“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 December 6th at 8 PM, and only ONE team member should submit.

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. Any team names deemed inappropriate are subject to sanitization.

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 7, and 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/cs233fa16/SPIMbot+documentation
Problem Statement

In this assignment, you are to design a SPIMbot that will compete with other SPIMbots in the strange land introduced in Lab10. After completing Lab10, SPIMbot decided to settle in this land, believing it could survive with its newfound agricultural skills. However, as soon as SPIMbot settled in, the distantly related bots showed their true colors. It had become increasingly clear to SPIMbot’s cousins that it would be impossible for both species of SPIMbot to live in this land, with limited land and resources. SPIMbot must compete against other SPIMbot species, in a fight for survival. Your bot must harvest as much food for itself as possible, and may even sabotage the enemy bot. Furthermore, resources are extremely limited, and SPIMbot will only be able to retrieve resources from the mysterious crop and agricultural buildings that are in this strange land. These buildings are locked, and SPIMbot can only unlock the resources in these buildings by completing KenKen challenges.

On December 7th, SPIMbot and its enemy species will battle in a fight to the finish, once and for all, in a double elimination tournament held in class. The program that performs the best will emerge as the sole surviving SPIMbot species and will live a prosperous life in the strange desert land!

This handout will provide you with details on both the Game and the Puzzle parts of the tournament. May The Force Be With You!

The Game

Objective

In each round of the tournament, we will compete two robots to see which can harvest the most food. Both bots will be placed at the same random starting location, and both bots will start with limited resources. You are thus required to write a SPIMbot that will participate in this tournament by planting, growing, harvesting, and (optionally) burning plants. As in Lab 10, you will be using the TILE_SCAN memory mapped I/O to request the locations of the plants. Refer to the Lab 10 handout or the online SPIMbot documentation for more details.

Planting Seeds

When a bot is positioned over an empty tile and it has at least 1 seed, it can plant a seed at the empty tile. This converts the tile into a growing tile. The growing tile will start with 1 unit of growth (growing tiles will have non-zero, positive value of growth at all times) and 0 units of water. The bot will lose 1 seed. Seeds cannot be planted on a non-empty tile. Attempting to plant a seed on an incorrect tile type will result in losing the seed.

To plant a seed, you must have at least one seed resource. You may request seeds by setting the resource type to 1, and requesting a puzzle. After solving the puzzle, you will be awarded your requested resource (in this case, seeds). See the Puzzle section of this handout for more information about requesting resources/puzzles, and submitting puzzle solutions.

Use the following to plant a seed, and refer to the online SPIMbot documentation for further details:

```
sw  $0, SEED_TILE
```

Plant Growth

Crops will slowly grow on their own without any interaction from the bot (the growth rate is guaranteed to be positive and non-zero). The owning bot can water the crops to encourage faster growth.

---

1 That’s not how the Force works!
Watering Mechanism

Recall from Lab10 that a bot can water plants to increase their growth rate. When a bot is positioned over a growing tile, it can water the tile which will increase the amount of water stored in the tile and neighboring tiles. The bot can decide exactly how much water to dump at the current tile, but no more water than it has stored. Attempting to dump more water will only dump all the water the bot has. Attempting to water an opponent’s tile will also work, but is not a helpful thing to do (obviously). Attempting to water an empty tile will result in losing the water.

To water a tile, you must have at least one unit of the water resource. You may request water by setting the resource type to 0, and requesting a puzzle. After solving the puzzle, you will be awarded your requested resource (in this case, water). See the Puzzle section of this handout for more information about requesting resources/puzzles, and submitting puzzle solutions.

Use the following to water a tile, and refer to the online SPIMbot documentation for further details:

```assembly
li $t0, 10 # Dump 10 units of water
sw $t0, WATER_TILE
```

Water Spread

The tile the bot is over will receive exactly the amount of water dumped by the bot. Any neighboring tiles (excluding diagonals) will also receive some amount of water. The exact amount will be some fraction of the amount of water dumped by the bot. Note that this is not taken away from the original tile, but instead is “magically” generated water that happens to be proportional to the amount dumped by the bot. This effect affects neighboring growing tiles belonging to either bot.

Growth Rate

The growth rate is directly related to the amount of water the tile currently holds. The more water the tile currently has, the faster the growth rate will be for the current tick of time. The tile cannot grow beyond its maximum amount of growth, so the growth rate will effectively be 0 when the tile is fully grown.

Growth rate and the amount of growth over time will be modeled as follows:

\[
G_{\text{rate}} = k \cdot \max(w, G_{\text{min\_rate}}) \\
G_{\text{amount}}(t) = G_{\text{amount}}(t-1) + \min(G_{\text{rate}}, G_{\text{max\_amount}})
\]

where \( k \) is some constant, \( w \) is the amount of water in the tile, \( G_{\text{min\_rate}} \) is the minimum growth rate (which is 1), and \( G_{\text{max\_amount}} \) is some constant that represents the maximum amount of growth.

Maxed Out Growth

We have added a new interrupt, MAX_GROWTH that will fire when a tile has maxed out in growth. Once this interrupt has fired, you can use the MAX_GROWTH_TILE memory mapped I/O to retrieve the location of the tile that maxed out in growth.

Harvesting Tiles

Much of the mechanics regarding harvesting a tile is the same as Lab10.1. When a bot is positioned over a growing tile that it owns, it can harvest the tile to get points. The tile will revert to being an empty tile. Attempting to harvest an opponent bot’s tile or a non-growing tile will simply fail\(^2\). The points gained is exactly the amount of growth in the tile.

\(^2\)Recall from Lab10 that the enemy bot consumes different food that your bot. Why harvest plants that your bot cannot consume?
Use the following to harvest a tile, and refer to the online SPIMbot documentation for further details:

```assembly
sw $0, HARVEST_TILE
```

**Fire!**

**Sabotage**

SPIMbot has been tasked with harvesting its own plants, and SPIMbot’s goal is to harvest more food for itself than the enemy bot. This means that it is advantageous for SPIMbot to stop the enemy bot from successfully harvesting more food.

To sabotage the enemy bot, SPIMbot can set the enemy bot’s plants on fire. To set a tile on fire, you must have at least one fire starter resource. You may request fire starters by setting the resource type to 2, and requesting a puzzle. After solving the puzzle, you will be awarded your requested resource (in this case, fire starters). See the Puzzle section of this handout for more information about requesting resources/puzzles, and submitting puzzle solutions.

Then, SPIMbot can use a fire starter by using the `BURN_TILE` memory-mapped I/O over the tile it wants to burn. This changes the tile’s state to a burning tile and triggers an interrupt for the opponent bot. The bot must have at least 1 fire starter to burn a tile and this will consume 1 fire starter. Attempting to burn a non-growing tile will waste 1 fire starter and achieve nothing.

Use the following to burn a tile, and refer to the online SPIMbot documentation for further details:

```assembly
sw $0, BURN_TILE
```

Your enemy’s bot may decide to burn your plants by using fire starters. To detect this treachery, SPIMbot can use the `ON_FIRE` interrupt. This interrupt will fire when any tile starts burning, unless you are the one that set the tile on fire. To counter this sabotage, you can get the fire location by using the `GET_FIRE_LOC` memory mapped I/O, and then use the `PUT_OUT_FIRE` memory mapped I/O address to dump the exact amount of water needed to put out the fire at the burning tile. See Lab 10 or the online SPIMbot documentation at the link from the beginning of this handout for more details.

**Burning Mechanics**

The burn rate is directly related to the growth amount of the tile. So if a plant is closer to fully grown, it will burn down faster as well. However, this rate is scaled such that all tiles will burn to nothing in the same amount of time, regardless of the current growth amount. Being too far away from a burning tile will likely result in losing the tile entirely. Therefore, if your bot responds quickly, you increase your chances of saving a burning tile if you act fast!

If a bot successfully saves a burning tile, its growth amount will be reduced by the burn amount at the time the tile was saved. In other words, quickly saving a burning tile will reduce the amount of growth lost.

**Spreading Fire**

Since Desert World is incredibly dry, it is likely for fires to spread to neighboring growing tiles. The fire can spread up, down, left, or right, but not diagonally. A burning tile will randomly decide (1) if it is dry enough to spread and then (2) pick a direction to spread to. If there exists no neighboring tile in the chosen direction, or if the tile is not growing, then the fire will not spread.

---

3. Recall from the previous section that you cannot simply harvest and steal the enemy bot plants. This is why you must use fire!

4. Again, pun intended.
Note that fire will spread to any growing tile regardless of which bot owns it. Use fire starters with great caution, there is a risk that it could destroy your own crop by spreading to your own growing tile.

The Puzzles

Objective
The role of the puzzles in this competition is to acquire useful resources from the mysteriously locked buildings in desert land. As discussed earlier, your bot will need to unlock these buildings by completing KenKen puzzles. Upon successful completion of a puzzle, the bot will be granted with the resource in the building.

Puzzle Description
The puzzle, as you may recall from labs 7 and 8, is a KenKen puzzle. Not much has changed from Labs 7 and 8 regarding the puzzle representation, but here are some important points that you should know regarding the puzzle:

- Since your bot will be solving multiple puzzles, you need to zero out the solution struct after submitting a puzzle solution, i.e. Every time you want to solve a new puzzle, you should start out with a zeroed out solution struct.
- Make sure you allocate enough space for your Tile array, Puzzle struct, and Solution struct in case of overflow:
  - Tile Array: 1,600 bytes
  - Puzzle: 4,096 bytes
  - Solution: 328 bytes
- There was a subtle bug in our released lab7.cpp and lab8.cpp files as well as our solutions. It shouldn’t affect your Lab7, Lab8, or this competition. However, if you wish to have the fix, you can find updated files in the _shared directory in SVN.

We will next discuss the process of requesting and submitting puzzles. You will have find the best way to solve the puzzles to acquire resources quickly and beat your opponent.

Requesting and Submitting Puzzles
In order to request a puzzle, the same memory mapped I/O scheme that you used to request the tile array in Lab 10 will be used. You will need to allocate one static memory location in your .data section for your puzzle. Before you request a puzzle, you will first need to use the SET_RESOURCE_TYPE memory mapped address to indicate the resource type that you are requesting (0 for water, 1 for seeds, 2 for fire starters).

In order to request a puzzle, a memory mapped address called REQUEST_PUZZLE is provided. As with TILE_SCAN, you should store the address of the memory allocated for the puzzle into the REQUEST_PUZZLE memory mapped I/O address. Below is an example of how to request a resource, see the online SPIMbot documentation for more information about requesting puzzles.

```
li $t0, 0  # 0 for water, 1 for seeds, 2 for fire starters
sw $t0, SET_Resource_Type
la $t0, puzzle_data
sw $t0, REQUEST_PUZZLE
```

However, as opposed to the case of requesting the tile array, your bot will not receive the puzzle instantaneously;

---

5 With great power comes great responsibility
instead, an interrupt will fire when the puzzle is ready. The request puzzle interrupt mask and acknowledge address are 0x800 and 0xffffffff00d8 respectively. You should use the knowledge you gained in this course about interrupts to handle this interrupt. When you receive the interrupt, the puzzle would have been written into the memory address you provided. The format of the written puzzle would be the following:

- 32 bit integer, size, representing the dimension of the puzzle (same as width, same as height); followed by
- 1 Cell pointer, pointing at an array of size × size Cells that make up the Puzzle; followed by
- The Cage elements that also make up the Puzzle.
- The size × size Cell elements that make up the puzzle (which contain pointers to Cage elements); followed by

Overall, the Puzzle (including the Cell and Cage elements that make up the Puzzle struct) should not exceed 4,096 bytes, so make sure you allocate 4,096 bytes of space for the Puzzle to be copied into.

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 using much of the code that you wrote in Labs 7 and 8. After solving the puzzle and filling out your Solution struct with your solution, your bot should submit the solution. In order to do so, a third memory mapped address has been provided, SUBMIT_SOLUTION. You should therefore provide the address of your Solution struct by storing it’s memory address in the provided memory mapped I/O.

Note: For a list of all the new, and old, memory mapped I/O addresses, refer to the SPIMbot documentation on the wiki page.

**Strategy**

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 5,000 points when run by itself.
- Solve at least 1 puzzle.

**Winning**

In order to win the competition you should get a higher score than your opponent. This can be achieved, not just by harvesting your plants, but also by destroying your enemy’s crop.

There are many ways to optimize your bot to beat your opponent. Here are a few things you may want to consider examining and optimizing if you want to create a highly competitive bot:

- Solving many puzzles quickly, or solving them while your bot is moving to a new location, will allow you to carry more resources, which can save on travel time.
- Being smart about seed placement. When it comes to watering plants, it may be helpful if you are growing plants near each other (because of water spread). On the other hand, this placement strategy would be problematic if someone sets a fire in the middle of your crop. Keep water and fire spread in mind when you are choosing seed placement and when you are choosing what tiles to burn.

**Good Luck!**