"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")

Learning Objectives

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

Work that needs to be handed in (via SVN)

This lab is due Apr 30th at 8 PM, and only ONE team member should submit. Multiple submissions or incorrectly formatted submissions (potentially including submitting teamnames that require sanitizing) will result in at least a 5 point penalty.

1. `spimbot.s`, your SPIMbot tournament entry,
2. `partners.txt`, a list of you and your 1 or 2 partners' NetIDs,
3. `writeup.txt`, a few paragraphs (in ASCII) that describe your strategy and any interesting optimizations that you implemented, and
4. `teamname.txt`, a name under which your SPIMbot will compete. Team names must be 40 characters or less and should be able to be easily pronounced (must be ASCII). Any team names deemed inappropriate are subject to sanitizing.

Guidelines

- **You must do this assignment in groups of 2 or 3 people.** Teamwork is an essential skill for future courses and in the professional world, so it's good to get some practice. If you do the assignment individually, you won't be entered in the tournament, so you can earn at most 60% of the points for this lab.
- Use any MIPS instructions or pseudo-instructions. In fact, anything that runs is fair game (*i.e.*, you are not required to observe calling conventions, but remember calling conventions will aid debugging). Furthermore, you are welcome to exploit any bugs in SPIMbot or knowledge of its algorithms (the full source is provided in the `_shared/LabSpimbot` directory in SVN), as long as you let us know in your `writeup.txt` what you did.
- All your code must go in `spimbot.s`.
- We will not try to break your code; we will compete it against the other students.
- Solution code for Lab 8 will be provided in the `_shared` directory in svn. You are free to use it in your contest implementation. We will also provide some useful trig functions in the taylor.s file.
- syscalls will be disabled for this lab
- 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.
- Refer to the SPIMbot documentation for details on its interfaces: `https://wiki.cites.illinois.edu/wiki/display/cs233sp18/SPIMbot+documentation`

Problem Statement

In this assignment, you are to design a SPIMbot that will compete with other SPIMbots in the SPIM Galaxy introduced in Lab 10. Spimbots need to collect asteroids, drop asteroids off at moving space stations, and avoid failing into the supernova! The SPIMbot that brings the asteroids worth the most points to the space station wins the game (and earns glory!) Unlike Lab 10, SPIMbot no longer has the magic energy source to power its engine. Luckily, SPIMbot found a good alternative that can supply infinite energy with the small cost of solving some puzzles. Moreover, solving puzzles also gives SPIMbot a secret superpower to prevent the opponent bot from taking away its favorite asteroids.

This handout will provide you with the general game and puzzle settings. For a comprehensive list of memory-mapped I/O addresses, interrupts, and their usage, please refer to the online SPIMbot documentation.

May The Force Be With You![1]

[1] That's not how the Force works!

The Game

Objective

In each round of the tournament, we will compete two robots to see which one can bring asteroids worth more points back to its space station. Both bots will start at the same x location with the same limited amount of energy. The two space stations will be on canvas for roughly the same number of cycles. You are required to write a SPIMbot that will participate in this tournament by collecting asteroids and dropping asteroids off at the space station.

Collecting Asteroids and Dropping Off at the Space Station

For asteroids, space stations, and gravity, we follow the same settings as in **Lab 10 Part 2**. Refer to the Lab 10 handout for more details.

Note that during the competition, it is possible that the asteroids your bot aims at will be collected by the opponent bot first. You might want to update your asteroids map accordingly to locate new targets.

The Superpower

The secret Superpower is revealed in the SPIMbot Interactions section.

Energy Consumption

Unlike Lab 10, maintaining a non-zero engine velocity will cost energy. The energy cost per cycle is proportional to the magnitude of SPIMbot's engine velocity. SPIMbot will start with a limited amount of energy that is insufficient for the entire game. You can gain more energy through solving puzzles, as described in the Puzzle section.

The Puzzle

Objective

The role of the puzzles in this competition is to acquire energy and activate the secret superpower. You will solve the count disjoint regions problem from Lab 8.

Puzzle Description

Recall from Lab 8, the puzzle consists of a canvas and a set of lines. You need to compute the number of disjoint regions after drawing each line on the canvas. Refer to the Lab 8 handout for the details of the `Canvas`, `Lines`, and `Solution` structures.

Puzzles will come in different sizes. The maximum canvas size will be 12×12 and the maximum number of lines will be 12. This information is useful for computing the size of puzzle and solution structures.

Requesting and Submitting Puzzles

In order to request a puzzle, the same memory-mapped I/O scheme that you used to request the asteroid map in Lab 10 will be used. You should statically allocate a region of memory in your `.data` section for your puzzle. Make sure you allocate enough space for the Puzzle struct. As with `ASTEROID_MAP`, you should store the memory region's address into the `REQUEST_PUZZLE` memory-mapped I/O address. Below is an example of how to request a puzzle:

```
la  $t0, puzzle_data
sw  $t0, REQUEST_PUZZLE
```

However, as opposed to the case of requesting the asteroid map, your bot will not receive the puzzle instantaneously; instead, a request puzzle interrupt will fire when the puzzle is ready. The wait time for the interrupt is random. The interrupt will be delivered immediately after the puzzle is done copying into the memory address you provided. After receiving the interrupt, you will need to acknowledge the interrupt and solve the puzzle. The puzzles can be solved using much of the code that you wrote in Labs 8.

The format of the puzzle written into the memory is equivalent to the following `Puzzle` struct:

```
struct Puzzle{
    // Canvas
    Canvas canvas;

    // Lines (at offset 16)
    Lines lines;

    // The rest of the struct stores the actual data for canvas and lines.
    // You should not need to know the exact format of this field.
    // You should access data following the pointers in canvas and coords.
    unsigned char[300] data;
}
```

You should also allocate static memory for a Solution struct in the `.data` section. After solving the puzzle and filling out a Solution struct with your solution, your bot should submit the solution. A memory-mapped I/O address, `SUBMIT_SOLUTION`, has been provided. Store the address of your Solution struct into the `SUBMIT_SOLUTION`.

Since your bot will be solving multiple puzzles, you need to zero out the solution struct after submitting a puzzle solution. That is, every time you want to solve a new puzzle, you should start out with a zeroed out Solution struct.

Optimization

The solver in Lab 8 requires iterating through the entire canvas for each line, which can be costly for canvas with a large number of lines. We encourage you to optimize the puzzle solver by designing an algorithm that does not require iterating through the entire canvas in every iteration. Specifically, you might find the data structure presented in Lab 7, Disjoint Sets, useful for keeping track of the disjoint regions on the canvas.

SPIMbot Interactions

After solving too many puzzles and getting an energy surplus, SPIMbot discovers a secret way to use puzzles: Puzzles can be thrown to the opponent to freeze the opponent bot. Once a puzzle is thrown, the engine velocity of the opponent bot will be set to zero until it submits the puzzle solution.

Freeze the Opponent Bot

To throw a puzzle at your opponent (instead of gaining energy from it), you need to write to the `THROW_PUZZLE` memory-mapped I/O address anytime before writing to `SUBMIT_SOLUTION`. Writing to `THROW_PUZZLE` **only** activates puzzle throwing for the **next** submission. If the **correct** solution to the current puzzle is submitted, the opponent bot will be frozen. Submitting an incorrect solution will not freeze the opponent bot. The puzzle received by the opponent bot is **unrelated** to the puzzle that you just solved. Instead, the puzzle received by the opponent bot will have the maximum possible canvas size and number of lines.

You only have limited chances to use this superpower. In particular, once the `THROW_PUZZLE` option is successfully set for the next puzzle submission, the number of chances left will be decremented by one regardless whether the next puzzle is correctly solved or not. Once all puzzle throws have been used, all future puzzle submissions can only be used for getting energy.

You cannot activate puzzle throwing while the opponent bot has already been frozen. You can check whether the opponent bot is frozen or not by reading from the memory-mapped I/O address `CHECK_OTHER_FROZEN`. Activating puzzle throwing while the opponent bot is frozen has no effect: neither the opponent bot will be frozen at your next puzzle submission nor you will lose a throw.

Unfreeze Your Own Bot

Not surprisingly, your opponent can freeze your bot as well. You will be notified that your bot is frozen through the `BOT_FREEZE` interrupt. Along with the interrupt, you will receive a puzzle to solve in order to unfreeze your bot. To get the puzzle, you need to acknowledge the interrupt at `BOT_FREEZE_ACK` with **a valid address** for a memory region in which to store the puzzle. You can use the same memory allocation procedure as the one for `REQUEST_PUZZLE`. We recommend that you allocate a separate memory for the thrown puzzle, instead of using the memory allocated for the puzzles you request. The memory address will be populated instantaneously upon the acknowledgment. The puzzle will have the same structure as the ones you get through `REQUEST_PUZZLE`, but with the maximum possible canvas size and number of lines. You can solve the puzzle with the same puzzle solver.

It is in your best interest to solve the puzzle as soon as possible to regain the control of SPIMbot. Once the puzzle is solved, the solution should be submitted to the memory-mapped I/O address `UNFREEZE_BOT`, following the same procedure as submitting to `SUBMIT_SOLUTION`. The engine velocity of your bot will be restored immediately after writing to `UNFREEZE_BOT`, but there will be a significant energy penalty for turning in wrong solutions.

Testing and Debugging

Testing

Test your bot with the `-randommap` argument, which will start the game with different asteroids and puzzles at each run. You can use the following command to run the program:

```
QtSpimbot -f spimbot.s -randommap
```

You should also run your bot with an opponent bot to check if your bot handle interaction related interrupts and memory-mapped I/Os correctly. A proper implementation of bot interaction is critical for the tournament, as your bot will be run against random selected bots. The simplest way to do this is run your bot against itself, assuming you have implemented bot interactions. The following command runs two bots under the tournament settings:

```
QtSpimbot -f spimbot.s -f2 spimbot.s -randommap -tournament
```

We encourage you to implement multiple bots with different strategies and test them against each other for the best performance.

Debugging

Before digging into your code, make sure it follows the MIPS style guide. Assembly code with insufficient formatting can be impossible to read and debug.

If your program raises exception while running, you can set break points and follow the Debugging Notes for MIPS labs.

You can also run your program with the `-debug` argument as follow:

```
QtSpimbot -f spimbot.s -debug
```

Running the program with `-debug` argument will print out memory-mapped I/Os and interrupts that are triggered and received by the bot, as well as the information related to asteroids and space stations.

By default, the debugging information will print to the terminal, which might be hard to save and analyze. You can redirect the program output into a file by running the following command:

```
QtSpimbot -f spimbot.s -debug > run.log
```

This command will create a file called `run.log`, which will contain all the debugging information printed out by your program. You can now search through key words in the log. For example, if your bot failed to solve any puzzle, you can search for `puzzle`, `REQUEST_PUZZLE`, and `solution` in the log file. The matched debugging prints will provide you information on whether the bot correctly requests the puzzle, acknowledges the interrupt, and submits the solution. In general, search for the names of memory-mapped I/Os and interrupts in the log could provide you useful information.

Before posting your buggy bot on Piazza or asking course staffs to debug in office hours, please try to debug with the two methods described above. We will ask for the evidence of your debugging attempts before assisting you.

Grading

This lab is graded in two parts, 60% for a baseline bot, and 40% based on how the bot fairs in the final tournament. A basic 60% implementation should:

- Earn 60 points when run by itself.
- Solve at least 1 puzzle.

Good Luck!