“Always code as if the (person) who ends up maintaining your code will be a violent psychopath who knows where you live.”  
– M. Golding

This is a solo lab!

You will need to re-download QtSpimbot using the instructions from CS233 On Your Own Machine!

Learning Objectives

1. Introduction to memory-mapped I/O
2. Introduction to interrupt handlers
3. Understanding of SPIMBot simulator

Work that needs to be handed in (via github)

1. part1.s: Pick up corn from around the map.  
   Run on EWS with:  
   QtSpimbot -part1 -file part1.s

2. part2.s: Pick up corn and use the puzzle interrupt.  
   Run on EWS with:  
   QtSpimbot -part2 -file part2.s

Be sure to read the entire handout!

Guidelines

- Same procedure as previous labs on MIPS. Use any MIPS instructions or pseudo-instructions you want. We may try to break your code.
- We will post solutions for Lab 7 so that you can use the correct Dominosa implementation tailored for Lab SPIMBot.
- As always, follow all calling and register-saving conventions you’ve learned.
- **Remember to test your code often!**
- Use the SPIMBot documentation as a reference:  
The Morrow Plots

The Morrow plots security have all caught COVID-19! Now thousands of students are stealing all the corn in the Morrow plots. As a upstanding member of society (and totally not planning on keeping the corn for yourself), you will build a autonomous bot to harvest as much of it as possible and steal it from other students.

Part 1: Corn Pickup [20 points]

For now, we will setup a small simulation for you to get used the robot environment. The first task we will ask you to do is pickup corn around the map. The map is constructed of many different tiles, some off these will be floors and some walls. Scattered across some of these tiles will be ears of corn. **You job in this part is to collect 4 ears of corn.**

The map is split up into discrete cells. The entire kitchen is $40 \times 40$ cells, with each cell being $8 \times 8$ pixels. As a brief introduction, there are two main ways of accomplishing this: you can either plan a path to the corn, set your velocity towards your destination, and loop until you reach your destination. So, this method will use the following Memory Mapped I/O: ANGLE, VELOCITY, BOT_X, BOT_Y, and PICKUP. Alternatively, you can use calculate the time it would take to get to a points and use the timer interrupt to alert you that you reached the point, so you would likely need: ANGLE, VELOCITY, TIMER, and PICKUP. Some common catches, the SPIMBot travels continuously across the entire map without any jumps in its movement. This means after setting your velocity, your bot will continue to move in that direction until you bonk into a wall. To prevent this, set your velocity to zero when you do not intend to move.

![Map Image]

This is what the map will look like.

In order to implement this part of the lab, you need to know how to pick up corn. If you write any non-zero value to the PICKUP address, then the SPIMBot will pick up any corn that your bot is on. The behavior when the value is 0 is undefined.

Some tips that may be useful for debugging:
You might find that giving SPIMBot the `-debug` parameter will be useful for your debugging; it prints out lots of information when SPIMBot interacts with the world.

You can use the `PRINT_INT` memory-mapped I/O address to do print-statement debugging.

You can use the `-drawcycles` argument to change the speed the game runs at. This is useful to slow down the game and see what SPIMBot is doing.
Part 2: Solving Puzzles [80 points]

Continuing from where we left off, we will also be collecting corn in this part of the lab. This constitutes 20 points of the 80 points in this part. The rest of your points come from solving puzzles. We will be using the same dominosa puzzle with some slight modifications to the question struct that you solved in Lab 7 as the puzzle for this lab - full details in the spimbot documentation. We will provide MIPS for the solver that follows the same algorithm implemented in Lab 7. So, all you need to do is integrate the puzzle solver with the rest of your spimbot.

By solving puzzles, you bots get some currency. Although this has no use right now, this will be used in the full spimbot tournament later in order to pay for other MMIO. To request a puzzle to solve, allocate a space in the .data segment capable of holding an entire dominosa_question. Then, write the address of the start of this memory to the PUZZLE_REQUEST memory-mapped I/O address. When a puzzle is ready, the I/O device will raise a puzzle interrupt. Then, you need to read from REQUEST_PUZZLE_ACK to acknowledge the puzzle. When you acknowledge it, the address you wrote to REQUEST_PUZZLE_ACK will be filled with the DFS tree. Make sure to acknowledge REQUEST_PUZZLE_ACK by writing to it. Check for the interrupt to acknowledge REQUEST_PUZZLE_ACK using the REQUEST_PUZZLE_INT_MASK interrupt mask. Solve this puzzle using the rules defined in Lab 7 and write back the address to the solution to the SUBMIT_SOLUTION address.

```mips
const mem_addr REQUEST_PUZZLE_ACK = 0x.....;
const mem_addr SUBMIT_SOLUTION = 0x.....;
const mem_addr REQUEST_PUZZLE = 0x.....;

// Stuff to put into the data segment
int puzzle_received = 0; // puzzle_received: .word 0
dominosa_puzzle_t puzzle; // puzzle: .space SIZE_OF_PUZZLE
solution_t sol; // solution: .space SIZE_OF_PUZZLE

void interrupt_handler() {
    ...  
    // Puzzle interrupt part
    *REQUEST_PUZZLE_ACK = (int)1; // ACK the interrupt
    puzzle_received = 1;
    // End puzzle part
}

/**
 * Solves two puzzles
 */
void puzzle_part() {
    for (int i = 0; i < 2; i++) {
        puzzle_received = 0;
        *REQUEST_PUZZLE = &puzzle;
        while (puzzle_received == 0) {
            // At some point in time, the we will be interrupted
            // and we will receive the puzzle, but we don’t know when
        }
        // Received puzzle!
        puzzle_solve(puzzle); // This will store the solution into sol
        *SUBMIT_SOLUTION = &sol; // Send the solution location
    }
}
```
void collect_flags() {
    // Code from part 1
}

int main() {
    puzzle_part();
    collect_flags();
}

Hints

- Keep in mind that the map layout is fixed, so you can hardcode the path that SPIMBot will take.
- You will probably find it very helpful to write functions that move SPIMBot one cell at a time. For example, you could define four functions move_east, move_west, move_north, move_south. Then, you could easily express your path as a sequence of moves between cells and not have to worry about any tiny details. To implement these functions, you can poll the SPIMBot’s position. The SPIMBot moves 1 pixel every 1,000 cycles at top speed.
- You will have to switch between moving and solving puzzles. You can choose to either solve all puzzles at the beginning before moving, or use timer interrupts in order to solve puzzles while moving.
- LabSpimbob isn’t out yet and we will likely make substantial changes to the game before releasing it.