CDA 4205L Lab #9: System I/O

University of South Florida

Lab Date: March 12

Report due: One week after the lab section, see Canvas

This is a group lab. *One submission per group is sufficient*.

Welcome to CDA 4205L Lab #9! The goal of this lab is to help you understand how input and output to a computer work with memory-mapped I/O (MMIO)

Prelab

Ecalls and Memory-Mapped I/O

RARS provides some functionality that normally would be handled by an OS or other low-level software – ecalls. Ecalls will handle things like input/output, including writing to or reading from the console, as well as writing to or reading from files.

RARS also provides a simulation environment for peripheral devices, like a keyboard or a display. These differ from other system-level tasks (like file I/O) because they do not require ecalls. Instead, you can directly access information about the peripheral or modify its behavior simply by reading from or writing to specific registers with `lb`/`lh`/`lw`/`ld` instructions (or the store equivalents). In turn, these peripherals can read/write information from/to the same memory locations to communicate with the processor.

In this lab, we will be using ecalls for file I/O and one memory-mapped peripheral. In particular, we will use `ecall`s to read a bitmap image, then use the bitmap display to show the picture using RISC-V assembly.

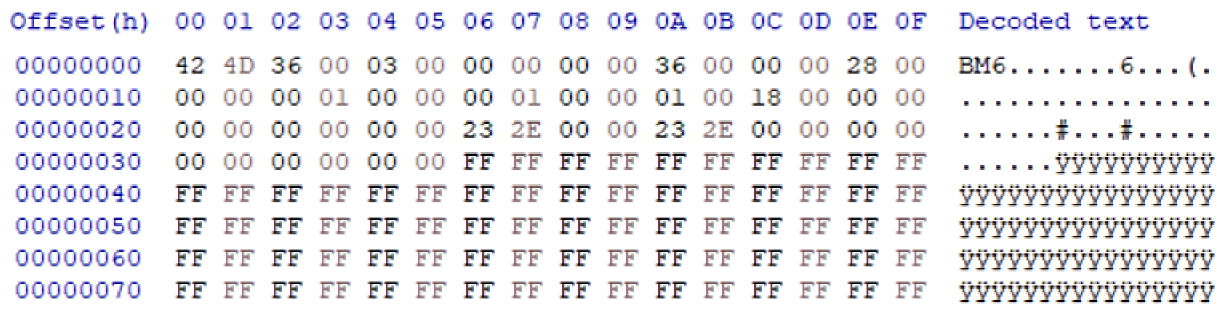
Files are identified by a *descriptor*. To get a file descriptor, you will provide the system routine with the file location (e.g. the full path to the file, or a relative path from the binary) and the permission (read-only, read/write, etc.) The following table lists information about (some) file-handling routines:

Name	**Call #**	**Description**	**Inputs**	**Outputs**
Open	`1024`	Opens a file from a path. Flags include read-only (0), write-only (1), and write-append (9).	`a0` = null-terminated string (`.string` or `.asciz`) `a1` = flags	`a0` = file descriptor (-1 if error)
Read	`63`	Read from a file descriptor into a buffer	`a0` = file descriptor `a1` = address of buffer `a2` = max length to read	`a0` = length actually read (or -1 if error)
Close	`57`	Closes a file	`a0` = the file descriptor	N/A

Other `ecall`s for seeking a position in a file or writing to a file are available but not needed for this lab.

File Formats and Headers

Most files (bitmaps included) have defined binary formats. Most will have a *header* that identifies the file type and provides important information for processing the file. The bitmap file header format is available here https://en.wikipedia.org/wiki/BMP_file_format. To summarize, the basic header consists of `0x36` bytes of data including `0x42 0x4D` (“BM”) at the start (for bitmap), followed by information like the size of the file (4 bytes), 4 bytes of reserved space, and a 4-byte value that specifies where the actual image data begins (this is located at offset `0x0A`). In the following example, a 256×256 pixel bitmap was opened in a hex editor which shows the values for each byte at the start of the file:



```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text

00000000  42 4D 36 00 03 00 00 00 00 00 36 00 00 00 28 00  BM6.......6...(.
00000010  00 00 00 01 00 00 00 01 00 00 01 00 18 00 00 00  ................
00000020  00 00 00 00 00 00 23 2E 00 00 23 2E 00 00 00 00  ......#...#.....
00000030  00 00 00 00 00 00 FF FF FF FF FF FF FF FF FF FF  ......ÿÿÿÿÿÿÿÿÿÿ
00000040  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000050  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000060  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000070  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
```

As expected, the first two bytes are `0x42` and `0x4D`, followed by 4 bytes indicating the size of the file: `0x36 0x00 0x03 0x00`. Note that bitmaps store information in *little Endian* format, so the LSB is first. This field can be read from right to left: `0x00 0x03 0x00 0x36` or `0x30036`. In decimal, this is equal to $256_{10} \times 256_{10} = 65{,}536_{10}$ pixels. We are storing 3 bytes per pixel (1 each for red, green, and blue). $65{,}536_{10} \times 3_{10} = 196{,}608_{10}$. The header is `0x36` bytes (or 54_{10} bytes) for a total of $196{,}662_{10}$ bytes, which is `0x30036` in hex – the same value given by the file header.

Note: this only works out because the width and height of the image are powers of 2 – if not, we would have to include padding bytes in the calculation.

Bitmap Display in RARS

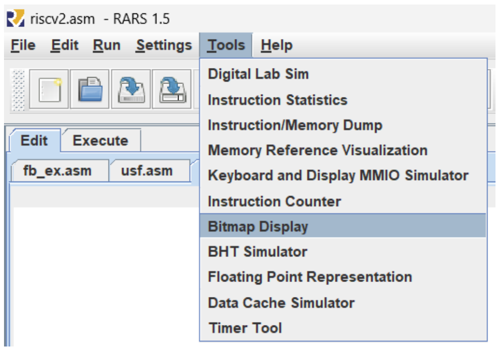
Now that you are familiar with the bitmap file format, we can discuss the bitmap display in RARS. This tool lets you write to a region of memory, identified by a base address (the default is `0x10010000`). The tool associates the value of 1 pixel with 1 `word`. Individual bytes in this word represent the red, green, and blue color components. To change the color of a pixel, you simply store the desired `word` at that appropriate location in memory.

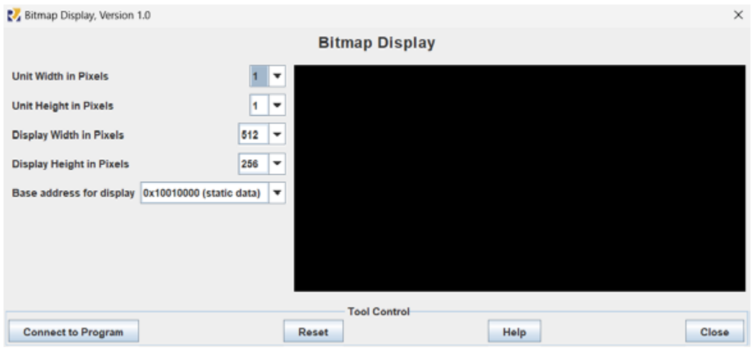
Note: in practice, it is not a good idea to draw directly to the screen buffer (or frame buffer) in the graphics memory like this because it can cause flickering or other visual artifacts. Typically, an approach called double (or even triple) buffering is used. You have two (or three) equally sized regions of memory for this purpose. You draw (write data) to one while the other is displayed; then, they swap roles. Each “refresh” (swapping of buffers) displays a complete, “finished” screen to the user.

Lab

Part 1: Using the Bitmap Display

The first exercise will focus on using the bitmap display. The bitmap display can be found under Tools > Bitmap Display. Once open, you should see this screen (right):





Unit Width in Pixels	1
Unit Height in Pixels	1
Display Width in Pixels	512
Display Height in Pixels	256
Base address for display	0x10010000 (static data)

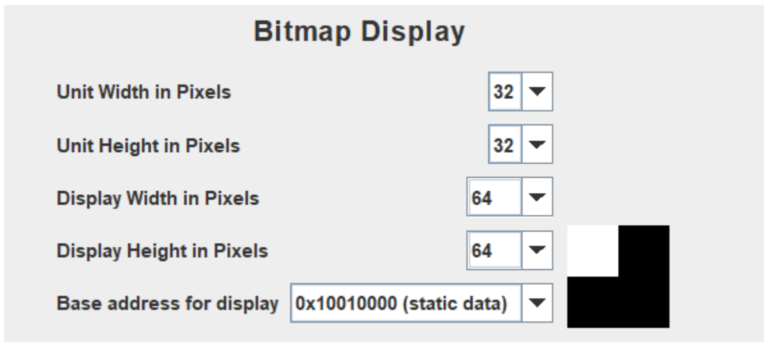
Set the following options:

- Unit Width in Pixels: 32
- Unit Height in Pixels: 32
- Display Width in Pixels: 64
- Display Height in Pixels: 64
- Base address for display: (leave as-is, the static data segment)

We will change these values later on, but for now, it simplifies the experiment.

2. Begin by creating (and saving) a new assembly file in RARS. Create a `.data` segment and `.text` segment. In the `.text` segment, do the following:
   a. Load the base address of the static data segment (`0x10010000`) as an immediate in `s0`. Load another immediate, `0x00FFFFFF`, into `t0`.
   b. Write (`sw`) the value in `t0` to the location in `s0` (with offset `0`).
   c. Exit the program

3. Click the "Connect to Program" button on the Bitmap Display window, and run your code. If everything is correct, you should see the following:



Unit Width in Pixels	32
Unit Height in Pixels	32
Display Width in Pixels	64
Display Height in Pixels	64
Base address for display	0x10010000 (static data)

4. Modify your code so that you draw a red square in the top right, a green square in the bottom left, and a blue square in the bottom right. Test out different values (`0x00ff0000`, `0x0000ff00`, etc.) to see what byte positions correspond to the colors red, green, and blue).

   Note: to draw to other parts of the display, you need to change the address where you are storing data. Remember, you are using `sw`, so offset accordingly!

   T1: Take a screenshot showing white, red, green, and blue in top left, top right, bottom left, and bottom right squares in the same bitmap display.

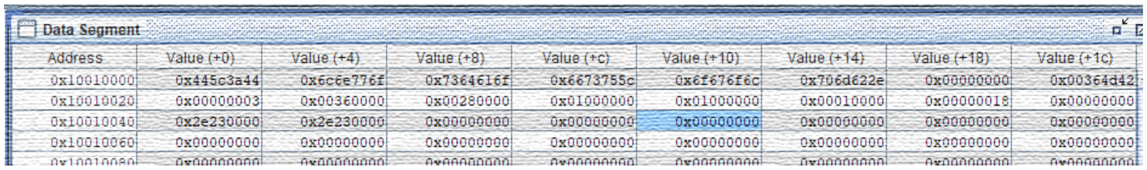
   T2: Which bytes correspond to which colors? How are different locations of the display addressed?

   T3: Save this file and submit it with your report.

Part 2: Reading and Displaying a Bitmap

5. Download the part 2 template from Canvas. Notice the data section includes a `.string` (this is an alias for `.asciz`, or a null-terminated ASCII string) object labeled *`usfbmp`*. This should be modified to point to the directory where you downloaded the image. Be sure to use the full path, and include `\\` (double backslashes) in place of `\` (which the system will misinterpret as an escape character). The path should also be in quotes, e.g., `“C:\\Path\\To\\Image.bmp”`.

6. Using the `ecall`s available to RARS, write code that “opens” the file and reads the first `0x36` bytes of the contents into the pre-defined buffer (“*header*”). Finally, include `ecall`s for closing the file and exiting the program. After running your program, you should see the memory contents in the static data section has changed. Your results will be slightly different depending on the length of your *`usfbmp`* `.string`.



Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x445c3a44	0x6c6e776f	0x7364616f	0x6673755c	0x6f676f6c	0x706d622e	0x00000000	0x00364d42
0x10010020	0x00000003	0x00360000	0x00280000	0x01000000	0x01000000	0x00010000	0x00000018	0x00000000
0x10010040	0x2e230000	0x2e230000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

   Note: you only get the descriptor from the “open” `ecall`, and others will overwrite it. <u>Be sure to store the result in a saved register after opening the file so you can use it later.</u>

   T4: Take a screenshot of the data section in RARS showing the bitmap header info.

   If you get an error, or the contents do not match the bitmap data provided, step through the code and check the return value (in `a0`) of the ecalls. If you receive a `-1` for opening, your file path may be incorrect. Check for any runtime exceptions in the message window.

7. Continue reading the bitmap using the “read” `ecall`. The descriptor includes the current location in the file, so you can pick up where the last “read” call left off. A 3-byte *`buffer`* is provided for your use. As you read the file, immediately write data (`sw`) to the appropriate location in memory to draw the image to the screen.

8. Open the bitmap display window, and set the following:
    - Unit Width in Pixels: 1
    - Unit Height in Pixels: 1
    - Display Width in Pixels: 256
    - Display Height in Pixels: 256
    - Base address for display: (leave as-is, the static data segment)

    Once you have connected the display, run your program.

    Note: it is extremely inefficient to read only 3 bytes at a time. It would be simpler (but require more computation) to read a larger chunk of the image at once. Displaying will take some time.

    T5: Take a screenshot of the bitmap display showing the image after your program runs.

    T6: The provided image is the same size as the display (256×256). If the image were smaller than the display, how would the image look if you used this code?

    Although the dimensions of the image match the display, it still does not look right. Bitmap files store data from bottom to top, from left to right. By starting at the top and writing to the bottom, you effectively flipped the image. Modify your code so the image displays correctly.

    T7: Take a screenshot of the bitmap display showing the image after modifying your code.

    T8: What information in the header (if any) did you have to use to display the image properly (even if you hard-coded it rather than reading it from the header)?

In-class work - Complete **T1, T2, T3, and T4,** show it to the TA for review and submit it on Canvas before the lab session ends. Submit your in-class work as a Word document (.docx) format.

Final Report - Submit your report pdf with answers to all questions and screenshots. Submit both .asm files (part 1 and part 2) for a total of 3 files in a single .zip file. One submission per group.