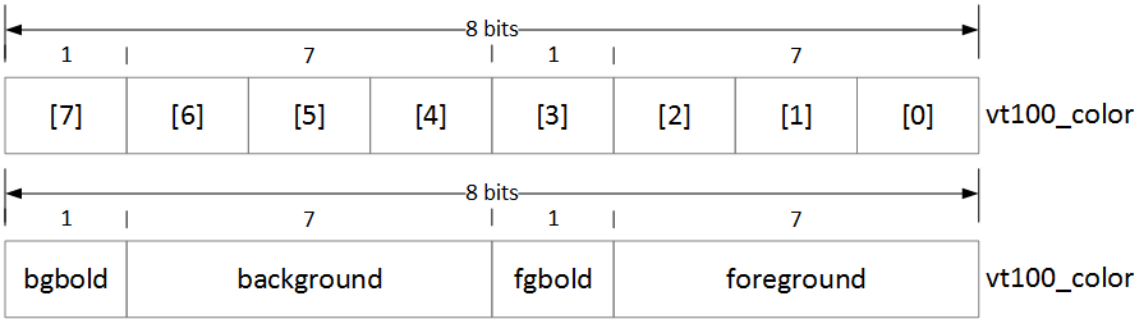

MEMORY MAPPED I/O REGION
(24,79)
(24,78)
0xFFFF0F9C
(24,77)
(24,76)
0xFFFF0F98
(1,1)
(1,0)
0xFFFF00A0
(0,79)
(0,78)
0xFFFF009C
(0,5)
(0,4)
0xFFFF0008
(0,3)
(0,2)
0xFFFF0004
(0,1)
(0,0)
0xFFFF0000
0xFFFEFFFF
0xFFFEFFFE
0xFFFEFFFD
0xFFFEFFFC
0xFFFEFFFC

VT100 colors

Colors on the VT100 display are controlled by specific values stored in the upper byte of each half word. The foreground color is the color of the ASCII character which is displayed. The background color is the color of the entire cell. Below is a chart of the colors available and their corresponding decimal values. The **ATTRIBUTE_VALUE**, also known as the **BOLD BIT**, determines the depth of the color.

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

To store all of the color information into a single byte, the following format is used. For example, for Green background and Bright Magenta foreground the hexadecimal value of the color format is `0x2D`.



Part 2 - VT100 MMIO Functions

To create the text editor we need to have a structure to store information about the current state of the editor, such as the position of the cursor and the default foreground and background colors. The following `state` structure is defined to hold the default foreground and background color for the console, the foreground and background color for syntax highlighting (to be specified in Part 3), and the (x,y) position of the cursor within the VT100 display.

```
struct state {
    # a nibble is 4 bits (half a byte)
    nibble default_fg;
    nibble default_bg;
    nibble highlight_fg;
```

```
    nibble highlight_bg;
    byte cursor_x;
    byte cursor_y;
};
```

In a typical text editor, the position of the cursor in the display is specified by some visual character, such as | or _. This type of visual is difficult in our MMIO display. Therefore, instead we will invert the bold bit for the foreground and background of the MMIO cell for which the cursor is currently positioned at. For example, if the default colors for the display is Bright Blue background with Black foreground then the color byte will have value `0xC0`. If the cursor is positioned on this cell, the color byte will instead be set to `0x48`.

In this part of the assignment the goal is to implement functions which interact with the state of the text editor and the VT100 MMIO.

(e) `void set_state_color(struct state, byte color, int category, int mode)`

This function sets one of the color properties of the state struct. If `category` is 1 then the `highlight` color property is set, otherwise, the `default` color property is set. The mode specifies the operation of this function as shown in the table below:

Mode	Operation
0	set both fg and bg
1	set only fg
2	set only bg

`color` has the same format as a VT100 color byte.

Function parameter and return value summary:

- `state` : The address of the state struct representing the current state.
- `color` : The byte describing the VT100 color in the VT100 color format.
- `category` : The color category that is being set (0 is default, 1 is highlight).
- `mode` : The mode specifies if the foreground, background or both colors are set.

(f) `void save_char(struct state, char c)`

This function updates the ASCII value displayed on the VT100 to `c` at the cursor's position specified in the state struct.

Function parameter and return value summary:

- `state` : The address of the state struct representing the current state.
- `c` : The character to put at the cursor's position.

(g) `void reset(struct state, int color_only)`

This function resets the VT100 display. If the `color_only` option is specified then all the color is set to the default value specified in the state struct. If `color_only` is set to 0 then update the color and set the ASCII byte to null.

Function parameter and return value summary:

- `state` : The address of the state struct representing the current state.
- `color_only` : If 1, clear color only, otherwise, clear both color and ASCII.

(h) `void clear_line(byte x, byte y, byte color)`

This function clears ASCII characters ('\0') and sets the color to `color` start at the `(x,y)` position until the end of the row, `(x, 79)`.

Function parameter and return value summary:

- `x` : The row on the VT100 display.
- `y` : The column on the VT100 display.
- `color` : The color to set the VT100 cells to.

(i) `void set_cursor(struct state, byte x, byte y, int initial)`

This function updates the location of the cursor. First, clear the cursor color from the original position, update the cursors' location in the state struct to `(x,y)` and then set the VT100 cell to show the cursor at the new location. If `initial` is set to 1, then this function **will not** clear the cursor first.

ℹ Clearing and setting the cursor are identical. The foreground and background bold bits need to be inverted.

Function parameter and return value summary:

- `state` : The address of the state struct representing the current state.
- `x` : New row value for the cursor.
- `y` : New column value for the cursor.
- `initial` : If initial is set to 1 then the cursor is not cleared first, otherwise it is.

(j) `void move_cursor(struct state, char direction)`

This function moves the cursor from the `(x,y)` position specified in state by one cell in the direction specified by the `direction` argument. The value of `direction` argument specifies the direction as shown in the table below:

ASCII Char	Direction
h	Left
j	Down
k	Up
l	Right

This function does not support cursor wrapping around the edges of the screen. Instead, if the caller function attempts to move the cursor in a direction outside the range of the screen, the function does not move the cursor. E.g:

- If the cursor is at `(0,0)` and the direction is specified as `'h'` or `'k'`, the cursor remains at `(0,0)`.
- If the cursor is at `(24,79)` and the direction is specified as `'l'` or `'j'`, the cursor remains at `(24,79)`.

⚠ This function **MUST** call `set_cursor`.

Function parameter and return value summary:

- `state` : The address of the state struct representing the current state.
- `direction` : The ASCII letter specifying the direction to move the cursor in.

(k) `int mmio_streq(String mmio, String b)` This function takes the address of two strings, `mmio` and `b`, as arguments and compares the two to see if they match. The function will check the strings until (and including) a whitespace is encountered in either of the strings. If the strings are equal, the function will return 1. If they are not equal, the function will return 0.

The important difference here is that `mmio` is a sequence of VT100 cells and `b` is a regular string.

⚠ This function **MUST** call `cmp_whitespace`.

Function parameter and return value summary:

- `mmio` : Address of the start of the MMIO string.
- `b` : The regular string that the MMIO string should be compared to.
- *returns*: 1 if strings are equal, 0 if they are not.

The Main Program

The main program uses the functions you implement to simulate our version of the vi editor. Upon start up, the main program will prompt you to enter your selection for the default foreground and background colors. The color range of the VT100 colors is 0-15, any input outside this range will result in a re-prompt. It will then prompt you to enter your selection for the highlight foreground and background colors. After the colors have been set, the main will then call `reset` to initialize the MMIO and initializing the cursor by calling `set_cursor`. It then enters an indefinite loop that waits for a user to enter a character and then performs an action based on the input. If it is a regular character then it is simply displayed on the screen and the cursor is advanced. If it is a newline, the main program calls `handle_nl` (to be implemented in Part 3). If it is a backtick character (‘`’), the main program calls `handle_cmd` (to be implemented in Part 3). This function will read the command and call functions you implemented appropriately. `highlight_all`, the function which manages the syntax highlighting feature, is called after a space character, newline character or a command is entered.

Command mode

Similar to the vi text editor, we will allow commands to be executed. To enter a command we will first type the backtick character (‘`’), commonly located under the tilde on the keyboard. Then a single letter is accepted to specify the type of operation to be performed, such as moving the cursor, setting the colors, or performing the backspace operation. The following table specifies the supported commands in our text editor.

Command	Operation
`h	Move cursor left 1 position (unless at boundary)
`j	Move cursor down 1 position (unless at boundary)
`k	Move cursor up 1 position (unless at boundary)
`l	Move cursor right 1 position (unless at boundary)
`d	Backspace
`F#	Set the default foreground color to the # specified
`B#	Set the default background color to the # specified
`f#	Set the highlight foreground color to the # specified
`b#	Set the highlight background color to the # specified
`q	Quit the program

⚠ When setting the foreground or background colors for the state of the text editor the commands will additional accept an integer value in the range of [0-15]. You must press enter after inputting the number the complete the read int syscall. The syscall does not accept any non numerical characters.

Our text editor would not be complete without syntax highlighting! The highlight colors in the state structure

specify the foreground and background colors to apply to any symbols or words found in a specified dictionary. We have provided a sample dictionary in `hw3_dict.asm`. The dictionary array may contain any number of entries, with the terminating entry in the array set to the value of NULL (aka 0x00000000).

⚠ All the functionality described in the main program subsection above has been provided to you. You **DO NOT** need to implement the main. You should implement your own main(s) to test each of your functions individually. Once, all of them are working correctly, you can use our main to run the vi text editor.

Part 3 - UI/UX Functions

In this part of the assignment, you will create functions to handle different input operations, such as commands, newlines, and to syntax highlight specified dictionary words/characters in the terminal.

(l) `void handle_nl(struct state)`

This function handles the newline action of the editor. First, save a newline character in the current position of the cursor. Then, clear (set default color and ASCII to '`\0`') the rest of the line from the cursor until the end of line. Lastly, it will move the cursor to the start of the next line. If the cursor is already at the last row, the function moves the cursor to the start of the row.

⚠ This function **MUST** call `clear_line` and `save_char`. The implementation will also need to use `set_cursor` or `move_cursor` or both.

(m) `void handle_backspace(struct state)`

This function handles the backspace action of the editor. The action only applies to the row the cursor is currently positioned on. In row `x`, it will copy the n+1th byte to the nth position starting from `y` until 79. `x` and `y` are obtained from the location of the cursor. `(x,79)` will be reset to the default color and the ASCII character will be set to '`\0`'. The cursor will also be moved left one position.

If the function is called when the cursor is at the start of the line, no action is performed.

ℹ This function **MUST** call `strcpy` and `set_cursor`.

(n) `void highlight(byte x, byte y, byte color, int n)`

This function highlights `n` VT100 cells with `color` starting from `(x,y)`. This function should traverse in row major order.

Function parameter and return value summary:

- `x` : Row of starting location.
- `y` : Column of starting location.
- `color` : The VT100 color that should be set.
- `n` : The number of VT100 cells that should be highlighted.

(o) `void highlight_all(byte color, String[] dictionary)`

This function will manage the syntax highlighting of all the ASCII text that is visible on the MMIO display. This function is detailed by the following pseudocode.

```
while (not end_of_display) {
    while (is_whitespace) {
```

```
            move to next MMIO cell
        }
        // save the current cell position
        for each (word in dictionary) {
            check if string starting at current cell is in the dictionary
            if (match) {
                highlight word
            }
        }
        // starting from current cell position
        while (not is_whitespace) {
            move to next cell
        }
    }
```

⚠ The last entry of the dictionary array holds a null (0) word to denote that there are no more entries.

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 examples, one output line ONLY.

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 struggling, stop by office hours.

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.