"One who hasn't hacked assembly language as a youth has no heart. One who does so as an adult has no brain." – John Moore

"Real programmers can write assembly code in any language." – Larry Wall

Learning Objectives

This lab involves writing MIPS procedures. Specifically, the concepts involved are:

1. Arithmetic and logical operations in MIPS
2. Arrays and pointers in MIPS
3. MIPS control flow (conditionals, loops, etc.)
4. MIPS function calling conventions

Work that needs to be handed in (via SVN)

First deadline

1. `add_dot.s`: implement the `add_dot` function in MIPS. Run with:
   `QtSpim -file common.s add_dot_main.s add_dot.s lab7_given.s`

Second deadline

1. `get_origin.s`: implement the `get_origin` function in MIPS. Run with:
   `QtSpim -file common.s get_origin_main.s get_origin.s lab7_given.s`
2. `add_line.s`: implement the `add_line` function in MIPS. Run with:
   `QtSpim -file common.s add_line_main.s add_line.s lab7_given.s add_dot.s get_origin.s`
3. `is_connected.s`: implement the `is_connected` function in MIPS. Run with:
   `QtSpim -file common.s is_connected_main.s is_connected.s lab7_given.s get_origin.s`

Important!

Unlike previous labs, this is a solo lab and you may not work with anyone else on this lab.

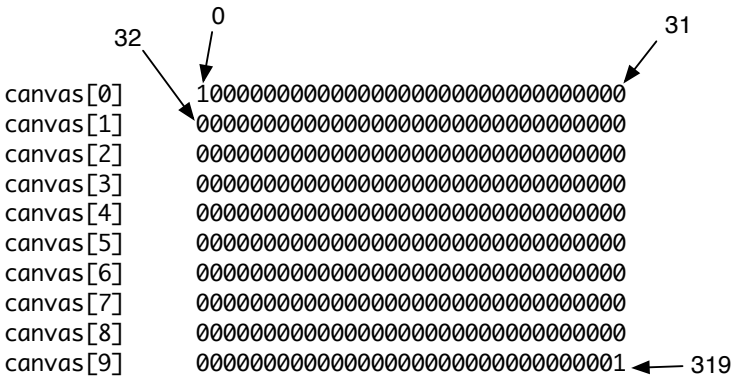
We also cannot stress enough the importance of reading the *entire* lab handout.

For Lab 7, we are providing all "main" files (*e.g.*, `add_dot_main.s`) and files for you to implement your functions (*e.g.*, `add_dot.s`). We will need to load both of these files into QtSpim to test your code.

We will only be grading the files you implement your functions in, and we will do so with our own copy of the main files, so make sure that your code works correctly with original copies of those.

Rules

- You may use any MIPS instructions or pseudo-instructions that you want. A full list of instructions can be found at `http://cs.wheatonma.edu/mgousie/comp220/SPIM_Quick_Reference.html#instructions`
- Follow all function-calling and register-saving conventions from lecture. **If you don't know what these are, please ask someone.** We will test your code thoroughly to verify that you followed calling conventions.
- We will be testing your code using the EWS Linux machines, so we will consider what runs on those machines to be the final word on correctness. Be sure to test your code on those machines.
- **Our test programs will try to break your code.** You are encouraged to create your own test programs to verify the correctness of your code. One good test is to run your procedure multiple times from the same main function.
- We have a set of style guidelines for MIPS (`https://wiki.illinois.edu//wiki/display/cs233sp18/MIPS+style+guidelines`). Assembly code can be close to impossible to understand and debug without sufficient formatting and debugging, and doing so isn't an efficient use of our time in office hours. Therefore, **we will only assist you in office hours if your MIPS meets a certain standard of formatting and commenting**. If it doesn't, we'll ask you to format and comment it before we help you. Specifically, as far as formatting is concerned, the first example on the linked page definitely won't fly, the second is iffy, and the third is really what you should be going for. For commenting, since we're translating C to MIPS, it should be immediately clear from your comments what C statement each MIPS instruction is corresponding to; the commenting scheme on the linked page is one way (but not the only one) to achieve this.



Part1 [20 points]

As a MIPS programmer, you will create artwork on a small canvas by painting one pixel at a time. Your canvas is 10×32 pixels, which is stored in memory as an integer array with 10 integers. Each pixel is a single bit on one of these integer and drawing a dot is as simple as setting a bit. The function below sets a bit based on asingle integer that specifies the position of the pixel to paint from 0 to 319 in row-major order. See the figure above.

Add Dot on Canvas [20 points]

The first task you are working on is adding dot at the given position by setting the corresponding bit to 1.

```
void add_dot(unsigned int pos, unsigned int* canvas) {
    unsigned int row = pos >> 5;
    unsigned int col = 31 - (pos & 31);
    canvas[row] |= (1 << col);
}
```

The C++ code and a test case for `add_dot` function are in the file named `lab7.cpp`. You have to write a MIPS translation of this function in `add_dot.s`. There are some test cases in the assembly file named `add_dot_main.s`, and you can compile and run the C++ code and compare its output to your MIPS code to verify its correctness. Feel free to add more test cases to both the C++ and `add_dot_main.s`.

0	1	2	2	2	2	6	7
0	33	34	35	4	37	38	39
0	65	66	67	4	69	70	71
0	97	98	99	4	101	102	103
0	129	130	131	132	133	134	135

Part 2 [80 points]

In the second part of this lab, you will practice making function calls and writing loops.

We now introduce the notion of strokes. We group pixels into strokes by keeping track of the *origin* of each pixel in an array of integers (where there is one integer per pixel). A pixel doesn't belong to any stroke stores its own index as its origin. All pixels in a given stroke have the same origin, which is the start position of the stroke. In the example image, there are two strokes, one starting at pixel 0 and the other starting at pixel 2.

The stroke starting at pixel 2 is actually composed of two strokes that have been *connected*. The two sub-strokes are the horizontal one starting at pixel 2 and a vertical one starting at pixel 4. These strokes are considered connected, because the origin (location 4) of the vertical sub-stroke does not contain its own value, but instead contains the origin (value 2) of the horizontal line. This saves us from having to update the origins of every pixel in the vertical sub-stroke.

As a result, finding the origin is an interative procedure. Given a pixel, we look up its origin. We keep looking up the origin of that pixel until the pixel is its own origin.

Search for Origins [20 points]

Now given a map of origin values, return the origin at the query position.

```
unsigned int get_origin(unsigned int pos, unsigned int* origins) {
    while (pos != origins[pos]) {
        pos = origins[pos];
    }
    return pos;
}
```

The C++ code and a test case for `get_origin` function are in the file named `lab7.cpp`. You have to write a MIPS translation of this function in `get_origin.s`. There are some test cases in the assembly file named `get_origin_main.s`, and you can compile and run the C++ code and compare its output to your MIPS code to verify its correctness. Feel free to add more test cases to both the C++ and `get_origin_main.s`.

Check Connectivity [20 points]

Now you can check whether two positions are connected by a set of lines or not. Note that parallel adjacent lines are NOT connected unless there is a third line that crosses both of them.

```
bool is_connected(unsigned int pos1, unsigned int pos2,
                  unsigned int* origins) {
    return get_origin(pos1, origins) == get_origin(pos2, origins);
}
```

The C++ code and a test case for `is_connected` function are in the file named `lab7.cpp`. You have to write a MIPS translation of this function in `is_connected.s`. There are some test cases in the assembly file named `is_connected_main.s`, and you can compile and run the C++ code and compare its output to your MIPS code to verify its correctness. Feel free to add more test cases to both the C++ and `is_connected_main.s`.

Note that `is_connected.s` calls your `get_origin` function from `get_origin.s`. You should use `jal` to call this function in your code. **Do NOT inline** `get_origin` . Also, do not include the `get_origin` functions in the `is_connected` file. Our tests will ensure that you're actually calling the function.

Draw Lines [40 points]

The final task is drawing lines on your canvas and updating the origin map.

```
void add_line(unsigned int start_pos, unsigned int end_pos,
              unsigned int* canvas, unsigned int* origins) {
    int step_size = 1;
    // Check if the line is vertical.
    if (!((start_pos ^ end_pos) & 31)) {
        step_size = 32;
    }
    if (start_pos > end_pos) {
        step_size *= -1;
    }
    // Update the origin map.
    add_dot(end_pos, canvas);
    for (int i = start_pos; i != end_pos; i += step_size) {
        add_dot(i, canvas);
        origins[get_origin(i + step_size, origins)] = get_origin(i, origins);
    }
}
```

The C++ code and a test case for `add_line` function are in the file named `lab7.cpp`. You have to write a MIPS translation of this function in `add_line.s`. There are some test cases in the assembly file named `add_line_main.s`, and you can compile and run the C++ code and compare its output to your MIPS code to verify its correctness. Feel free to add more test cases to both the C++ and `add_line_main.s`.

Note that `add_line.s` calls your `add_dot` function from `add_dot.s`. You should use `jal` to call this function in your code. **Do NOT inline** `add_dot` . Also, do not include the `add_dot` functions in the `add_line` file. Our tests will ensure that you're actually calling the function.

We will release our own version of `add_dot` after the first part of the lab is due. We recommend that you copy over our solution into your `add_dot.s` file so that you can test `add_line.s` in the same environment that we will run the autograder.

Calling Conventions

Remember calling conventions; treat the functions that you call (e.g. `add_dot`) as black boxes (*i.e.*, pretend that you don't know how they are implemented), and make sure you're correctly saving and restoring across the function call. We'll test your code against **our own versions** of the called functions to ensure you're following conventions.

Suggestions

Much of the course staff finds this code (in particular `add_line`) easier to write using the (callee-saved) `$s` registers! Using `$s` registers allows you to save registers once at the beginning of the function to free up the `$s` registers; you can then use `$s` registers for all values whose lifetimes cross function calls. We've posted videos in Piazza about using `$s` registers.

There are also some helper functions for printing out values in `common.s` and `lab7_given.s` which you may find helpful for debugging purposes.