“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!

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**: Implement a turn right algorithm to explore enough of the maze to earn 100 points.
   **Run on EWS with:**
   QtSpimbot -part1 -file part1.s

2. **part2.s**: Use timer interrupts to navigate a maze with cycles and request puzzles to unlock enough treasures to earn 10 points.
   **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 8 so that you can use the correct DFS implementation for this lab and the correct Rule 1 implementation 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: https://wiki.illinois.edu/wiki/display/cs233fa18/SPIMBot+documentation
Into The Morrow Plots

A recent research program at UIUC designed the ultimate crop: the supercorn. Created by fusing the genes of many exotic fruits and vegetables, the supercorn requires half as many nutrients but grows twice as fast as traditional corn. However, before the researchers could unveil their creation to the world, a grad student carrying a basket of the supercorn tripped, scattering them across the poorly-mapped Morrow Plots.

That’s where the SPIMBots come in. The SPIMBots are undergrads working on the supercorn project. Because the grad student who spilled the corn had to leave for a conference, they were tasked with recovering the corn (and other unrelated treasures) scattered in the depths of the Morrow Plots.

However, there’s a twist: because the research was going so well, the supercorn is indistinguishable from normal corn. They must use a complex algorithm (which suspiciously reduces to solving a 16x16 Sudoku board) to detect if a corn is a supercorn or just a regular one.

So, with the fate of the project (and maybe the opportunity to get their names on a paper) loaded in their caches, the SPIMBots bravely venture into the mysterious Morrow Plots.

![Maze Image]

This is what the complete SPIMBot tournament looks like. Of course, before you can pick up the corn, you need to learn how to navigate the maze.
Part 1: Turn Right [20 points]

Because the Morrow Plots is so overgrown, there are often thick walls of corn that block the movement of the SPIMBot. Together, all of the walls of corn form an maze (or acyclic graph). Because this maze is acyclic, you can explore it using an always-turn-right strategy (or DFS).

Your goal for Part 1 is to implement an algorithm to turn right whenever there is a space open to the right of the SPIMBot.

The maze is split up into discrete cells. The entire maze is $30 \times 30$ cells. Each cell can have a wall on any of its edges, and some cells may contain a treasure. Whenever a SPIMBot visits a cell, that cell’s background color will change to indicate which SPIMBot, red or blue, visited it. Keep in mind that even though the maze is split up into discrete cells, the SPIMBot travels continuously across the entire arena without any jumps in its movement.

This is what the map will look like. For the first part of this lab, you won’t have to worry about the keys. Instead, you should just focus on implementing the turn-right strategy. The score you earn will increase as you get closer to the center of the maze, up to a maximum of 100 points. In addition, for part 1 we will be using the same random seed every time to generate the maze.

There are two basic rules to implement for this part of the lab:

- If SPIMBot stops seeing a wall to the right, then it should turn right. There is a memory-mapped right wall sensor at address `RIGHT_WALL_SENSOR` built into the SPIMBot. Reading from this sensor will return 1 if there is a wall to the right of SPIMBot, and 0 otherwise. **It’s important that you only turn right if the right wall sensor has changed from closed to open.**

- Sometimes, SPIMBot will reach a dead end and have to turn around. You can detect this by handling bonk interrupts; if SPIMBot reaches a dead end and hits a wall, a bonk interrupt will fire and you should do a 180° turn.
Your task will be to reach the center of the maze. The closer your SPIMBot navigates to the center of the maze, the more points you will earn.

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: Pick Up Treasures [80 points]**

Much like DFS, the turn right algorithm will get stuck in an infinite loop if there is a cycle. For the next part, you will be navigating a maze that has been designed to trip up a turn-right algorithm by having many cycles. In order to navigate this maze, you must pre-plan SPIMBot’s path.

In addition, you must also solve puzzles to get enough keys to collect 8 treasures. The puzzles you will be solving are the DFS puzzles from Part 1 of Lab 8. We’ll provide working code for that lab, so the difficult part will be requesting and submitting your solutions for each puzzle.

This is what the map for part 2 will look like. There is another treasure underneath the starting location. Each treasure will be worth 10 points for this lab, so in order to get the full 80 points for this part, your SPIMBot will need to collect them all.

To request a puzzle to solve, allocate a space in the `.data` segment capable of holding an entire tree array (512 bytes). 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 PUZZLE_REQUEST 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. For the sake of this lab, we will be assuming that i and input will be 1 for each generated puzzle. Remember that the output will be an int indicating the depth where input is found. Solve this puzzle using the rules defined in Lab 8 and write back the 4 byte answer to the PUZZLE_SOLVED address.

For each puzzle you solve, you will receive 1 key. In addition to using memory mapped I/O to solve puzzles, you will need it to pick up treasures. Whenever you write a value to the PICKUP_TREASURE address, the SPIMBot will pick up whatever treasure is in its current tile. If there is a treasure, then the SPIMBot will use 1 key to obtain 1 point. If there is no treasure, then nothing will happen but you will still lose a key! To prevent this, you can request a treasure map by using the TREASURE_MAP I/O. This works in a similar way to the puzzle request. You must first allocate space for the treasure_map struct in your data segment, and then write the address of that space to the I/O address. The map struct is laid out as follows:

```c
struct treasure {
    short x;
    short y;
    int points;
};

struct treasure_map {
    unsigned length; // Note: there may not always be 50 treasures!
    struct spin_treasure treasures[50]; // Make sure to read the length field!
};
```

After you write this address, the I/O device will fill it with this struct, which contains treasure locations and point values. The goal of this lab is to use timer interrupts and memory mapped I/O to gain 8 points.

Hints

- Keep in mind that this second maze 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 may use timer interrupts or just keep polling the SPIMBot’s position. SPIMBot moves 1 pixel every 10,000 cycles at top speed.

- You will have to switch between moving, scanning the treasure map, and solving puzzles. The timer interrupt may be helpful with this.

- For the full SPIMBot tournament, you will be navigating a full maze in the style of Part 1, but you will also have access to a maze map. You can use this map in combination with the four functions listed above to efficiently find paths to treasure. For more information on this, look at the SPIMBot documentation page.

- LabSpimbot isn’t out yet and we will likely make substantial changes to the game before releasing it, but many of the features of the full game (like breaking walls) are already in the binary. If you run QtSpimbot without any -part1 or -part2 arguments, you can try them out for yourselves.