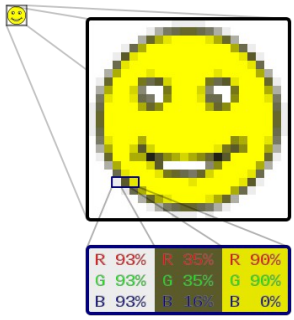


Lab 5: Functions and Graphics

Due Friday March 12, 2021, 11:59 PM *(Late hours cannot be used)*

Minimum Submission Requirements

- Ensure that your Lab5 folder contains the following files (note the capitalization convention):
  - Lab5.asm
  - README.txt
  - It is ok if you also have lab5_w21_test.asm, but we will not require or check it.
- Commit and push your repository
- Complete the Google Form with the correct commit ID of your final submission

Lab Objective

In this lab, you will implement functions that perform some primitive graphics operations on a small simulated display. These functions will allow users to change the background color of the display, and “draw” horizontal and vertical lines on the display. To simulate a display, we’ll be using the memory-mapped bitmap graphics display tool included with MARS.

To do this you will utilize:

1. Arrays
2. Memory-mapped Input/Output (IO)
3. Subroutines (a.k.a. Functions or Procedures)
4. Macros

Color and Computers

A pixel is commonly represented as a triplet of uint8s (i.e. unsigned 8-bit integers ranging from 0-255) specifying the intensity of red, green, and

blue.[1] Together this totals to 24 bits (i.e. 3 bytes) per pixel. Often this triplet is written in hex notation.

> E.g. in this system, white = (255, 255, 255) = #ffffff, black = (0, 0, 0) = #000000, red = (255, 0, 0), yellow = (255, 255, 0), and (128, 64, 32) = #804020 is a brownish color. Here's a tool you can play with to help you understand.

Often an extra 8-bits is used for a transparency channel, making the total 32 bits (= 4 bytes). We won't use the notion of transparency here, but we will use this 4*8=32-bit standard (leaving the most significant 8 bits as 0). I.e. in this assignment, white = #00ffffff, black = #00000000, red = #00ff0000, and yellow = #00ffff00.

In our simple simulation, our display is equivalent to an uncompressed 128x128 32-bit "true color" image. To store (128 x 128 =) 16384 pixels, each being 4 bytes, takes 16384 x 4 = 65536 bytes. Note that 65536 = 2^16 = 16^4 = 0x10000. Our image will be stored in a memory segment spanning 65536 bytes, starting at memory address 0xffff0000 and taking up the remainder of the memory in our 32-bit address space.

Lab Preparation

1. Familiarize yourself with RGB colors (e.g. make sure you understand the basic ideas explained in the above note on "Color and Computers"). You might also consider reading some background on Raster graphics.

2. Introduction To MIPS Assembly Language Programming
   chapters 5, 6; sections 8.1, 8.2
3. Macros

4. Procedures
   watch videos 2.7 - 2.12
5. Functions
   watch video tutorials 15 – 18

Specification

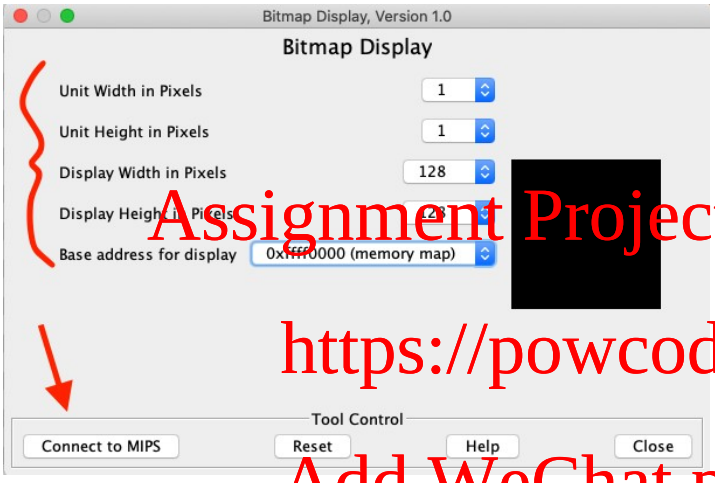
You will need to implement a set of specific subroutines indicated in these lab instructions. You are required to start with the skeleton code provided (lab5_w21_template.asm) and **may not change the function names or arguments at all.** Please rename the file to Lab5.asm and start with it. To receive any credit for your subroutines, **Lab5.asm must assemble** both **on its own** and with the test file. On its own, the template file shouldn't print or draw anything -- it is just a set of subroutines.

1 There's a lot being swept under the rug here for simplicity. It won't help much with this assignment, but if you're curious, here are some links: Color in the brain, color matching functions, rgb.

A test file (lab5_w21_test.asm) tests each one of your subroutines and includes (at the very end) your subroutines from Lab5.asm (based on the above template file). You should modify the test to include Lab5.asm instead of lab5_w21_template.asm. **Don't modify the test file** -- we will **not** use your test file during grading, we will use a similar but not identical test file of our own. Our test file will call your functions and macros. That's why it's so important your functions and macros follow the specifications given. In order for your subroutines to function properly, **you must use** the instructions **JAL and JR** to enter and exit subroutines. You must save and restore registers as required in MIPS. Our test file will look very much like this one, so you should ensure that your functions work with it!

Bitmap Display Tool

To visualize what you're doing, you can use the bitmap display tool (Tools->Bitmap Display).



Functionality

The functionality of your program will support the following:

1. All pixels should be in the range x in [0,128) and y in [0,128) (the parenthesis means not including 128).
2. Pixels start from (0,0) in the upper left to (127,127) in the lower right.
3. Pixel values are referenced in a single word using the upper and lower half of the word. So, for example, 0x00XX00YY) where XX and YY can be 0x00 to 0x7F.
4. All colors should be RGB using a single 32-bit word where the top byte is zero. So, for example, 0x00RRGGBB where RR, GG, and BB can be 0x00 to 0xFF.
5. All functions (subroutines) and macros described below. Note: signatures for each are included in the template.

Macro Descriptions

You are required to implement and use the three following macro definitions. Make sure not to alter their signatures as provided in the template as they may be called by a grading script. You may use additional macros if you like.

getCoordinates, formatCoordinates, and **getPixelAddress**

Subroutine Descriptions

These subroutines should be in the Lab5.asm file. You may use additional functions if you like. Again, these procedures will be called by the grading script, so make sure not to alter their signatures. Please only use registers beginning with $t, $a, and $v when implementing these functions (except **draw_crosshair**, which should only make use of s, a, and v registers). Otherwise, our grading script may not work.

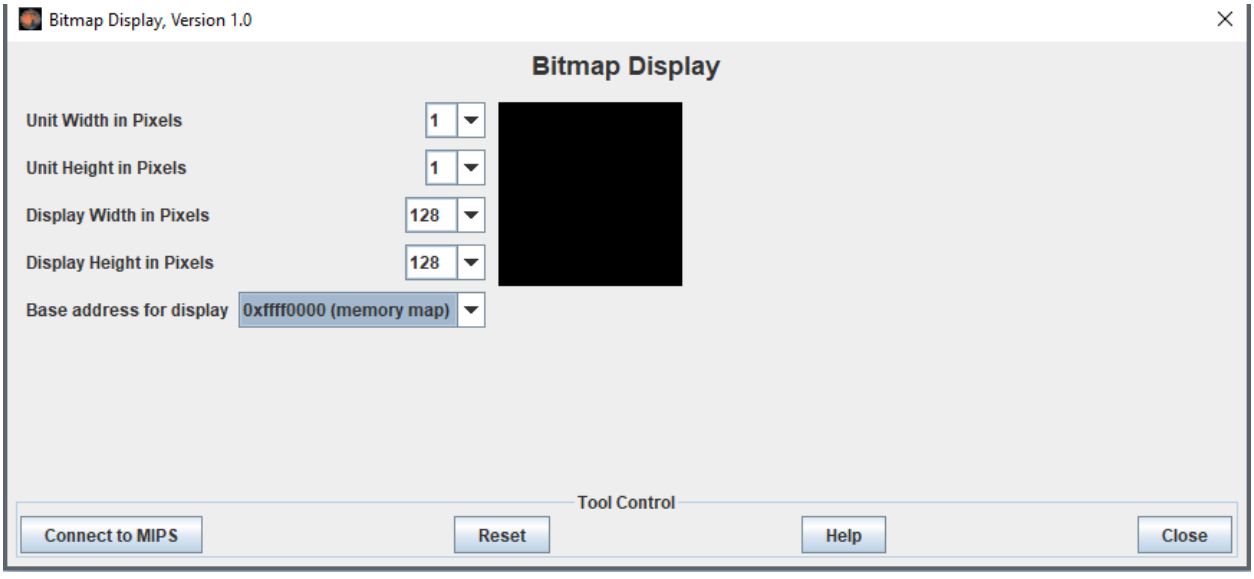
clear_bitmap, draw_pixel, get_pixel, draw_horizontal_line, draw_vertical_line, and draw_crosshair.

A Note on Debugging

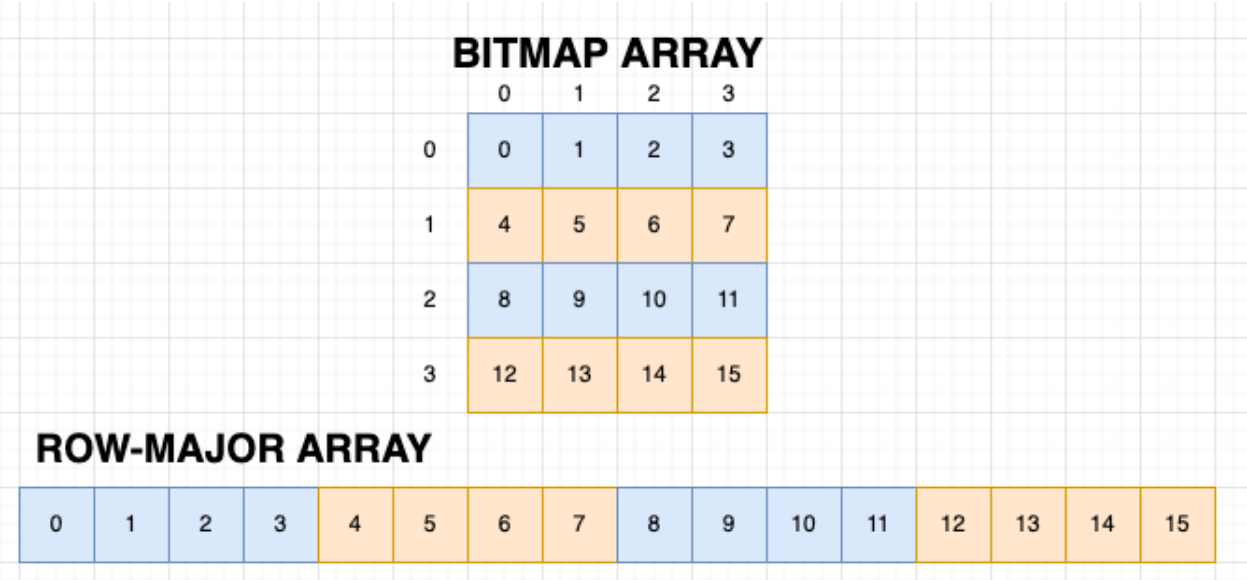
Note that if you need to add print statements to lab5.asm for debugging purposes, **make sure to remove them before submitting** as otherwise they will interfere with our grading scripts.

Test Output

The test output for this lab is visual and requires you to use the MARS Bitmap Display tool (in Mars select Bitmap Display from the Tools menu). You should modify the settings of the bitmap display to be 128 x 128 pixels and to have a base address of the memory map (0xffff0000) as shown here:



Press “Connect to MIPS” to use this in your program.



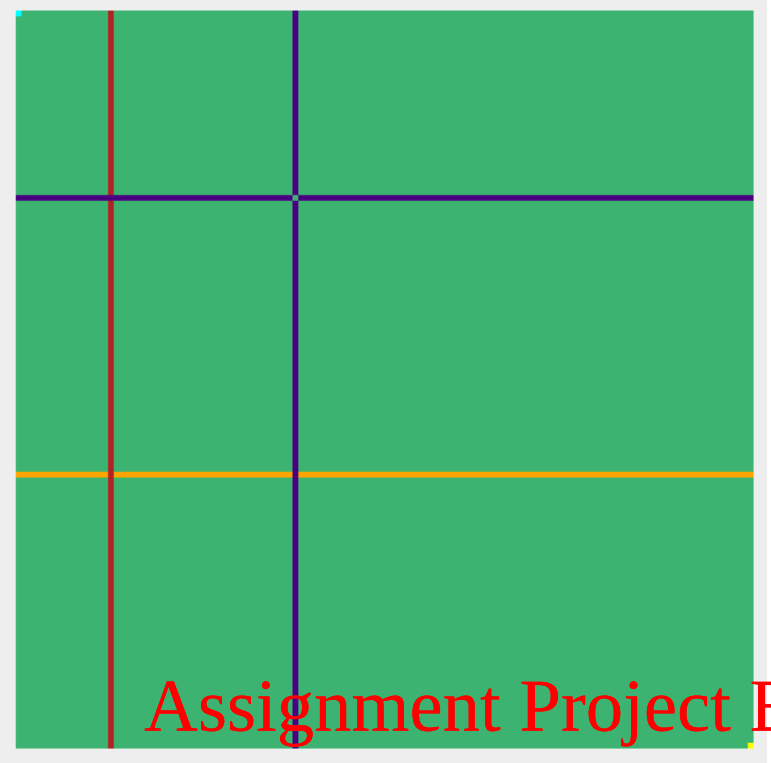
The bitmap display is a grid of 128 x 128 pixels that displays a color based off the value written to the address corresponding to that pixel. In the example above you can see how the coordinates of the pixel relate to the array in memory for a 4 x 4 pixel bitmap. For example if you wanted to color the pixel at row 2, column 3 (i.e. at 0x00030002 ~ (3,2)) you would take the base address of the of the first pixel and offset that by +11 which is (2 * row_size) + 3 to locate the correct pixel to color.

We will be grading your solution by dumping the memory-mapped IO segment as hexadecimal ASCII and comparing with the correct results. **You will miss all the points if you do not use the above size and base address configuration! In addition, your Lab5.asm should not display any text using syscalls as this will interfere with the grading output.** If you want, you can also display the memory-mapped segment using a command line argument like this:

```
java -jar Mars4_5.jar nc 0xffff0000-0xfffffffc lab5_w21_test.asm
```

Sample Input/Outputs

When you're finished, the bitmap will look like this (not including the gray outer border):



You are expected to read through and understand how the provided lab5_w21_test.asm file works. The test file will call the subroutines in your lab5.asm file and print to the console your results as well as the expected results. This is what the output of your completed lab should look like:

```
-------------------------------
Clear_Bitmap Test:
Paints entire bitmap a medium sea green color

(Check the bitmap display tool to see if it worked.)

-------------------------------
Pixel Test:
Draws a cyan and a yellow pixel in the top left and bottom right respectively

Get_pixel($a0 = 0x00400040) should return: 0x003cb371
Your get_pixel($a0 = 0x00400040) returns:  0x003cb371

Get_pixel($a0 = 0x00000000) should return: 0x0000ffff
Your get_pixel($a0 = 0x00010001) returns:  0x0000ffff

Get_pixel($a0 = 0x007F007F) should return: 0x00ffff00
Your get_pixel($a0 = 0x007e007e) returns:  0x00ffff00

-------------------------------
Horizontal Line test:
Draws an orange horizontal line

Get_pixel($a0 = 0x00550010) should return: 0x00ffa500
Your get_pixel($a0 = 0x00550010) returns:  0x00ffa500

Get_pixel($a0 = 0x00000010) should return: 0x00ffa500
Your get_pixel($a0 = 0x00000010) returns:  0x00ffa500

Get_pixel($a0 = 0x007F0010) should return: 0x00ffa500
Your get_pixel($a0 = 0x007F0010) returns:  0x00ffa500

Get_pixel($a0 = 0x00100040) should return: 0x003cb371
Your get_pixel($a0 = 0x00100040) returns:  0x003cb371

-------------------------------
Vertical Line test:
Draws a firebrick colored vertical line

Get_pixel($a0 = 0x00500055) should return: 0x00b22222
Your get_pixel($a0 = 0x00500055) returns:  0x00b22222

Get_pixel($a0 = 0x00500000) should return: 0x00b22222
Your get_pixel($a0 = 0x00500000) returns:  0x00b22222

Get_pixel($a0 = 0x0050007F) should return: 0x00b22222
Your get_pixel($a0 = 0x0050007F) returns:  0x00b22222

Get_pixel($a0 = 0x00400050) should return: 0x003cb371
Your get_pixel($a0 = 0x00400050) returns:  0x003cb371

-------------------------------
Crosshair Test:
Draws an indigo crosshair

Get_pixel($a0 = 0x00300020) should return: 0x003cb371
Your get_pixel($a0 = 0x00300020) returns:  0x003cb371

Get_pixel($a0 = 0x00450020) should return: 0x004b0082
Your get_pixel($a0 = 0x00450020) returns:  0x004b0082

Get_pixel($a0 = 0x00300045) should return: 0x004b0082
```

This output of the tests are available in this hex dump if you wish to compare. You can compare files online using a "diff" utility like Diffchecker or the bash "diff" command.

If your bitmap is correct, you should be able to make an exact copy of the hex dump using

```
java -jar Mars4_5.jar lab5_w21_test.asm 0xffff0000-0xffffffC > my_output.hex
```

For full credit, **your output should match ours exactly**.

Automation

Note that part of our grading script is automated, so **it is imperative that your program's output matches the specification exactly**. Output that deviates from the spec will cause point deduction.

You should not use a label called "main" anywhere in Lab5.asm. If you do, it will fail to work with our test cases and your assignment will not be graded.

Files

You do not need to include lab5_w21_test.asm in your repo, but you may if you like. We will be using our own test script, similar to the one you're given, just with **different xy-coords and colors**.

Lab5.asm

This file contains your pseudocode and assembly code for all of the functions and macros and should be the only file you edit (except perhaps for debugging purposes). Follow the code documentation guidelines here. By itself, this file should not actually do anything but define the functions.

README.txt

This file must be a plain text (.txt) file. It should contain your first and last name (as it appears on Canvas) and your CruzID. Instructions for the README can be found here.

Google Form

You are required to answer questions about the lab in this Google Form. Answers, excluding the ones asking about resources used and collaboration should total at the very least 150 words.

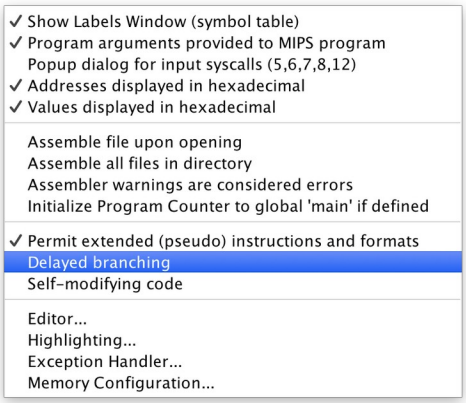
Syscalls

You may use syscalls in the lab5_w21_test.asm file, but you should not use any syscalls in Lab5.asm. We inserted an exit syscall in the template to prevent it from running on its own and you can leave that there, but do not add any more.

Other Requirements

Turn Off Delayed Branching

From the settings menu, **make sure Delayed branching is**



unchecked

Checking this option will insert a "delay slot" which makes the next instruction after a branch execute, no matter the outcome of the branch. To avoid having your program behave in unpredictable ways, make sure Delayed branching is turned off. In addition, add a NOP instruction after each branch instruction. The NOP instruction guarantees that your program will function properly even if you forgot to turn off delayed branching. For example:


```
      LI   $t1 2


LOOP: NOP
      ADDI $t0 $t0 1
      BLT  $t0 $t1 LOOP

      NOP                  # nop added after the branch instruction
      ADD  $t3 $t5 $t6
```

Grading Rubric (100 points total)

12 pt assembles without errors
80 pt outputs (and function signatures) match the specifications
   15 pt getCoordinates, formatCoordinates, getPixelAddress
   10 pt draw_pixel, get_pixel
   25 pt clear bitmap
   20 pt draw draw_horizontal_line, draw_vertical_line
   10 pt draw_crosshair
Note: credit for this section **only** if program assembles without errors

8 pt documentation
   4 pt README file complete
   4 pt Google form complete
-100 pt no Google form submitted or incorrect commit ID