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

Robots should be respectful. from Calvin and Hobbes by Bill Watterson

Learning Objectives

This MP involves writing MIPS procedures and using the SPIMbot simulator. 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
5. Introduction to SPIMbot

Work that needs to be handed in (via SVN)

1. `problem1.s`: implement the `new_angle` function in MIPS.
2. `problem2.s`: implement the `collision` function in MIPS.

Important!

For each Lab 7 part we are providing a main file (*e.g.*, `p1_main.s`) and a file for you to implement your function (*e.g.*, `problem1.s`). You will need to load both of these files into QtSpim to test your code.

We will only be grading the `problem*.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`.

How to run code (SUPER DUPER EXTRA IMPORTANT)

You can run Problem 1 in QtSpim on the EWS machines using the command

```
~cs232/Linux/bin/QtSpim -file p1_main.s problem1.s
```

In addition, you can see a simulation of the snake game with your code by running

```
~cs232/Linux/bin/QtSpimbot -file p1_spimbot.s problem1.s
```

You can run Problem 2 in QtSpim by running

```
~cs232/Linux/bin/QtSpim -file p2_main.s problem2.s
```

Unfortunately we don't have a spimbot file for this one. ☹

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.

Practice Problems

```
if (x != 0) {
  x = x << 5;
}
x |= 1;
```

```
void
init_array_of_length_12(int array[]) {
  for (int i = 0 ; i < 12 ; i ++) {
    array[i] = 0;
  }
}
```

```
void
woot(int x, int y) {
  if (x > y) {
    y = 0;
  } else if (x == 0) {
    return y;
  } else {
    y *= 2;
  }
  return x + y;
}
```

```
void
foo(int A[], int x) {
  int temp = bar(A[x]);
  A[x] += temp;
}
```

Problem 1: Snake motion planning

This semester in CS 398, we will be writing MIPS programs to play the classic "snake" game, where you try to eat apples to increase the length of your snake and your snake needs to avoid running into itself.

The first step of this process is to compute a path to an apple given the (x,y) location of the head of the snake and the (x,y) location of an apple. Since our snake is constrained to travel in one of the 4 cardinal directions – 0°(+x), 90°(+y), 180°(-x), 270°(-y) – our code simply tries to align us with the apple's X position (left/right) and then aligns with the apple's Y position.

```
int
new_angle(int my_x, int my_y, int apple_x, int apple_y) {
  if (my_x != apple_x) {    // are we already aligned in x?
    if (my_x < apple_x) {   // if not, are we too far to left?
      return 0;             // then face to the right
    }
    return 180;             // otherwise face to the left
  } else {                  // otherwise my_x == apple_x
    if (my_y < apple_y) {   // are we above the apple?
      return 90;            // then face down
    }
    return 270;             // otherwise face up
  }
}
```

We've provided a number of test cases for this code in `p1_main.s`, but you are encouraged to add your own. Also we've provided `problem1.c`, which you can compile and run to confirm what the correct output for the test cases should be. After compiling it, run it with:

```
executable_name  <snake_x> <snake_y> <apple_x> <apple_y>
```

For example: (if you named your exeuctable "problem1")

```
problem1 50 100 200 200
```

Also, for fun, we've provided you some code to allow you to use code to control a SPIMbot snake in `p1_spimbot.s`.

Problem 2: Doing collision detection with snake bodies

With the code that you wrote in Problem 1, the snake is oblivious of its own body (and the bodies of other snakes) when it plans its path. If you run the SPIMbot tester code with your solution, you'll see that the snake runs into its own body (with bad consequences). The first step to avoiding running into a snake body is to predict when a collision is going to happen. That is what we'll do in Problem 2.

For this problem, we're again given the current (x,y) location of the snake's head and of an apple. In addition, we're given a data structure that describes a snake's body (our own or someone else's). Since a snake can only move in cardinal directions, a snake's body consists of an alternating series of connected horizontal and vertical line segments (as shown in Figure 1). These segments can be described by the series of points (vertices) that when connected make up the snake. Thus, we represent the snake to our code using two arrays, one for the **x** values of the vertices and one for the **y** values. All six of these values are inputs to our function.

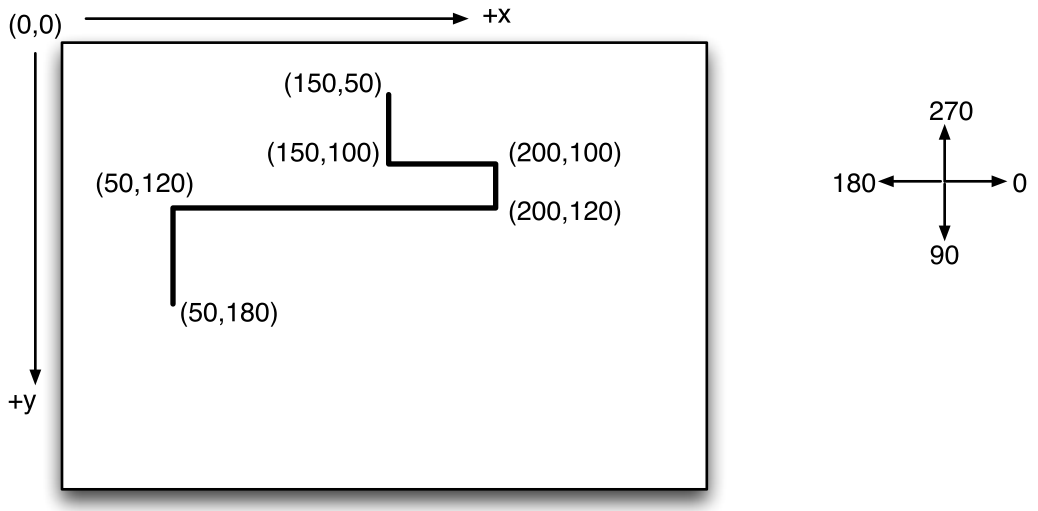

Figure 1. A snake consisting of the points (50,180), (50,120), (200,120), (200,100), (150,100), (150,50).

The first 4 arguments can be passed in registers `$a0-$a3`, but the rest (2 in this case) have to be passed on the stack. The calling convention is that the additional arguments are placed at the bottom of the caller's stack, with the 5th argument placed at `0($sp)` and subsequent arguments above that.[1]

The goal of our collision detection function is to determine whether the snake's body crosses the line between the current location of the snake's head and the target location. Since our snake only moves in cardinal directions, we're going to assume that either `my_x == target_x` or `my_y == target_y`; that is, either we're moving only in the X or only in the Y direction. In fact, the code that you need to implement only handles the case where the proposed movement is up or down.

When moving up or down, we're primarily focused on detecting collisions with horizontal segments. As such, our loop determines whether the first horizontal segment starts with vertex 0 or 1 and, then, proceeds to check for collisions with every other segment there after. A collision is

[1]The actual MIPS calling conventions are slightly different; they require the caller to reserve space for all arguments, even those passed in the `$a` registers, so that the 5th argument would actually be placed at `16($sp)`. We're omitting this detail to simplify things.

detected when one point is on each side of the line segment (`my_loc_above`$\oplus$`target_loc_above`) and if our path is neither to the left or to the right of the snake segment. Our code returns the first segment of the snake that intersects the proposed path, which is not necessarily the one that would be first encountered.

```
int
collision(int my_x, int my_y, int target_x, int target_y, int *snake_x, int *snake_y) {

  int i = 0;

  if (my_x == target_x) {              // going up or down
    if (snake_x[0] == snake_x[1]) {    // first segment is verticle
      i ++;                            // skip first segment
    }
    for ( ; (i < MAX_LENGTH) && (snake_y[i] != -1) ; i += 2) {
      int my_loc_above = (my_y < snake_y[i]);
      int target_loc_above = (target_y < snake_y[i]);
      int left = (my_x < snake_x[i]) && (my_x < snake_x[i+1]);
      int right = (my_x > snake_x[i]) && (my_x > snake_x[i+1]);

      // ^ is the symbol for XOR
      int intersection = (my_loc_above ^ target_loc_above) && !left && !right;
      if (intersection) {
        return i;
      }
    }
  }

  return -1;
}
```

We've provided working C code as `problem2.c`. In fact, this code also has an implementation of the case when the snake's target is to its left or right, but you don't need to implement that part in MIPS, so we commented it out. Like `problem1.c`, `problem2.c` can be used as the definition of correctness to test what should be returned for various inputs.