COMP1521 17s2

Assignment 1
Game of Life

Computer System Fundamentals

Objectives

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

Admin

Marks	7(towards total course mark) (Note that the marks below add up to XX; they will be scaled to a mark out of 8)
Group?	This assignment is completed **individually**
Due	by 11:59:59pm on Sunday 10th September
Submit	`give cs1521 ass1 prog.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 4.1/7)

Background

Conway's Game of Life is a mathematical zero-player game whose history and rules are described in Wikipedia. It takes place on an infinite grid of cells, where each cell is either alive or dead. Each cell has eight neighbours: left, right, above, below, and diagonal. On each turn, the whole grid is examined and the content of each cell is changed according to the following rules:

- Any live cell with fewer than two live neighbours dies, as if caused by underpopulation.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by overpopulation.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

In this assignment, you will implement a version of Game of Life in MIPS assembler. It is different from the "standard" version in having a finite board. This means that some cells (on the edges) will have fewer than eight neighbours. The size of the board is determined by the data definitions included at the start of the program.

Setting Up

Create a new directory for working on the assignment, change into that directory, and then run the command:

```
$ unzip /home/cs1521/web/17s2/assigns/ass1/ass1.zip
```

This will add the following files into the directory:

- `life.c` ... a working version of the Game of Life in C
- `board1.h` ... a 10x10 board state, for inclusion in `life.c`
- `board2.h` ... a 15x15 board state, for inclusion in `life.c`
- `prog.s` ... a template for the Game of Life program in MIPS
- `board1.s` ... an initial 10x10 board state and board size
- `board2.s` ... an initial 15x15 board state and board size

The C code in `life.c` is a working version of the Game of Life. It includes an initial state of the grid, reads a number `maxiter` for how many iterations to run, and then uses the rules to change the state

`maxiter` times, displaying the state after each iteration. It is not written in a style we would normally use, but is intended to be a little closer to how you would render it in MIPS.

```
#include <stdio.h>
#include <stdlib.h>

#include "board1.h"

int neighbours(int, int);
void copyBackAndShow();

int main(void)
{
    int maxiters;
    printf("# Iterations: ");
    scanf("%d", &maxiters);
    for (int n = 1; n <= maxiters; n++) {
      for (int i = 0; i < N; i++) {
         for (int j = 0; j < N; j++) {
            int nn = neighbours(i,j);
            if (board[i][j] == 1) {
               if (nn < 2)
                  newboard[i][j] = 0;
               else if (nn ==2 || nn == 3)
                  newboard[i][j] = 1;
               else
                  newboard[i][j] = 0;
            }
            else if (nn == 3)
               newboard[i][j] = 1;
            else
               newboard[i][j] = 0;
         }
      }
      printf("=== After iteration %d ===\n", n);
      copyBackAndShow();
   }
   return 0;
}

int neighbours(int i, int j)
{
    int nn = 0;
   for (int x = -1; x <= 1; x++) {
      for (int y = -1; y <= 1; y++) {
         if (i+x < 0 || i+x > N-1) continue;
         if (j+y < 0 || j+y > N-1) continue;
         if (x == 0 && y == 0) continue;
         if (board[i+x][j+y] == 1) nn++;
        }
    }
   return nn;
}

void copyBackAndShow()
{
   for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) {
         board[i][j] = newboard[i][j];
         if (board[i][j] == 0)
```

```
58.             putchar('.');
59.          else
60.             putchar('#');
61.       }
62.       putchar('\n');
63.    }
64. }
65.
```

Note that the game board does not appear directly in the C code file. Instead it is `#include`'d, to make it easier (slightly) to include a different initial state in the program. The `board1.h` file simply contains:

```
 1. // Game of Life on a 10x10 grid
 2.
 3. #define NN 10
 4.
 5. int N = NN;
 6.
 7. char board[NN][NN] = {
 8.    {1,0,0,0,0,0,0,0,0,0},
 9.    {1,1,0,0,0,0,0,0,0,0},
10.    {0,0,0,1,0,0,0,0,0,0},
11.    {0,0,1,0,1,0,0,0,0,0},
12.    {0,0,0,0,1,0,0,0,0,0},
13.    {0,0,0,0,1,1,1,0,0,0},
14.    {0,0,0,1,0,0,1,0,0,0},
15.    {0,0,1,0,0,0,0,0,0,0},
16.    {0,0,1,0,0,0,0,0,0,0},
17.    {0,0,1,0,0,0,0,0,0,0},
18.
19. };
20. char newboard[NN][NN];
21.
22.
```

Note that the `board1.h` file contains a global variable which holds the board size. This is used by the C code, rather than referencing the `#define`'d value, to make it more like what the assembly code does.

The intention of `#include`'ing the initial board state is that people can write their own starting board states and share them with others without having to reveal their program code. Of course, you've already got the C program code, but this setup is aimed at the MIPS data/code files.

For example, the `board1.s` file looks like:

```
 1. # board.s ... Game of Life on a 10x10 grid
 2.
 3.    .data
 4.
 5. N: .word 10  # gives board dimensions
 6.
 7. board:
 8.    .byte 1, 0, 0, 0, 0, 0, 0, 0, 0, 0
 9.    .byte 1, 1, 0, 0, 0, 0, 0, 0, 0, 0
10.    .byte 0, 0, 0, 1, 0, 0, 0, 0, 0, 0
11.    .byte 0, 0, 1, 0, 1, 0, 0, 0, 0, 0
12.    .byte 0, 0, 0, 0, 1, 0, 0, 0, 0, 0
```

```
13.     .byte 0, 0, 0, 0, 1, 1, 1, 0, 0, 0
14.     .byte 0, 0, 0, 1, 0, 0, 1, 0, 0, 0
15.     .byte 0, 0, 1, 0, 0, 0, 0, 0, 0, 0
16.     .byte 0, 0, 1, 0, 0, 0, 0, 0, 0, 0
17.     .byte 0, 0, 1, 0, 0, 0, 0, 0, 0, 0
18.
19. newBoard: .space 100
20.
```

Note that the definitions are intended to be analogous to the definitions in `board.h`. One difference is that MIPS assembly code does not have an equivalent to C's `#define`, and so the board size is repeated as a constant several times within the data definitions.

Exercise

The aim of this exercise is to complete the supplied program skeleton:

```
 1. # prog.s ... Game of Life on a NxN grid
 2. #
 3. # Needs to be combined with board.s
 4. # The value of N and the board data
 5. # structures come from board.s
 6. #
 7. # Written by <<YOU>>, August 2017
 8.
 9.     .data
10. main_ret_save: .space 4
11.
12.     .text
13.     .globl main
14. main:
15.     sw   $ra, main_ret_save
16.
17. # Your main program code goes here
18.
19. end_main:
20.     lw   $ra, main_ret_save
21.     jr   $ra
22.
23.
24. # The other functions go here
25.
```

You *must* implement the three functions given in the C code: `main()`, `neighbours()`, and `copyBackAndShow()`. You do not need to implement stack-based function calls, but can if you wish.

For testing, you will need to load both a board definition and your program code into the MIPS emulator to get a complete executable program. This is easy in `qtspim`, where you can load multiple files into the memory before executing them. The `spim` command only accepts one code file, and so you need to merge a `boardX.s` file and the `prog.s` file to create a complete program. A simple way to do this is to run two commands:

```
$ cat board1.s prog.s > life.s
$ spim -file life.s
# Iterations: 3
=== After iteration 1 ===
##........
```

```
##........
.###......
....#.....
....#.....
...##.#...
...##.#...
..##......
.###......
..........
=== After iteration 2 ===
##........
..........
####......
..#.#.....
....#.....
..........
..........
.#........
.#.#......
..#.......
=== After iteration 3 ===
..........
..........
.###......
..#.#.....
...#......
..........
..........
..#.......
.#........
..#.......
$
```

In order to conduct testing, you could compile the `life.c` program, run it to collect the expected output, then run the SPIM version to collect the observed output, and compare them, e.g.

```
$ dcc life.c   ... or gcc -std=c99 if doing at home ...
$ echo 3 | ./a.out > c.out
$ echo 3 | spim -file life.s > mips.out
$ diff c.out mips.out
... we expect no output here ...
$
```

If the output from `diff` is empty, then the MIPS program is behaving as expected.

You should try this for at least the two supplied boards. Remember that you will need to edit `life.c` to change what's `#include`'d, then re-compile, and then build a new `life.s` by merging `prog.s` with the corresponding `boardX.s` and repeating the above testing.

It would be great if people created new, more interesting, initial board state files and made them available for others to use in their testing.

Things you should not do:

- use a cross-compiler to convert the above C code to MIPS assembler
- copy Victor Torres' or anyone else's solution from an online source like GitHub

The whole point of this exercise is for **you** to better understand how assembly programs are built and how they work. The above two strategies ruin the point of this assignment.

Extensions

(Worth kudos, but no marks)

- create some larger and more interesting initial board states
- run the game continuously and use additional characters to provide animation

Have fun, *jas*