He who hasn't hacked assembly language as a youth has no heart. He 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

1. `p1.s`: implement the `fill_array` function in MIPS. **This is due by the first deadline.**

   Run on EWS with: `QtSpim -file p1_main.s p1.s`

2. `p2.s`: implement the `selection_sort` and `draw_gradient` functions in MIPS. **This is due by the second deadline.**

   Run on EWS with: `QtSpim -file p2_main.s p2.s`

Important!

For Lab 6 we are providing "main" files (*e.g.*, `p1_main.s`) and files for you to implement your functions (*e.g.*, `p1.s`). We will need to load both of these files into QtSpim to test your code.

We will only be grading the `p1.s` and `p2.s` files, and we will do so with our own copy of `p*_main.s`, so make sure that your code works correctly with an original copy of `p*_main.s`. We provide you with a `code.c` file that contains a C implementation of all the functions that you need to implement for this Lab.

Guidelines

- You may use any MIPS instructions or pseudo-instructions that you want.
- 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.
- Keep your code clean and follow the style guides on the website. **The course staff can and will refuse to help you if your code is not formatted correctly**.

Problem 1: `fill_array` [20 points]

This function takes an unsigned integer and an integer array as arguments. The function splits the requests into overlapping chunks, masks them, and fills the array with the corresponding values. As per convention request is passed through `$a0` and the array is passed through `$a1`.

```
// Sets the values of the array to the corresponding values in the request
void fill_array(unsigned request, int* array) {
  for (int i = 0; i < 6; ++i) {
    request >>= 3;

    if (i % 3 == 0) {
      array[i] = request & 0x0000001f;
    } else {
      array[i] = request & 0x000000ff;
    }
  }
}
```

The `fill_array` function requires you to implement a loop.

The next two problems involve doubly-nested loops, dealing with structs, and needing to save/restore registers and call other functions.

Problem 2: `selection_sort` [40 points]

For this part of the lab, you will implement a selection sort function. To review, selection sort subdivides the array into sorted (left side) and unsorted(right side) regions. The algorithm continuously selects the "minimum" element from the unsorted section and sets it as the greatest element in the sorted section. This shrinks the unsorted section by 1 until the entire array is sorted.

With this information, we can implement a selection sort as follows:

```
//Performs a selection sort on the data with a comparator
void selection_sort (int* array, int len) {
  for (int i = 0; i < len -1; i++) {
    int min_idx = i;

    for (int j = i+1; j < len; j++) {
      // Do NOT inline compare! You code will not work without calling compare.
      if (compare(array[j], array[min_idx])) {
        min_idx = j;
      }
    }

    if (min_idx != i) {
      int tmp = array[i];
      array[i] = array[min_idx];
      array[min_idx] = tmp;
    }
  }
}
```

For this selection sort, we use a function called `compare` in order to determine the ordering of the numbers. In the files we give you, we define `compare` as follows:

```
//Performs a selection sort on the data with a comparator
int compare (int lhs, int rhs) {
  if (lhs * (lhs - 1) < rhs * (rhs - 1)) {
    return 1;
  }
  return 0;
}
```

We will change `compare` during testing, so **do not inline compare**. Additionally, expect `compare` to fill any register that is not preserved across a function (see bottom left of the front page of the green sheet) with garbage values. Additionally, reading from any value in memory that is not part of `array` will also produce garbage.

Problem 3: `draw_gradient` [40 points]

For this portion of the lab, we will be representing a point as

```
typedef struct {
  char repr;
  int xdir;
  int ydir;
} Gradient;
```

For this problem, we will draw out the gradient direction map of a grid in ASCII. The map itself has been given to us as an address in `$a0` and we will use the values of the xdir and ydir to determine what character best represents the gradient. It returns the number of changed characters in through `$v0`.

The C code is as follows:

```
//Draws points onto the array
int draw_gradient(Gradient map[15][15]) {
  int num_changed = 0;
  for (int i = 0 ; i < 15 ; ++ i) {
    for (int j = 0 ; j < 15 ; ++ j) {
      char orig = map[i][j].repr;

      if (map[i][j].xdir == 0 && map[i][j].ydir == 0) {
        map[i][j].repr = '.';
      }
      if (map[i][j].xdir != 0 && map[i][j].ydir == 0) {
        map[i][j].repr = '_';
      }
      if (map[i][j].xdir == 0 && map[i][j].ydir != 0) {
        map[i][j].repr = '|';
      }
      if (map[i][j].xdir * map[i][j].ydir > 0) {
        map[i][j].repr = '/';
      }
      if (map[i][j].xdir * map[i][j].ydir < 0) {
        map[i][j].repr = '\';
      }

      if (map[i][j].repr != orig) {
        num_changed += 1;
      }
    }
  }
  return num_changed;
}
```

This function is fairly straightforward to implement. However, it can easily become a spaghetti code monster due to how many if statements are contained. I strongly suggest that you write sub-functions so that the each snippet can fit onto a single page. This makes the code cleaner and easier to debug.

There are two things of note for here: loading ascii characters and indexing 2D arrays. Ascii characters are integers, so the following are all equivalent

```
li  $t0, 0x41
li  $t0, 'A'
li  $t0, 65
```

Additionally, note that a 2D array is just a 1D array with fancy indexing. For this class we use row major ordering (See `https://en.wikipedia.org/wiki/Row-_and_column-major_order`). In short, row major ordering means that each row of the matrix is contiguous in memory and is followed by successive rows in the matrix. To load $A[row][col]$, you need to load $A[row * N\_COLS + col]$.

Suggestion

Much of the course staff finds this code easier to write using the (callee-saved) $s registers! $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.