Project Life

Out: 23 Oct 2009
Helpsession: 26 Oct 2008, Time TBD
Design Check: 27-30 Oct 2008
Due: 9 Nov 2009, 11:59pm

0 Description:

This assignment is based upon an invention by the British Mathematician J. H. Conway (in 1970) called "Life". It is a simulation of a population of lifeforms (or organisms) over a sequence of generations. The framework of the population is a two dimensional array (or grid) of cells. Each cell can be inhabited by at most one organism. The game starts off with an arbitrary initial population (dictated by whether a particular cell contains an organism or not). Occupied cells are referred to as "alive", whereas unoccupied cells are "dead". From this initial population the next generation of organisms is obtained by applying the following rules:

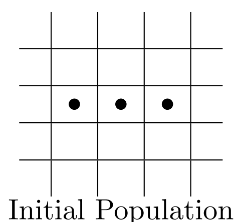
1. The neighbors of a cell are the 8 cells that immediately surround it vertically, horizontally and diagonally.
2. If a cell is alive and has 2 or 3 live neighbors, it will remain alive in the next generation.
3. If a cell is alive and has fewer than 2 live neighbors, it will die of loneliness.
4. If a cell is alive and has 4 or more live neighbors, it will die of overcrowding
5. If a cell is dead and has exactly 3 live neighbors, a new organism will be born in that cell. Otherwise, it remains dead in the next generation.
6. All births and deaths take place at exactly the same time, so that all the cells' neighbors are counted simultaneously (based on the current generation) before the next generation is produced. It is possible for a new cell to be born based on counting a neighbor in the current generation that will be dead in the next generation.

This simulation is interesting because from very simple initial configurations, quite complicated progressions of Life communities can develop.

For an interesting online life simulator go to http://sciris.shu.edu/thinklets-/Games/Life/Life.html.

1 Example:

Let's look at a section of an (infinite) grid (where full box=alive, and empty box=dead):



Initial Population

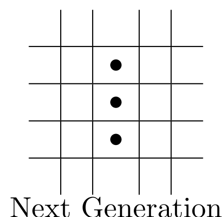
Here we have three cells that are alive (assume all cells not shown are dead). For each cell we will first count how many of its eight neighbors are alive. The upperleftmost cell is not adjacent to any living cells, so its neighbor count is zero. The same goes for all the cells along the top and bottom rows. Moving to the leftmost cell of the second row, we see it has exactly one living neighbor, the one diagonally to the lower right. Going through the entire grid this way will result in the following count of living neighbors for each cell.

0	0	0	0	0
1	2	3	2	1
1	1	2	1	1
1	2	3	2	1
0	0	0	0	0

Neighbor Count

We now combine the neighbor count with the information of whether it was alive or dead to generate the new population. The second cell of the second row has a neighbor count of two. If it had been alive, it would have stayed alive according to the rules outlined. However, since it was dead it will stay

dead in the next generation. The center cell also has a neighbor count of two, but because it was alive it stays alive. Just above the center cell is an example of a dead cell coming alive since it has three living neighbors. The next generation will look like the following:



Next Generation

2 Assignment Requirements:

You should first write Java/C/C++ code for the above algorithm to be presented at the design check. Then you can move on and implement your life in assembly, which will constitute the final handin. We strongly recommend that your Java/C/C++ design be fully functional before moving on to assembly.

2.1 The Specifics:

Please read the details of this section carefully, as you will be marked down for criteria that are not met, particularly those that we rely on for testing. You need to implement this simulation in MIPS assembly language (MAL) using mipscope. The grid is obviously best kept in a two dimensional array. Note, however, that the array is not necessarily square and your program should take this into account (i.e. make the array dimensions adjustable parameters in the data section). Your program should be well structured and modular so that each logical function to be performed is represented by a separate subroutine.

Of course, now that there are various subroutines, you need to pass them information (such as passing the location of the cell to the routine that computes the number of neighbors). In this assignment, you are allowed to do parameter passing to your routines in $a registers. So, for example, you can pass the x and y coordinates in two registers. (See the lecture slides for register usage conventions.) All such parameter passing must be documented

clearly in header comments and inline comments. When passing the return value of a subroutine, use $v registers.

An alive cell is to be represented with '1' and a dead cell is to be represented with '0'. You should print out (the initial population and) each generation after you have calculated it. You should print them out (to the mipscope console) so that it looks like a grid (i.e., print out one row of the array per line and make sure the columns line up nicely). As long as the columns line up correctly any extra spacing or formatting is optional. Also, print an extra newline at the end of each generation to make the different generation grids distinct. You should make the program perform this loop of printing and calculating generations for as many generations as specified by some easily adjustable variable in the program. For the assignment, there will be five (5) generations (including the starting generation), but do not code this number directly into the loop's test instruction, use a memory location declared in the data section.

To aid in the process of testing your code, you must have:

- The first few lines of your `.data` section at the top of your file should be constant declarations for the number of `ROWS` in your life array, the number of `COLUMNS` in your life array, the number of `GENERATIONS` your program will run, and the `ARRAY_SIZE`, which is the result of multiplying `COLUMNS` by `ROWS`. You should use these constants in your code, using these exact all-caps names. The number of generations includes the first generation, so one generation should just print the initial setup. Your code should initially have 10 columns and 8 rows. Since MIPS doesn't allow arithmetic in the .data section you will have to hand calculate `ARRAY_SIZE` as being `COLUMNS` times `ROWS`. Your code can assume that the constants are correct, so don't worry about checking for error conditions such as `ARRAY_SIZE` not being the correct number, but do make sure that you don't accidently set it wrong or your code will behave very poorly. During testing we will only change these constants, so make sure you use them everywhere in your code that you need these values.

- There must also be a subroutine named `add_live_cell` which takes parameters row in register $a0 and column in register $a1. This subroutine should put a live cell at that location in your initial array, **using zero-based indexing**.

- We will be using the above code to test your program, so please indi-

cate where to set up the initial population using the `add_live_cell` subroutine. Put the line `#jal add_cells_for_testing` into your code. When we run your code we will uncomment this line and add this subroutine. You SHOULD NOT write an `add_cells_for_testing` subroutine; we will do this.

- Once you have tested your Life to your satisfaction, you should remove your own calls to `add_live_cell` so that we can initialize the array the way we want it. In other words, if our `add_cells_for_testing` subroutine is empty, your program should run with an all-dead array.

- Before every generation, you should print "`---`" (three dashes) on a line of its own. After every generation, print "`===`" (three equals) on a line of its own. You can print whatever kind of information you want in between generations – perhaps the generation number, the amount of cells still alive, or whatever interesting statistic you can cook up. The first two generations of a simple simulation might look like this:

```
Generation 1:
---
0 0 0
1 1 1
0 0 0
===

Generation 2:
---
0 1 0
0 1 0
0 1 0
===
```

- Please post questions about these guidelines to the newsgroup. Adhering to them will make grading your assembly much easier, meaning you will get your project back promptly.

The rules stated for "Life" assume an infinite grid in all directions. Since you are limited to a finite size grid, the problem arises of what to do on the borders of the grid where some cells do not have all eight neighbors. For this assignment you will assume that any neighbor that does not exist in the grid is a dead cell.

3 Subroutines That May Come in Handy:

Even though you are no longere required to implement two versions of this program, modularity in your code will make your "life" better. It will also make your code more readable and easier to debug, especially for assembly. Consider this very high-level pseudocode for how this program will run:

```
initialize the arrays
for # of generations
  for each cell
    count neigbors
    determine next generation state
  print board
  swap arrays
```

While you could write this all in one sloppy block of code which would be difficult to grade (hint: that's a bad thing), everyone will be happier if you break these steps up into subroutines. Learn to love `jal` and `$ra`.

4 Other Helpful Hints:

In case you think you only need to allocate one array for this program, think again. You will need two. Since the generation updates for each cell are based on the previous generation, you cannot make any changes to the previous generation until the new generation has been completely calculated. You should try to do this without copying the array each time.

Despite what you might think from the example shown in the description section, you do not have to generate the entire neighbor count array before generating the next population. Calculating a cell's next generation value just after you count its number of neighbors is the better way to go.

Generating output to the mipscope console window will require you to use system calls. You can use the `print_int` system call and just reserve an integer (1 word = 4 bytes) for each cell. This is quite space inefficient, since we are using 4 bytes to represent 1 bit. If you try hard to imagine the extra (tedious) work of packing and unpacking bits in and out of a word you will start to get the feel of what real assembly language programming is all about.

To get one row per line printing out on the mipscope console you will need to output a newline character at the end of each row. Use the print_string system call and have the line:

```
newlinechar: .asciiz ``\n''
```

in the data section. This sets up a string consisting solely of the newline character. Making the system call with newlinechar as the string address parameter will advance the cursor to the beginning of the next line in the mipscope console.

When you first start to debug your program, use very small populations (e.g. 4 x 3). You should make the size of the array adjustable so that you can easily change it while debugging. Remember, do not assume the array has to be square.

Try to keep the functionality of each subroutine to a minimum. This will make your "life" (you're going to get very sick of this pun) easier. The more you try to do in a routine, the harder it will be to debug.

Avoid the general use of global variables (see the next paragraph for the two exceptions). See the cs031 MAL style guide for anin-depth discussion of when global variables are appropriate and when they are not. Note that this doesn't mean you can't use declarations in your .data section, since those are required. In MIPS we define global variables as being a value loaded into a register in one subroutine and then used in a different subroutine.

You are allowed to use exactly two global variables: one for the current array, and one for the array that it will be switched with. The registers which contain these should be clearly documented.

5 Grading Criteria:

You will be graded on the following criteria:

- Does the program actually do what it's supposed to? Is it stable?
- Good code structure (loops, if-else, branching)
- Modularity (good break up and use of subroutines, avoidance of global variables)

- Compliance with the cs31 coding conventions
- Commenting (note that this includes register tables at the beginning of every subroutine)

6 Handing In:

You should turn your program electronically by typing:

```
cs031_handin life
```

7 Extra Credit:

If you do either of these, be sure to keep a copy of the original program, so that if the extra credit crashes, you can get credit for the standard code. If you are doing the extra credit, you must hand in the standard code along with the code which incorporates the extra credit. If you do not do this then you will not receive extra credit. Also, you should not start on the extra credit until you have a beautiful and working standard program. Be sure to clearly comment what you have done.

1. (5 points) Make your world a torus instead of a plane. That is, have it wrap around so that the bottom row is really considered to be adjacent to (and above) the top row and the right column is adjacent to (and left of) the left column. You get really cool effects when you have gliders wrapping around the edge of screen. Implementing this will overrule the dead cell assumption for nonexistent neighbors.
2. (5 points) Have each cell in the array represent one of four possible states. Instead of just alive and dead add newborn and just died (i.e. dead, but not quite cold). In this case the cell's value is dependent not just on whether the organism is dead or alive, but on the previous generation as well. A square will contain 3 if the "organism just died", 2 if the "organism was just born", 1 if the "organism stayed alive" and 0 if the "organism stayed dead". This option is much more interesting when the numbers are actually colors, but it retains some of the interest since it's still 5 extra points. If you're doing this extra credit you should initialize the population at the start of the program as being

"just born", which means that your `add_live_cell` subroutine should set cells to "just born".

8 Life Design Check

The checkpoint is 20% of your grade. Please come prepared with:

1. Well thought-out, written, complete pseudo code. This should translate directly into the assembly for the project. It can be Java, C, C++ or anything else that is easily understandable. Be prepared to walk through it with a TA when you come for your design check.
2. A list of your subroutine names and a hierarchy graph showing how each one is called (i.e. subroutine names with arrows showing the flow of control).
3. A testing plan.
   The TA will expect you to have a written testing plan that demonstrates you have put time into thinking about testing. You should include some test cases to consider.

Here are some things you should keep in mind when designing Life:

- How will your program be broken down into subroutines?
- How will your pseudocode translate into assembly?
- What, if any, special cases do you need to think about?