Assignment 8: Life

CSE30: Computer Organization and Systems Fall 2021

Instructors: Bryan Chin and Keith Muller

Due: Monday, November 22nd, 2021 @ 5:00PM

Please read over the entire assignment before starting to get a sense of what you will need to get done in the next week. REMEMBER: Everyone procrastinates but it is important to know that you are procrastinating and still leave yourself enough time to finish. **Start early**. **You MUST run the assignment on the pi cluster. We are using custom libraries that are only available on the pi-cluster for this assignment. You HAVE to SSH: You will not be able to compile or run the assignment otherwise.**

NOTE: Please read the FAQ and search the existing questions on Edstem before asking for help. This reduces the load on the teaching staff, and clutter/duplicate questions on Edstem.

Table of Contents

Learning Goals

- Programming in ARM assembly
- Memory manipulation in ARM with loads and stores
- Loops in ARM
- saving and restoring preserved registers

Assignment Overview

In this assignment, you'll be implementing the meat and potatoes of Conway's Game of Life: The logic behind the cells on the board living and dying. You will do this by writing a function in ARM that iterates over one column of the board and updates all the cells' states.

1. Getting Started

Developing Your Program

For this class, you **MUST** compile and run your programs on the **pi-cluster**. To access the server, you will need your `cs30fa21cxx` student account, and know how to SSH into the cluster.

Need help or instructions? See Edstem FAQ or watch CSE30 Development Tools Tutorial
(Do NOT wait until Friday to try this. There will be limited or no ETS support on the weekends!)

We've provided you with the starter code at the following GitHub repository.

1. Clone/download the GitHub repository.[1]
2. Fill out the fields in the `README` before turning in.
3. Open your favorite text editor or IDE of choice and begin working.

Running Your Program

We've provided you with a `Makefile` so compiling your program should be easy!

Additionally, we've placed the reference solution binary at:
`/home/linux/ieng6/cs30fa21c/public/bin/mclife-a8-ref`.
You can directly use `mclife-a8-ref` as a command.

Makefile: The `Makefile` provided will create a `mclife` executable from the source files provided. Compile it by typing `make` into the terminal. By default, it will run with warnings turned on. To compile without warnings, run `make WARN=` instead[2]. Run `make clean` to remove files generated by `make` - important if you have an existing executable and you want to recompile a new version.

Starter Code: There are a few initial config files that you can load into the program. Your code will be called from the libraries[3] `libcse30life.a` and `libcse30liferv.a` located at `/home/linux/ieng6/cs30fa21c/public/local/arm/lib`. The Makefile will reference these libraries and you shouldn't need to do anything to get it to work.

gdb: To run with gdb, the command is `gdb [executable]` to start gdb with your executable. `r [arguments]` will run your executable with `[arguments]` as the command-line arguments. For tips on how to use GDB with assembly code, refer to this Edstem post.

---

[1] If you use your own Github repo, make sure it is private. Your code should never be pushed to a public Github repository. **A public Github repo is an AI violation.**
[2] But you should not have warnings, so you should fix them.
[3] Libraries are precompiled routines that may be used by multiple programs. You've already used functions from the c-standard library like "printf" and "strcmp". We made our own library of some cse30 functions for this simulation.

2. The Game of Life

Developed by John Conway, the Game of Life is a mathematical simulation of a simplified evolution process. The simulation occurs on a two-dimensional grid of cells over a series of discrete timesteps. At any given timestep, each cell will either be “alive” or “dead” (encoded as 1 and 0 respectively), and each cell’s value for the next timestep will be computed based on the current values of its 8 adjacent neighbors. The next-state rules are defined as follows:

For a live cell:

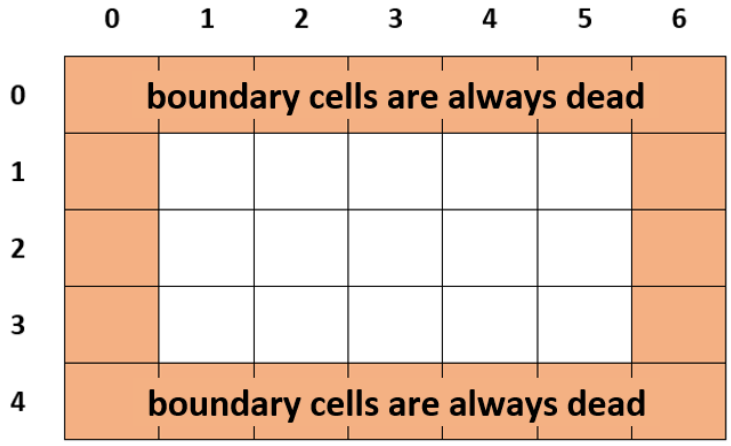
If 0 or 1 neighbors are alive, it will die in the next timestep from loneliness.
If 2 or 3 neighbors are alive, it will remain alive in the next timestep.
If 4 or more neighbors are alive, it will die in the next timestep from overpopulation.

For a dead cell:

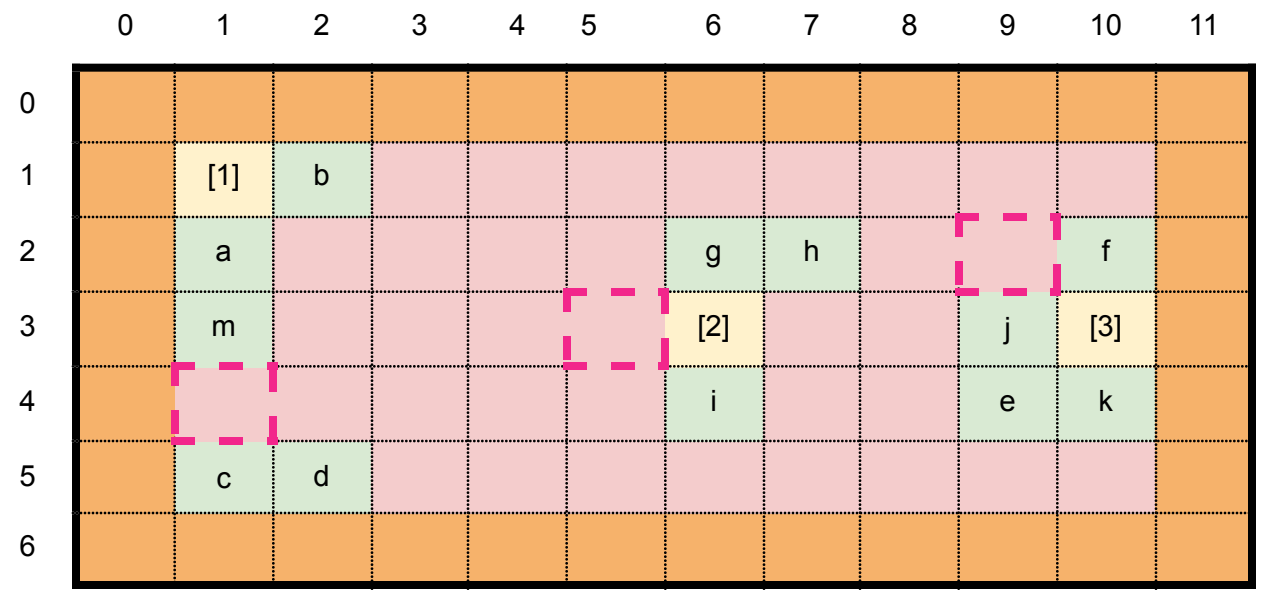
If exactly 3 neighbors are alive, it will become alive in the next timestep.
Otherwise, it will remain dead in the next timestep.

See this site for an interactive example of the Game of Life. See the Wikipedia page on Example Patterns to see moving gifs of common patterns (toad, glider, etc).

The original life algorithm assumes an infinite plane of cells. However, we are working with limited memory and will simplify the game by working with a finite board and assuming that all the board edges can never sustain life. For example, for a board of 100 x 200 cells, only the inner 98 x 198 active cells could potentially support life. Thus the boards you implement will schematically look like the figure below (shown for a 5 x 7 board).



For an example of how cells are processed, consider the following case where we have a 7 x 12 board. In the board shown below, cells with text are live. The boundary cells shown below never support life.



Cell [1] (1,1) has 2 live neighbors (a, b), Cell [2] (3, 6) has 3 live neighbors (g, h, i), and cell [3](3,10) has 4 live neighbors (e, f, j, k). In the next timestep, cells [1] and [2] are guaranteed to remain alive since they have 2 and 3 live neighbors respectively. Cell [3] will die as it has 4 live neighbors. The highlighted cell at (4, 1) has 3 live neighbors (m, c, d) and will become alive in the next time step. Similarly (3, 5) and (2, 9) have 3 live neighbors and will also become alive in the next time step.

Our implementation represents the 2-dimensional life boards as 1-D arrays. We represent the boards in a format called row-major order. That is, we represent the first row in the first $numCols$ entries of a 1-D array, and the next row in the next $numCols$ entries, and so on for each row. Notice that for any 2-dimensional position on the board given as a 2D location (row, column), you can access it in the 1D array at $index = row * numCols + col$. Think about how the 8 neighbors are located relative to a cell in a 1D array. For more details, refer to this link.

row\col	**0**	**1**	**2**	**3**	**4**	**5**	**6**
0	0	1	2	3	4	5	6
1	7	8	9	10	11	12	13
2	14	15	16	17	18	19	20
3	21	22	23	24	25	26	27
4	28	29	30	31	32	33	34

3. How the Program Works

- This program loads an input file, defining the initial board layout of dead/alive cells.
- You are able to interactively step through the program to see how the board evolves through each timestep.
- All argument processing is handled by the provided libraries. See arguments and interactive commands below for details.

Program Arguments

Format for calling this executable with arguments: `./mclife filename`

Arguments	Description
`filename` (required)	The name of the input file to initialize the dead/alive cells of the array. See below for details on the format.

Interactive Commands

After running the executable, interact with it by entering the following commands:

Command	Description
`s N`	Simulate `N` steps, displaying the intermediate results.
`n N`	Simulate `N` steps without displaying intermediate results.
`d filename`	Dump the current state of the board to `filename`. The file can then be used as input when running the executable.
`q`	Quit the program.

The Input File

The input file for generating a board has the following format:

```
[numRows]
[numCols]
[row col]  ← only lists row-col coordinates of alive cells
...
[row col]
```

Ex: A file that contains the following

```
4
3
1 1
2 1
```

produces a 4 row x 3 col game board with the cells at (1, 1) or (row 1, col 1) and (2, 1) or (row 2, col 1) starting out alive. Note: Remember `row` is the y coordinate and `col` is the x coordinate and by convention (0,0) is upper left corner.

Refer to files like `config.small`, `config.big`, `config.fancy`, etc. in the starter code repo to see examples of input files. Some of the larger config files may produce output that may exceed your terminal width and wrap to the next line. This can be fixed by changing to a smaller font size, zooming out on the terminal or redirecting the output to a file for viewing later.

All config files will have `numRows` and `numCols` with a minimum value of 3. That is, you should not consider board sizes smaller than 3 rows by 3 columns. Although we will not define a maximum board size, at some point the program will not have enough memory to hold the state of the board. Because of this we will not test with excessively large boards.

Config files will *never* specify live cells in the boundary rows and columns.

4. Function to implement

You have to implement a single function for this assignment. You will be implementing this function in assembly. Write your code in `doCol.S`.

- Assume non-degenerate input.
- We have provided correct implementations of the other necessary Life functions as library functions. **Your version of `doCol` MUST work with our library functions.** Look at the Style Guide for examples to write/modify the prolog and epilog.

`void doCol(unsigned char *dest, unsigned char *src, size_t rows, size_t cols)`

Arguments:

`unsigned char * dest` → Pointer to the first cell to update on the destination board (first non-boundary cell in a column)

`unsigned char * src` → Pointer to the corresponding cell on the source board (first non-boundary cell in a column)

`size_t rows` → The number of rows in the board including boundary rows)

`size_t cols` → The number of columns in the board including boundary columns

Operation:

- The first position to update is the cell in row 1, column N (*not* row 0).
- Count the number of living neighbors around this cell **in the source board**.
    - A cell has 8 neighbors surrounding it.
- Based on this count and the state (live/dead) of this cell **in the source board**, calculate what the next state of this cell should be.
- Write this new state (alive/dead) to this cell **in the destination board.**
- Iterate to the cell in the next row.
    - If our current position is (row, col) = (y, N), then the next position should be (y+1, N), assuming (0, 0) is the top left corner of the board.
- Once you have updated the cell at (row, col) = (`rows-2`, N), you are done. Return from the function.

You might notice that the function is called `asm_doCol` in the code. This has no effect on its functionality and is only used to distinguish it from C versions that the libraries may use to test.

Why 2 boards?

We want to store a cell's next value separately from its current value so the cell's neighbors can be computed correctly. We need two boards as a way to differentiate between the "current" and "next" board. (Think: Why do you think this is? What would happen if we updated each cell using only one board?)

`doCol()` processes one column of a board at a time. At a high level, the algorithm is:

For each cell in the column (other than the cells in the boundary rows), calculate whether that cell is going to be dead or alive in the next time step, based on the rules outlined in Section 2. Put the value that the cell will have in the next iteration into the destination board. A 0 value indicates "dead" and a 1 value indicates "alive."

doCol Tips

Here are some tips for simplifying `doCol()`.

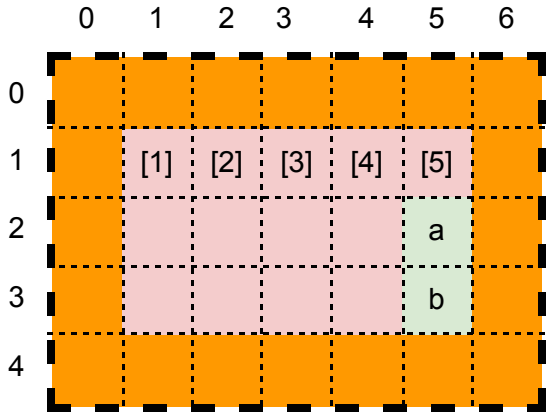
- The function `doCol()` takes 4 parameters. The 4 parameters will be in R0 to R3 in the order they are listed in the function signature.

Considerations for the main loop:

```
while (curRow < nRows - 1){
      // do your stuff
 }
```

- What possible values can be stored in each cell?
    - Is it possible to calculate the number of live neighbors without if statements to check neighbors' life states?
- Adjust `FP_OFFSET` according to the directions in the assembly template.
- R0-R3 need not be preserved across function calls.
- Pay attention to preserved registers. If you use them, you must save and restore them in the prologue and epilogue. For example, if you use r5 and r6, you need to push r5 and r6 in the prologue, pop r5 and r6 in the epilogue, and adjust `FP_OFFSET` accordingly.
- Each cell is represented with an `unsigned char`.
    - A live cell has a value of 1, otherwise it has a value of 0. How can you use this fact to calculate the number of live neighbor cells?
    - What sized load and store instructions should you use?

Example Calls to doCol():



Given this 5x7 board (above), our simulation loop will call `doCol()` 5 times with the following parameters, once for each column 1 to 5:

1. `doCol`(pointer to cell [1] in destination board, pointer to cell [1] in the source board, 5, 7)
2. `doCol`(pointer to cell [2] in destination board, pointer to cell [2] in the source board, 5, 7)
3. `doCol`(pointer to cell [3] in destination board, pointer to cell [3] in the source board, 5, 7)
4. `doCol`(pointer to cell [4] in destination board, pointer to cell [4] in the source board, 5, 7)
5. `doCol`(pointer to cell [5] in destination board, pointer to cell [5] in the source board, 5, 7)

As an example of processing one column, when processing column 5, your `doCol()` function will process cells `[5]`, `a` and `b` as shown on the figure above.

Allowed Instructions

You are only allowed to use the instructions provided in the ARM ISA Green Card. **Failure to comply will result in a score of 0 on the assignment. Do not use addressing modes or features of the ARM instruction set that are not on the green card.**[4]

5. Midpoint

This part of the assignment is due earlier than the full assignment, on Friday 11/19 at 5pm. There are no late submissions.

Complete the Gradescope assignment "A8: Midpoint", an Online Assignment that is done entirely through Gradescope. This assignment consists of 4 short questions about this writeup, and a free-response question where you will document your `doCol()` algorithm. This should be either a pseudocode or a C version of `doCol()`. This is a planning document and does not need to reflect your final implementation, although you are encouraged to keep it up to date.

6. Testing

We recommend running your program on an input file and dumping the result of the board into some file A. Then, run the solution executable on the same input file and dump the result of the board into some file B. `diff` the two files to see if there are differences. Run your program for varying numbers of simulation timesteps (more than just one timestep).

Test standard simple Life patterns to help you debug things like blinker, glider, toad and beacon (see https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life).

Some additional ideas: Test large boards and small boards. Try isolated cases, maybe no cells alive, or many cells alive, and see if your solution produces what you expect.

Note: Don't worry about Valgrind testing for this assignment.

Suggested Debugging/Development Tips

This program is not bigger than prior ones and there is only one function to implement. However, we still would recommend these steps.

---

[4] Some examples of forbidden features are conditional execution, setting condition code bits with other than CMP instructions.

1. START EARLY.
2. Read the instructions carefully.
3. Check the FAQ often and search Edstem for previous questions, if possible.
4. Outline the algorithm in pseudocode. Tutors may ask to see this!
5. If you are working on a bug, use GDB. You can set breakpoints at interesting points (e.g. set a breakpoint at the start of your loop, and step through your code to watch it calculate the number of living neighbors). You can make better use of tutor lab hours if you have a debug session to show them. If you have a segmentation fault, run GDB before going to a tutor. Otherwise, they may not have time to help you, especially when the queue gets busy.
6. TESTING: **List** all the test cases you can think of (re-read this document). There are not as many unique test cases for this assignment, but you should still always think through test cases for every assignment you do.

   The public autograder will only test some features. DO NOT RELY ON THE AUTOGRADER (many CSE30 students have been burned by this). **Test your code using your own test cases.**

7. Style Requirements

5 Points WILL be given for style for each part, and teaching staff won't be able to provide assistance or regrades unless code is readable. Please follow these Style Guidelines for ARM assembly. **Note: Don't mix tabs and spaces for indents or else the code on Gradescope will look misaligned.**

8. Submission and Grading

Submitting

1. Submit your files to Gradescope under the assignment titled "Assignment 8". You will submit the following files:

   ```
   doCol.S
   README.md
   ```

   You can upload multiple files to Gradescope by holding CTRL (⌘ on a Mac) while you are clicking the files. You can also hold SHIFT to select all files between a start point and an endpoint. Alternatively, you can place all files in a folder and upload the folder to the assignment. Gradescope will upload all files in the folder. You can also zip all of the files and upload the `.zip` to the assignment. Ensure that the files you submit are not in a nested folder.

2. After submitting, the autograder will run a few tests:
    a. Checks that all required files were submitted.
    b. Checks that the source code compiles without warnings.
    c. Runs some tests on the resulting `mclife` executable.

Note that passing these initial tests does NOT guarantee that your program will receive full credit on the hidden test cases or even works for any of the hidden test cases! Test very carefully to ensure that you catch errors even after submission to Gradescope.

Style Requirements

Points WILL be given for style, and teaching staff won't be able to provide assistance or regrades unless code is readable. Please follow these Style Guidelines for ARM assembly.

Grading Breakdown [50 pts]

Make sure to check the autograder output after submitting! We will be running additional tests after the deadline passes to determine your final grade. Also, throughout this course, make sure to write your own test cases. ***It is bad practice to rely on the minimal public autograder tests.***

The assignment will be graded out of 50 points and will be allocated as follows:

- **Midpoint writeup:** 5 points. **This part of the assignment is due earlier than the full assignment, on Friday 11/19 at 5pm.** Complete the Gradescope assignment "A8: Midpoint".
- Public tests with the provided config files.
- Private tests with custom, hidden config files.
- 5 points for style. See above ARM assembly style requirements.

NOTE: The end to end tests expect an EXACT output match with the reference binary. There will be NO partial credit for any differences in the output. Test your code - DO NOT RELY ON THE AUTOGRADER FOR PROGRAM VALIDATION (READ THIS SENTENCE AGAIN).

Make sure your assignment compiles correctly through the provided `Makefile` on the pi-cluster without warnings. **Any assignment that does not compile will receive 0 credit.**

Make sure to check the autograder output after submitting! We will be running additional tests after the deadline passes to determine your final grade.

Checking For Exact Output Match

A common technique is to redirect the outputs to files and compare against the reference solution[5]. In this assignment, there is a `d` "dump" command when you run the program to save the state of the board to a file. Here is an example that runs for 10 time steps and then dumps the output.

```
./your-program arg

/* on program command-line: */ d output

/* on program command-line: */ n 10

/* on program command-line: */ q

./our-reference arg

/* on program command-line: */ d ref

/* on program command-line: */ n 10

/* on program command-line: */ q

diff output ref
```

If the final command outputs nothing, there is no difference. Otherwise, it will output lines that differ with a < or > in front to tell which file the line came from. Note that in this program there is FILE CREATION from the dump command. This means that these files, instead of `stdout`, should be `diff`'d.

END OF INSTRUCTIONS, PLEASE RE-READ!

Writeup Version

1.0 Initial release.

[5] You might want to check out `vimdiff` on the pi-cluster (https://www.tutorialspoint.com/vim/vim_diff.htm).