As soon as we started programming, we found out to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.

M. Wilkes

Learning is never done without errors and defeat.

V. Lenin

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. Optimizing programs for performance
3. Concurrency (cooperative multithreading)
4. Creative problem solving (optimizations)

Guidelines

- **You must work in groups of 2 or 3 people.** Individual entries will not be permitted in the competition. (Otherwise, we'll have too many entries for the competition.)
- **Refer to the SPIMbot documentation linked from the CS232 assignments web page at:**

  `http://www.cs.uiuc.edu/class/sp12/cs232/assignments/spimbot.html`

- You may use any MIPS instructions or pseudo-instructions you want. As long as the program runs and accomplishes the tasks, you can get credit. This means you don't *have* to follow calling conventions; however, it might be a good idea to do so to avoid bugs.
- This assignment makes use of pieces from MP2 to MP4, so feel free to refer to the provided solutions. In fact, you can just copy and paste some pieces of code from the solutions as a starting point.
- The SPIMbot code for this assignment (*i.e.*, the C program that simulates the SPIMbots) is available on the website for you to get a better understanding of how things work under the hood. Feel free to exploit any bugs that you might find as long as you make a note of it in the writeup file. Alternatively, you can point out non-trivial bugs to us to prevent other people from exploiting them.

- We will be running your code using the `spimbot` executable on **EWS Linux machines**, so we will consider what runs on those machines to be the final word on correctness. Be sure to test your code with that executable on those machines. Shortly we hope to post a version of QtSPIMbot for those who prefer that tool.
- The SPIMbot tournament will be one day after the submission deadline. Late submissions will be accepted up to 2 days later, but they will not be part of the tournament.

Problem Statement

Like in MP4, you will be writing MIPS assembly code to make SPIMbot earn points by scoring goals. To make things interesting, your SPIMbot will be competing against another SPIMbot that is trying to score goals itself.

The playing field is a 300x300 area. Initially, there will be a single soccer ball in the middle of the field. Each time a ball is scored, a new ball is placed randomly on the field. Every quarter of the game that passes, an additional ball is placed on the field, such that during the final quarter of the game there are 4 balls on the field at all times.

Kicking balls is performed in the manner described in MP4. In this MP, however, a SPIMbot kicks harder the farther away it is from its own goal. That is, each energy spent results in more momentum applied to the ball the larger the X component of SPIMbot's position from its goal. Specifically, the scaling factor goes as `(150 + abs(spimbot_x - goal_x))`, so that (compared to your power in the middle of the field) you are half as strong in front of your own goal and 50% stronger in front of your opponent's goal.

The game runs for a fixed amount of time (10,000,000 instruction executions) and whichever robot has scored the most goals wins. In the case of a tie, the robot that scored last wins.

Your robot will be initially provided 100 energy with which to start the contest and it will slowly generate energy during the contest. The main way to generate energy, however, is by a mechanism that Dr. Smartypantz invented to convert the solution of Sudoku puzzles into energy.

The puzzles that the device generates have a distribution of difficulty that is roughly as follows:

1. 50% of the puzzles can be solved by just rule 1 (level 0);
2. 40% of the puzzles can be solved by rule 1 and rule 2, together (level 1);
3. 10% of the puzzles need a brute force solver (levels 2, 3, 4).

Solving the puzzles gives energy 100 energy plus 100 energy times the level of the puzzle.

Here is example pseudocode that explains the process. You are encouraged to get this working in code that doesn't drive SPIMbot first (i.e., get the pieces working separately first, then integrate them).

```
int board[9][9];
int sync = 0;

main() {
  enable_sudoku_ready_interrupts();

  while (1) {
    sync = 0;
```

```
    request_sudoku_board(board);  // this is a memory-mapped I/O command

    while (sync == 0) {}

    solve_board(board);
    sudoku_done(board);  // this is a memory-mapped I/O command
  }
}
```

Use your code (or our solution) from MP 3 for the `solve_board` function.

In addition, implement an interrupt handler for `sudoku_ready` interrupts that: 1) sets `sync` to a non-zero value, and 2) acknowledges the interrupt.

Note that the variable `sync` is used to communicate between the code in `main` and the interrupt handler. This kind of code can be tricky to get right because the interrupt handler could be executed between any pair of instructions in `main`. We (purposefully) are using a very simple protocol, where `main` never writes the value when there could be an interrupt. `main` always sets the value to 0 **before** making the call to `request_sudoku_board`.

What you need to do

This assignment is worth 150 points and there are two components of your grade:

100 points of your grade comes from how your submission performs on its own. When run by itself, you will get 10 points for each goal your code scores. *(Note: if you submit the MP4 solution you can get 40 points...)* Performing better than this will likely require you to integrate sudoku solving code with the driving code to have enough energy to influence the ball positions.

The remaining 50 points will be entirely based on your performance in the SPIMbot contest, as an encouragement for you to optimize your SPIMbot's performance. The top 4 teams will receive the full 50 points and be listed in the SPIMbot Hall of Fame. For the other teams, points will be prorated based on the point your team was eliminated from the competition. All competing teams will earn at least 10 points.

Late submissions are allowed, but will not be entered into the SPIMbot competition and thus will not be able to earn the 50 points based on performance. Submissions from students working alone will have their scores divided by 2 (*i.e.*, **find a partner!**).

For this assignment, you will submit four files:

- `spimbot.s`, your SPIMbot tournament entry

  `partners.txt`, a list of the 2 or 3 contributors' netids
- `writeup.txt`, a few paragraphs (plain text) that describe your strategy and any interesting optimizations that you implemented
- `teamname.txt`, a file containing the name under which your SPIMbot will compete. Team names must be 40 characters or fewer and should be easily pronounced. Any team names deemed inappropriate are subject to sanitization.

Strategy, Optimization, and Tips

While building an entry for the contest is relatively straightforward (see "What you need to do" for the minimum submission), the goal of this exercise is to explore program optimization. Building a

competitive SPIMbot program will require some analysis on your part to determine what are likely to be the most important factors in the contest and, hence, where to spend your time. Here are suggestions of possible factors to help get you started (this list is by no means exhaustive):

1. Should you play offense or defense? -or- (perhaps more precisely) when should you play one or the other?
2. The code from MP#3 solves only some of the sudoku puzzles; should it be extended?
3. Can you overlap driving somewhere with solving sudoku puzzles? (see: "Eliminate Busy Waiting" below.
4. Can you pipeline the process of requesting and solving sudoku puzzles? (see: "Pipelining" below.
5. Where are you spending your time in the computation? What can be made to run faster?

That said, it is important to not over do the complexity in your first implementation. Specifically:

1. Don't Prematurely Optimize

Build a simple implementation first and then optimize it; don't go straight to implementing a super complex uberstrategy. Two pragmatic reasons: 1) if you never get your uberbot working you'll have nothing to hand in, and 2) you will learn a lot in building even a relatively simple implementation that will help you correctly plan how the uberbot should work. This second reason is seconded by Fred Brooks, author of the all-time classic software development book 'The Mythical Man Month':

> "Where a new system concept or new technology is used, one has to build a system to throw away, for even the best planning is not so omniscient as to get it right the first time. Hence plan to throw one away; you will, anyhow."

2. Eliminate Busy Waiting

While it takes a lot of computation to plan a path or to solve the sudoku problems, it takes almost no computation to move the SPIMbot. Perhaps the biggest bang for the buck is to overlap computation with movement. For example, while you are driving somewhere solve sudoku puzzles, or compute what direction you will need to go to the next ball. Timer interrupts can be useful here.

3. Pipelining

Because you will have to wait for the puzzles to be available, it may be advisable to consider pipelining the process of solving puzzles. That is you can have multiple requests for sudoku puzzles outstanding (be sure to allocate them each a different place in memory). This means that you can overlap the solution of one sudoku puzzle with the request for another one (or more), so you are never waiting (after the initial pipeline fill time) for puzzles to arrive. Note that you do have to keep track of which request each "puzzle ready" interrupt relates to and that the solutions are expected back in a first-in, first-out order.

4. Some useful interfaces

Recall that SPIMbot's own X and Y coordinates can be read from addresses `0xffff0020` and `0xffff0024`, respectively. The X and Y position of the opposing SPIMbot can be read from addresses `0xffff00a0` and `0xffff00a4`, this may be useful for playing defense. Also, both robot's current scores can be read from memory-mapped locations, which can let you know whether you are currently winning and by how much.

There are a couple useful options available from the SPIMbot tool. The `spimbot` executable prints out some diagnostic feedback if you include the `-debug` flag.

```
spimbot -file spimbot.s -debug
```

The `spimbot` executable will also collect a complete profile of the execution so you can see where your program is spending its time (we'll talk about this more later in the semester). Use the `-prof_file <filename>` flag as follows to get a profile in a file called "my_profile". **Be sure to quit SPIMbot using the "quit" button to get this to work.**

```
spimbot -file spimbot.s -prof_file my_profile
```

You can run your spimbot against itself by specifying a second file as itself.

```
spimbot -file spimbot.s -file2 spimbot.s -run -maponly -largemap
```

We will run each match in the tournament using a random map and the "tournament" option that removes some of the feedback printed out to the console.

```
spimbot -file b1.s -file2 b2.s -run -maponly -largemap -randommap -tournament
```

We have set it up that the code should work equally well as either the first or second SPIMbot. Please let us know if you find a way that your robot behaves differently as the second robot.