Objectives

- to give you experience writing MIPS assembly code
- to give you experience with functions in MIPS
- to give you experience with data and control structures in MIPS

Admin

Marks	9 (towards total course mark)
Group?	This assignment is completed **individually**
Due	by 11:59:59pm on Sunday 15th April
Submit	`give cs1521 ass1 worm.s` or via Webcms
Late Penalty	0.08 marks per hour late (approx 1.9 marks per day) off the ceiling (e.g. if you are 36 hours late, your maximum possible mark is 6.1/7)
Assessment	For a guide to style, use the code in the lectures and tute solutions, and the supplied code.

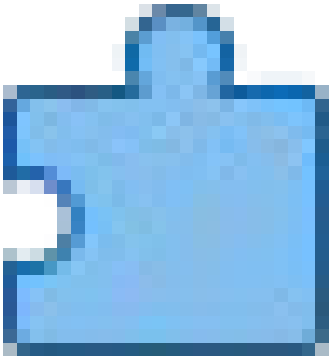
7 marks	for auto-testing on a range of parameter settings, ~~including different grid sizes~~
1 mark	for commenting the code; you don't need a comment on every line, but roughly one comment on each block of MIPS instructions that corresponds to a C statement
1 mark	for readable code; sensible names, lining up the opcodes and the args consistently

If your assembly code has syntax errors (according to `spim`) or run-time errors on all test cases, your auto-testing mark is limited to 3/7, depending on an assessment by your tutor.

Background

Snake is an old interactive ascii game where the user controls the movement of a "snake" on a rectangular grid to make the snake find and eat food, and thus grow larger. If the user ever makes the snake move back onto itself, they lose. The goal is to make the snake as long as possible before you get "stuck" and are forced to move back on yourself.

The following video gives a rough idea, although it's not strictly ascii:



Note that some versions of the game allow the snake to "wrap around" the grid (i.e. go down the bottom and reappear at the top). We do not allow this kind of movement in this assignment.

Since we don't want you wasting time *playing* the game, we're dropping the interactivity and the food. What we're left with is a simulation of something wiggling around on the grid. Let's call it a *worm* (which also sounds slightly less threatening than "snake"). The following video show how the worm should behave:

We'll describe the parameters to the simulation in more detail below. Note that the above video was created using the C solution to the assignment. You invoke the MIPS version differently, as described below, but the behaviour should be the same.

Setting Up

Create a private directory for doing the assignment, and put the assignment files in it by running the following command:

```
$ unzip /home/cs1521/web/18s1/assignments/assign1/assign1.zip
```

If you're working on this at home, download the ZIP file and create the files on your home machine. It's fine to work on your own machine but remember to *always* test your code on the CSE machines before submitting.

The above command will create the following files:

`Makefile`

A file to control compilation of `worm.c`. It is not relevant for the MIPS assembler part; it simply creates the executable C program to give you an exemplar.

`worm.c`

A complete solution, written in C. Your goal is to write a MIPS assembler program to copy the behaviour of this program.

`worm.s`

A partly complete solution to the assignment, written in MIPS assembler.

Initially, it would be worth compiling the C program and running it on some examples to get a feel for its behaviour. The compiled C program, called `worm`, expects three command-line parameters:

Length

The total length of the worm. This includes the head (denoted by `'@'`) and the body (denoted by `'o'`s). Must be in the range (4 ≤ Length < 40).

Moves

The maximum number of moves the worm should make. If the number of moves reaches this, the simulation ends. It may not reach this number of moves, because the worm might end up "blocked". Must be in the range (0 ≤ Moves < 100).

Random Seed

This value is used as a seed for a linear congruential random number generator. For a given worm length, if you supply the same seed, the worm should make exactly the same set of moves (the random number generator will generate the same sequence of "random" values).

You use the compiled C program (`worm`) as follows:


```
$ ./worm
Usage: ./worm Length #Moves Seed
$ ./worm 9 99 9999
... a worm with a 9 segments, running for up to 99 iterations
$ ./worm 9 99 54321
... a worm with a 9 segments, running for up to 99 iterations, with a
different random # seed
$ ./worm 30 40 765
... a worm with a 30 segments, running for up to 40 iterations
```


You use the MIPS version (once you've got it working) as follows:

```
$ spim  -file  worm.s 9 99 9999
... a worm with a 9 segments, running for up to 99 iterations
$ spim  -file  worm.s 9 99 54321
... a worm with a 9 segments, running for up to 99 iterations, with a
different random # seed
$ spim  -file  worm.s 30 40 765
... a worm with a 30 segments, running for up to 40 iterations
```

Both programs are structured the same, with a `main` function to handle the command-line arguments and then drive the simulation. The programs also have the same set of lower-level functions. In `worm.c`, there are comments describing the purpose of each function and the code is hopefully clear enough that you can understand how each function works. In `worm.s`, each function is preceded by comments documenting how the registers might be used within the function; note that these are suggestions only and you can use more/different registers if you wish. If you change register usage, you should update the comments to reflect this.

Exercise

The aim of this exercise is to complete the supplied program skeleton to behave exactly like the C program. The C program lives in the file `worm.c`, while the MIPS program lives in the file `worm.s`.

The simulation runs as follows:

- collect the command-line arguments and initialise the random # generator
- initialise the grid and place the worm in the middle of the middle row
- make a series of moves; for each move ...
    - decide where the worm's head should move next
    - if no possible move, indicate that the worm is blocked
    - update the positions of the worm's head and body segments
    - redraw the grid to show the worm's new position
    - delay for a while to give humans the chance to view the new state
- repeat the above until #moves is reached or the worm is blocked

The worm behaves as follows at the start of each move:

- examine the grid places adjacent to the head
- ignore any grid places that contain a body segment
- accumulate a list of possible place for the head to move
- choose one of these using the random number generator
- if there are no possible moves, indicate that the worm is blocked

In order to ensure that you pass the auto-testing, it is important that you generate the possible moves in exactly the same order as the C code does it.

In the code, the grid is represented as a two-dimensional array in C, hopefully this is obvious. In MIPS, you need to deal with a single large block of bytes, and interpret it as a sequence of rows, where each row has a sequence of columns.

In the code, the worm's position is represented by two "parallel" arrays. The `wormCol[i]` and `wormRow[i]` values, consider together, effectively give the (row,col) coordinates of the i'th worm segment. Note that `wormCol[0]` and `wormRow[0]` give the position of the head.

In `worm.s` each function has comments to:

- indicate which registers the function uses
- indicate which registers the function overwrites (clobbers)
- give a mapping between local variables in the C code and registers in MIPS

Note that these are suggestions only; you can use whatever registers you like, provided that you save and restore any `$s?` registers that you overwrite in the function code. And, of course, provided that the code behaves the same as the C code.

To save you some time, we have included function prologues and epilogues in some functions. These save and restore registers `$fp`, `$ra`, and any `$s?` registers that the function happens to use, and also maintain the stack. You can use these as templates for how to implement the prologue and epilogue in the functions that do not provide them.

Some of the functions from `worm.c` are already implemented, but others require you to write MIPS assembler for them. Here's a rundown of the functions in `worm.s` and their status:

`main`	Already complete.

`clearGrid`	Function prologue and epilogue ok. ToDo: function body.
`drawGrid`	Function prologue and epilogue ok. ToDo: function body.
`initWorm`	Function prologue and epilogue ok. ToDo: function body.
`onGrid`	ToDo: function prologue and epilogue, and function body.
`overlaps`	ToDo: function prologue and epilogue, and function body.
`moveWorm`	Function prologue and epilogue ok. ToDo: function body.
`addWormToGrid`	Function prologue and epilogue ok. ToDo: function body.
`giveUp`	Already complete.
`intValue`	Already complete.
`delay`	Function prologue and epilogue ok. ToDo: function body.
`seedRand`	Already complete.
`randValue`	Already complete.

Challenges

(Worth kudos, but no marks)

- Make the worm's movement less wiggly; disallow diagonal moves that go between two body segments.
- Make it interactive (more like the snake game) where the user can choose the next movement direction (using 'h' for left, 'j' for down, 'k' for up, and 'l' for right); if they move the head back onto a body segment, they lose.

Have fun, *jas*