"More than the act of testing, the act of designing tests is one of the best bug preventers known. The thinking that must be done to create a useful test can discover and eliminate bugs before they are coded – indeed, test-design thinking can discover and eliminate bugs at every stage in the creation of software, from conception to specification, to design, coding and the rest." – B. Bezier

"The speed of a non-working program is irrelevant." – S. Heller (in "Efficient C/C++ Programming")

- **There is a new version of QtSpimbot in the `_shared` repository. If you are on your own machine, you must do another `git pull` to update the binary.**
- **We cannot update this handout with clarifications or new information. You should consider the online documentation page as the final source of information on SPIMBot.**

Learning Objectives

1. Assembling larger programs from components
2. Creative problem solving

To run this lab: `QtSpimbot -tournament -file SPIMBot.s -mapseed [mapseed]`

Work that needs to be handed in (via github)

Only ONE team member should submit.

1. `spimbot.s`, your SPIMBot tournament entry,
2. `partners.txt`, a list of your and your 1 or 2 contributor's NetIDs,
3. `writeup.txt`, a few paragraphs (in ASCII) that describe your strategy and any interesting optimizations that you implemented. Explain what your team focused on, why you focused on these goals, and at what points these goals shifted.
4. `teamname`, a name under which your SPIMBot will compete. Team names must be 40 characters or less and should be able to be easily pronounced. Inappropriate names are subject to sanitization.

Guidelines

- **You must do this assignment in groups of 2 or 3 people.** Otherwise, we'll have too many programs for the competition.
- You'll want to use `QtSpimbot`. See Lab 8/10 for directions on running QtSpimbot.
- Use any MIPS instructions or pseudo-instructions you want. In fact, anything that runs is fair game (i.e., you are not required to observe calling conventions, but remember the calling conventions are there to help you avoid bugs). Furthermore, you are welcome to exploit any bugs in SPIMBot or knowledge of its algorithms, as long as you let us know after the contest what you did.
- You may only submit one file. All your code must go in that file.
- We will not try to break your code; we will compete it against the other students who might.
- We've provided solution code for Lab 8. You are free to use any of this code in your SPIMBot contest implementation.
- The contest will be run on the EWS Linux machines, so those machines should be considered to be the final word on correctness. Be sure to test your code on those machines.
- Make sure your code is well documented with comments explaining the logic and register usage if you expect course staff to help you. **Course staff can also ask to look at your psuedocode and your debugging attempts so far before deciding to help you.**

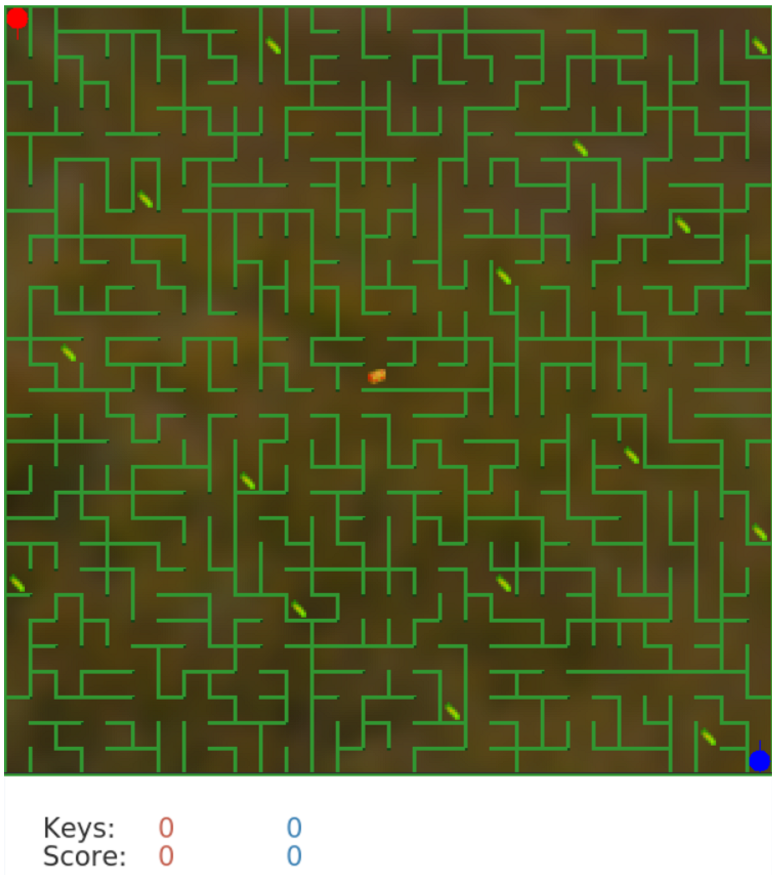
Into The Morrow Plots

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.



Collect the Corn

In this assignment you are to produce a program for SPIMBot that will allow it to earn more points than an opponent. This assignment is based off of the previous SPIMBot labs, but with added components. At the end of the semester, we will have a double elimination tournament to see which SPIMBot program performs the best.

The rules are as follows:

1. In each round of the tournament, two bots will compete in a **randomly generated map**. The winner will be the one finishing with a higher score at the end of 12,500,000 cycles. In the event of a tie, a random bot will win.
2. The map is once again a randomly generated acyclic maze, with the red and blue SPIMBots starting in upper-left and bottom-right corners respectively.
3. Both bots can use the `MAZE_MAP` IO to get a copy of the maze. The maze starts completely obscured (e.g. every cell is walled off). It is gradually revealed as your bot explores, and picking up large treasures will reveal large chunks of the maze immediately. Explored regions are highlighted in the color of your bot.
4. Scoring points can be obtained by collecting supercorn (small treasures) and treasure chests (big treasures). Collecting a supercorn requires 1 key and gives 1 point, while treasure chests require 3 keys and give both 5 points and reveal the map in a 7x7 square around it. The locations of treasures can be obtained by using the `TREASURE_MAP` IO. More information about this IO can be found on the SPIMBot documentation page. Be aware that **treasures are now randomly generated with the map**.
5. For the first 5,000,000 cycles of the game, small and large treasures will respawn when picked up.
6. For the remaining 7,500,000 cycles of the game, small treasures will not respawn but an additional large treasure will appear after the first small treasure is picked up. One can track the progress of the contest by reading the current time from the `TIMER` IO.
7. Keys are obtained by requesting and solving the same Sudoku puzzle in Lab 8. Each puzzle solved will yield 2 keys.
8. SPIMBots can break walls using the `BREAK_WALL` IO. More details on this can be found later in this handout and in the documentation. Be aware that even though the maze starts acyclic, this can be used to add a cycle.

Treasure Hints

The treasure struct is defined as:

```c
struct treasure {
    // The i and j locations are measured in cells not pixels
    short i;
    short j;
    int points;
};

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

To learn the location of treasures, the procedure you use will look something like this:

1. Allocate space for the treasure struct on the data segment. The struct has a fixed size: one `unsigned` length field, and then 50 `treasure` structs in an array, for a total of 404 bytes of space.
2. Write a pointer to this space into the `TREASURE_MAP` IO address.

Wall Breaking Hints

In addition to collecting treasures, a SPIMBot can break down walls in the maze. In order to break a wall, the SPIMBot must write the wall it wishes to break to the `BREAK_WALL` address. The walls are enumerated below:

Code	Wall
0	South
1	West
2	North
3	East

Puzzle Hints

To request a puzzle to solve, allocate a space in the .data segment capable of holding an entire 16×16 sudoku puzzle (512 bytes). Then, write the address of the start of this memory to the `REQUEST_PUZZLE` address. The IO device will fill this space with a 16×16 sudoku puzzle. Solve this puzzle using the rules defined in Lab 8 and write back the address with the solved board to the `SUBMIT_SOLUTION` address. Almost all of the sudoku puzzles can be solved by `rule1` alone, but a few of the puzzles require `rule2`. This I/O device will verify your solution and reward you with 2 keys if it is correct.

You will find that running SPIMBot with the `-debug` flag will be useful for your debugging; it prints out useful interactions that SPIMBot makes with the world. Note: you can use the `PRINT_INT` memory-mapped I/O address to print out relevant values for debugging purposes. **We have noticed that some bots do not perform correctly without the debug flag enabled.** Your SPIMBot will be run without the `-debug` flag during the tournament, so be sure that your code works without it.

For more information, read the documentation at `https://wiki.illinois.edu/wiki/display/cs233fa18/SPIMBot+documentation`.

Scoring

Your SPIMBot will be scored out of 100 points.

Qualifying

The first 60 points will be awarded when you qualify for the tournament. To qualify, your bot needs to score at least 20 points on each of the following map seeds:

```
-mapseed 233
-mapseed 1
-mapseed 8
-mapseed 16
```

This part is doable with a simple implementation. A working solution:

1. Requests and solves sudoku puzzles whenever possible
2. Uses a simple turn right algorithm to explore the maze

Remember: **You've already written a lot of this code in Lab 8 and Lab 10**. Reuse as much as you can!

Competing

The remaining 40 points will be earned during the tournament and will be based on your performance. Note that **bots that do not successfully complete the above section will not qualify for the tournament**. You cannot earn these points unless you meet the above baseline.

Helpful SPIMBot and MIPS Tips

- When naming functions and memory addresses, use long and meaningful names. This prevents you from accidentally reusing a name and also makes debugging easier.
- Minimize the time spent in interrupt handlers. While you are executing your interrupt handler, you can't respond to other interrupts like bonks and timers.
- Refer to the SPIMBot documentation frequently. If there's a clarification we make or a bug that we patch, we can't update this handout, but we will update Piazza and the documentation page.
- While you don't need to exactly follow the calling conventions we've used before, there's a reason why almost every language works in terms of functions and procedures. Organizing your code according to calling conventions makes it far easier to debug and extend.
- Consider writing your program in C/C++, and then hand-compiling it to MIPS. C is infinitely easier to comprehend than assembly, and it's much quicker to debug and error-check C code than MIPS.
- Although the turn right bot you implemented in Lab 10 can explore a reasonably large portion of the maze, implementing a better pathfinding algorithm can significantly improve your score.
- 80% of your code will consume 20% of your time. 20% of your code will take up 80% of your time. Learn to recognize when your code is wasting cycles doing nothing and when optimizing a section code is a waste of your time.
- Donald Knuth said "Premature optimization is the root of all evil." Don't blindly optimize! Other than a few obvious tasks (pathfinding, puzzle solving), it's hard to figure out where your code will spend its time. QtSpim has a profiler built in; use it to see where you code spends its time before trying to optimize it!
- Shaving off a few instructions here and there is unlikely to result in speedups most of the time (unless it's in a frequently-run loop). Don't try to be clever with saving cycles - it makes your code harder to read.
- Learn how to use git branching. Knowing that you have a version of your code that you are happy with leaves you to freely experiment with new additions. Gitkraken (free for students) may be helpful with this.
- Consider using polling loops instead of interrupts. For movement, it is much easier to use `BOT_X` and `BOT_Y` instead of the timer interrupt, and it allows you to write more modular code. However, this is a tradeoff because constantly checking `BOT_X` and `BOT_Y` is more expensive.
- You can add another bot to QtSpimbot by passing the `-file2` argument. For example, if you wanted to run your bot against itself: `QtSpimbot -file SPIMBot.s -file2 SPIMBot2.s -mapseed [mapseed]`