ECE391: Computer Systems Engineering **Fall 2024**
Machine Problem 1 **Due: 11:59PM CDT Friday September 20th**

Starry Night Screen Saver

Note: Must commit to the **main** branch in your provided GitHub repository by assignment deadline.
Please read the entire document before you begin.

Introduction

A long long time ago, when computer screens were based on cathode-ray tubes (CRTs), if the same image is displayed on screen for a long time, the phosphor coating inside the screen could easily be permanently damaged, leading to a screen burn-in, or darkened shadow on the screen. Screen-saver avoids this problem by automatically changing the images on the screen after the computer has been idle for a long time. Although modern computers do not have a screen burn-in issue, we still use screensavers for decorative purposes. In MP1, we will develop a cool screen saver using **RISC-V** assembly code. There are three types of objects we will draw in our starry night screen saver: star, window, and beacon, and we will implement functions to add, remove, and draw these objects.

MP1 Assignment

You will add 7 new functions to the existing code base.

Your code will reside in `mp1.S`, a **RISC-V** assembly file. Assembly files with a capital-S extension (.S) are preprocessed using the standard C preprocessor before being assembled, so things like `#include` and `#define` are OK to use. Your code must be implemented using **RISC-V** assembly.

Frame Buffer

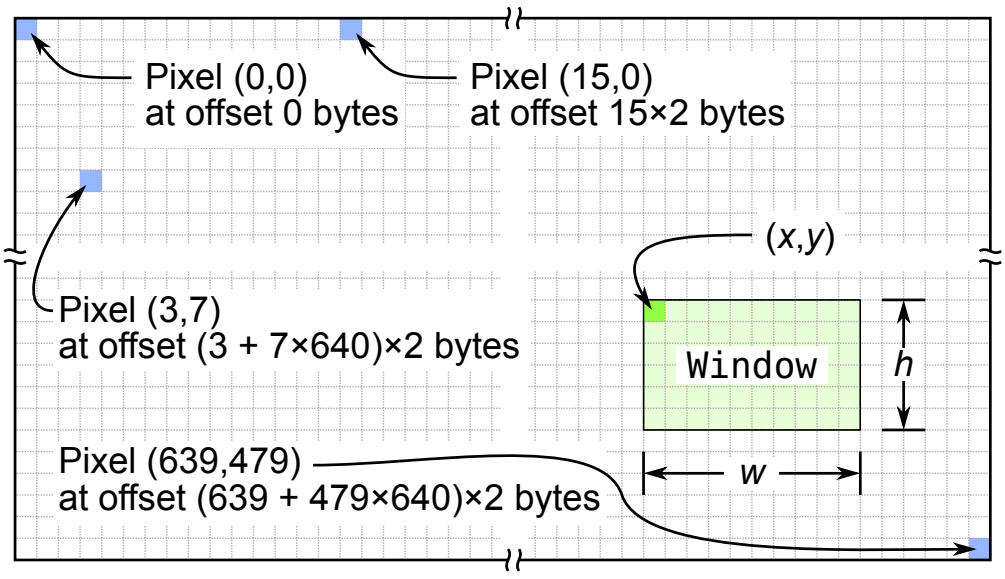

Figure 1: MP1 Screen

The frame buffer contains a representation of an image on the screen, with each pixel occupying a fixed number of bytes. Each pixel is identified by its Cartesian coordinates (x, y). The origin of the coordinate system is the upper left corner of the image. The x coordinate increases from left to right, the y coordinate from top to bottom. (Note that the y coordinate orientation is opposite that of the usual Cartesian coordinate system in mathematics.) The pixels are arranged in memory in row-major order, meaning that the pixels of the first row appear in memory before the pixels of

the second, and the left-most pixel of the second row comes immediately after the right-most pixel of the first row. In this assignment, we will use a frame buffer that is 640 pixels wide and 480 pixels high. (That is, there are 640 columns and 480 rows.) Each pixel contains the color data and occupies 2 bytes of memory. To calculate the memory offset of this pixel, we use the following formula:

$$\text{memory offset} = (x + y \times 640) \times 2 \text{ bytes}$$

We multiply by 2 at the end because each pixel occupies 2 bytes of memory.

MP1 Data Structure

Please refer to the graph above.

```
struct skyline_star {
    uint16_t    x;      /* x-coordinate of top left pixel of the star */
    uint16_t    y;      /* y-coordinate of top left pixel of the star */
    uint8_t     dia;    /* diameter of the star, the star should be a square */
    uint16_t    color;  /* 16-bit RGB color */
}

struct skyline_window {
    struct skyline_window * next;   /* pointer to the next window */
    uint16_t                x;      /* x-coordinate of top left pixel of the window */
    uint16_t                y;      /* y-coordinate of top left pixel of the window */
    uint8_t                 w;      /* width of the window */
    uint8_t                 h;      /* height of the window */
    uint16_t                color;  /* 16-bit RGB color */
};

struct skyline_beacon {
    const uint16_t * img;    /* Pointer to color data for beacon */
    uint16_t         x;      /* x-coordinate of top left pixel of the beacon */
    uint16_t         y;      /* y-coordinate of top left pixel of the beacon */
    uint8_t          dia;    /* diameter (i.e. width and height) of beacon */
    uint16_t         period; /* period of beacon on/off in timer ticks */
    uint16_t         ontime; /* number ticks per period for which beacon should be drawn */
}
```

MP1 Functions

Please refer to `skyline.h` for global variable and function definition.

`void add_star(uint16_t x, uint16_t y, uint16_t color)`
The `add_star()` function adds a star at position `(x,y)` to the `skyline_stars` array with the specified color and update `skyline_star_cnt`. If there is no room for the star in the array, the request should be ignored (star array remains unchanged). The star stored in an array, windows in a linked list.

`void remove_star(uint16_t x, uint16_t y)`
The `remove_star()` function removes the star at position `(x,y)` from the `skyline_stars` array, if such a star exists, and should update the star count. You may assume that at most one star will be added at each position. Remember that the stars must be contiguous in the star array. The order of the stars in the array does not matter.

`void draw_star(uint16_t * fbuf, const struct skyline_star * star)`
The `draw_star()` function draws a star to the frame buffer `fbuf`. Do not assume that the pointer is to a star in the `skyline_stars` array. (During testing, the function may be called with a pointer to a star struct that is not in the

array.) You may assume that the coordinates of the star are inside the screen area. That is, you may assume that `0 <= x < SKYLINE_WIDTH` and `0 <= y < SKYLINE_HEIGHT`.

`void add_window (uint16_t x, uint16_t y,uint8_t w, uint8_t h, uint16_t color)`
The `add_window()` function adds a window of the specified color and size at the specified position to the window list. The upper left corner of the window is at the `(x,y)` coordinate. The width and height of the window are given by `w` and `h`. The window color is given by `color`.

`void remove_window(uint16_t x, uint16_t y)`
The `remove_window()` function removes a window the upper left corner of which is at position `(x,y)`, if such a window exists. You may assume that there is at most one window at those coordinates. If there is such an element, remove it from the linked list and free its memory by calling `free`. If there is no such element, do nothing.

`void draw_window(uint16_t * fbuf, const struct skyline_window * win)`
The `draw_window` function draws the given window win onto the given frame buffer `fbuf`. Iterate over the pixels in the window and set the respective pixel in the `fbuf` to the `color` field of `win`. Be sure to skip pixels whose locations are outside the screen `(SKYLIGHT_WIDTH, SKYLINE_HEIGHT)`.

`void start_beacon (const uint16_t * img, ... uint16_t ontime)` (Given)
The `start_beacon` function initializes all fields of the `skyline_beacon` struct with the passed arguments. For each field of the `skyline_beacon` struct, this function copies the corresponding argument into the struct.

`void draw_beacon (uint16_t * fbuf, uint64_t t, const struct skyline_beacon * bcn)`
The `draw_beacon` function draws the beacon described by the passed `skyline_beacon` struct. This function has 3 parameters: `fbuf`, which is a pointer to the buffer containing pixel data to be drawn to the screen; `t`, which represents the elapsed demo time in timer ticks; and `bcn`, which is a pointer to the `skyline_beacon` struct characterizing the beacon to be drawn.
This function should only draw the beacon when it is meant to be on, otherwise it should not be drawn. Determine if the beacon should be drawn by comparing `t` with `bcn->period` and `bcn->ontime`. If it should be drawn, iterate over the color data of the beacon, copying the color data into `fbuf` at the offset corresponding to the color data's on-screen coordinates, which you can determine using `bcn->x` and `bcn->y`. Be sure that the beacon is not drawn outside of the window, skipping any pixels with coordinates outside of the boundaries of the window.

Allocating and Freeing Memory

User-level C programs make use of the `malloc()` and `free()` C library functions to allocate memory needed for storing dynamic structures such as linked list elements. The MP1 distribution contains two memory allocation functions in `memory.c/h` that behave similarly to `malloc()` and `free()`. Their prototypes are:

```
void * malloc(size_t size);
void free(void * ptr);
```

`malloc` takes a parameter specifying the number of bytes of memory to allocate. It returns a `void*`, called a "void pointer", which is the memory address of the newly-allocated memory.

`free` takes a pointer to a block of memory that was dynamically allocated with `malloc()` and releases/frees that memory back to the system. It does not return anything.

How to call the given function?

```
addi a0,x0,ARG # put the proper argument into a0
call malloc     # once the call is done, a0 will have the pointer to the allocated block
```

[RISC-V passes argument through register `a0-7`, the return value would be stored `a0-1`. Please refer to the calling convention documentation on the website for more detail.]

Coding Style and Design

In general, being able to write readable code is a skill that's just as important as being able to write working code. People and industry teams have their own preferences and rules when it comes to coding style. In this class, we won't nitpick over small things such as spaces, blank lines, or camel case, nor will we enforce any rigorous coding guidelines. However, we still do have a basic standard that we expect you to adhere to and will be enforced through grading. Our expectations are outlined below:

- Give meaningful and descriptive (but not too long) names to your variables, labels, constants, functions, and files. Be consistent in your naming conventions.
- Do **NOT** use magic numbers (any number that appears in your code without a comment or meaningful symbolic name). `-1`, `0`, and `1` are usually OK when used in obvious ways.
- Keep programs and functions relatively short. Don't write spaghetti code that jumps back and forth everywhere. Create helper functions instead and make it easy to follow the flow of the program. Note that helper functions must obey the C calling convention.
- Use comments to explain the interfaces to all functions or subroutines, lengthy segments of code, and any non-obvious line of code. However, do **NOT** overdo it. Too many comments are just as bad as too few. Use comments to explain *why*, not *what*.

How to get things started

You can follow these steps to pull the assignment code from the release repo

1. First switch to the main branch

   ```
   git switch main
   ```

2. Then fetch the MP1 branch from the release repo

   ```
   git fetch release mp1
   ```

3. Then create a local branch for MP1 and switch to it

   ```
   git branch mp1
   git switch mp1
   ```

4. Finally merge release/mp1 into the local MP1 branch

   ```
   git merge --allow-unrelated-histories release/mp1
   ```

Testing

For testing your code, we have provided a golden implementation of this MP in the executable `gold.o`. Initially, you will notice that when running:

```
make clean
make demo.elf
make run-demo
```

that this golden screensaver will be displayed on screen. The demo target compiles the golden code with weak symbols, meaning that the implementations of certain functions in the golden version will be overwritten by your implementations of the same functions in `mp1.s`. The golden function executable will not be linked during the grading process, so ensure that you have provided a complete implementation for every required function.

We are not providing any test cases for this MP. Writing thorough test cases is essential in industry and we strongly encourage you to learn how to test your code yourself. While your screensaver may look identical to the golden implementation, there may still be lurking issues in your code that our autograder will catch.

Handin

Please push your code to GitHub before the deadline.

Important Things to Note:

- The deadline is the same for everyone!
- You are free to develop your own system of code organization, but we will **STRICTLY** use only the `main` branch for grading, and will only make use of your `mp1.S` file.
- Your MP1 will be graded twice, one at the deadline and another one one week after the deadline. The first time will be graded for 100% credit, and the second time for 50% credit. So, you can get 50% back if you missed anything at the first time.

A Couple of Tips

Your work directory will be deleted at the end of the semester. Be sure to save any source code that you wish to retain by moving it to your home directory or to your own personal data storage.