CMPUT 229

Lab #4: Game of Life Bits

CMPUT 229

Background

Overview

Simulating the Game of Life

In this lab you will write an interactive simulator for Conway's Game of Life.

The simulation will use RISC-V's interrupt handling facilities for keyboard input and timing.

Output will be performed using the MMIO terminal display in RARS.

Conway’s Game of Life

A simulation that follows the evolution of a square grid of cells.

Each cell can be either living or dead.

Each step of the simulation, every cell is updated according to a set of rules.

Simulation Rules

A living cell that has less than two living neighbours dies.

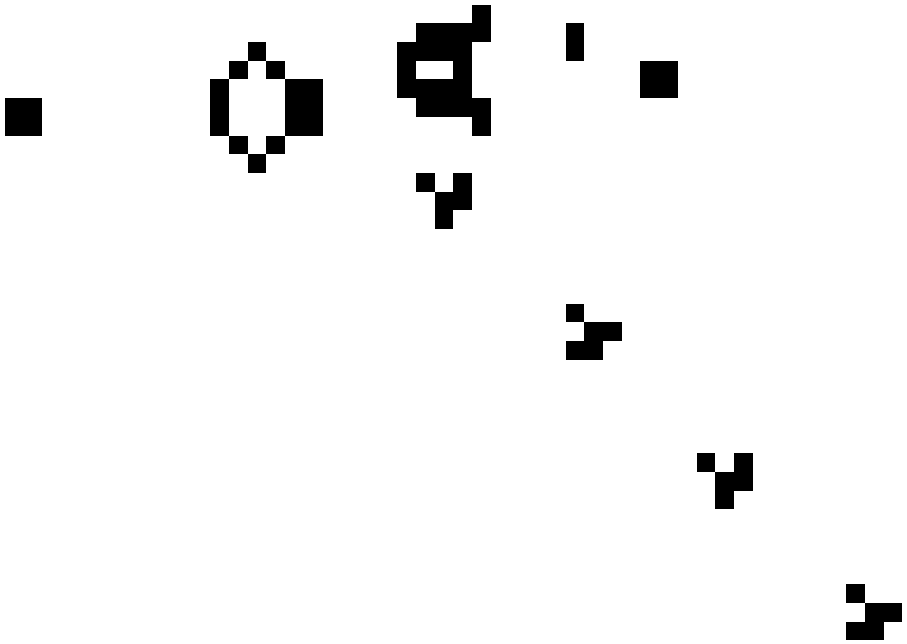
A living cell that has more than three living neighbours dies.

A dead cell that has exactly three living neighbours becomes alive.

Otherwise, the cell’s state doesn’t change.

Example simulaiton:

These simple rules can lead to complex behaviour:



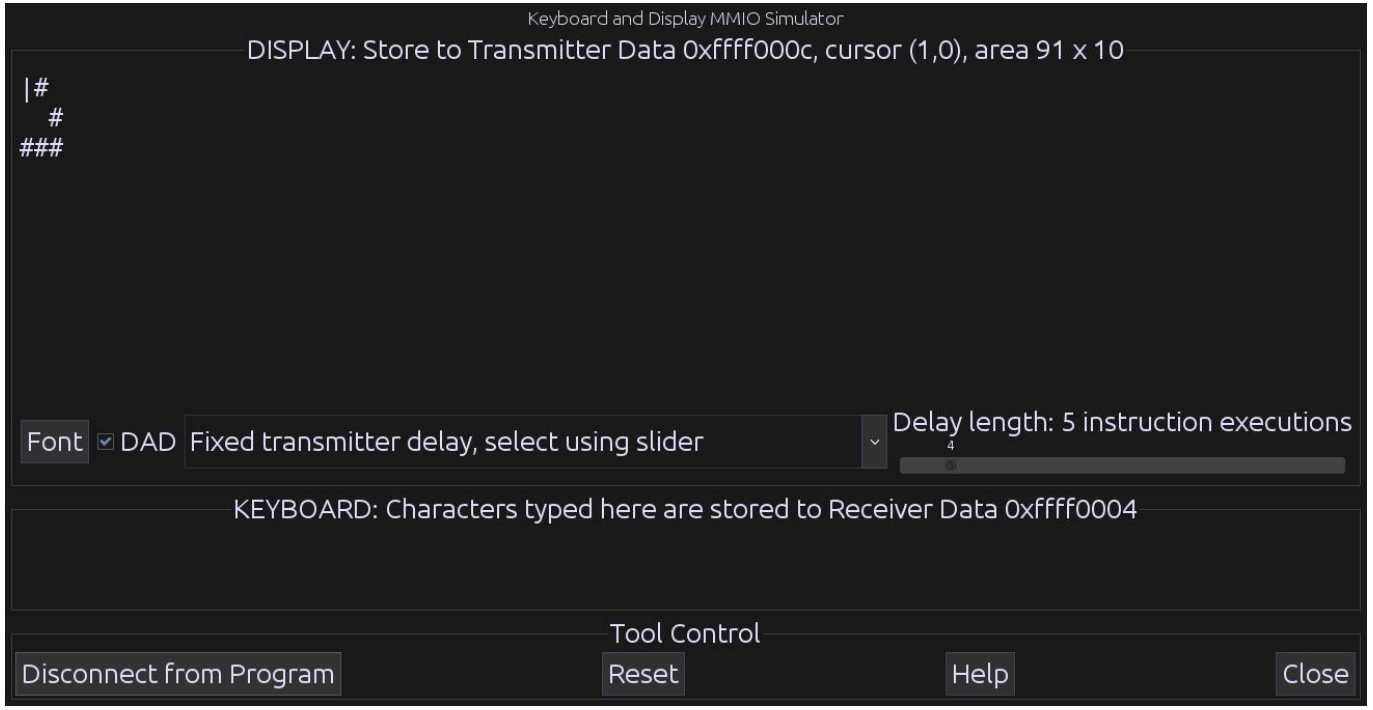
Interactive Simulator

Operation of the Simulator

The simulator will display a grid of cells and a cursor.

The user can control the simulation, and cause the simulation to update.

The simulation can be moved to a “running” state, where it steps automatically.



Controling the Simulator

Interacting with the Simulator

Keyboard controls are used to interact with the simulator.

The cursor can be moved up, down, left, and right with the `w`, `s`, `a`, and `d` keys.

The cell that the cursor is on can be set to the living state with the `j` key, and set to the dead state with the `k` key.

A single step of the simulation can be performed using the space key.

The state of the simulator can be switched between the “paused” and “running” states by pressing the `t` key.

Pressing the `q` key exits the simulator.

Interrupts

Interrupts

The simulator uses interrupts to interact with the hardware and external devices.

Specifically, the simulator uses timer and keyboard interrupts.

When an interrupt occurs (say a key press) the interrupt handler is called.

The interrupt handler must deal with the interrupt, then return control back to the running program.

Control and Status Registers

RISC-V uses “control and status” registers to configure interrupts.

Interrupts are disabled by default in RARS, so they must be enabled using these registers.

The address of the interrupt handler must also be set.

Refer to the website for specific information on which registers to use and how to use them.

RARS Keyboard & Timer Tools

Keyboard & Display Tool

Found under the “Keyboard and Display MMIO Simualtor” tab under the “Tools” menu in RARS.

After assembling your program, click the “Connect To Program” button.

If you resize your terminal, click the “Reset” button.

Timer Tool

Found under the “Timer Tool” tab under the “Tools” menu.

Press “Connect To Program” to connect the timer to your assembled program.

MMIO

Memory-Mapped IO

Memory-mapped IO allows interaction with external devices through an interface “pretending” to be system memory.

We can communicate with such devices using regular memory instructions.

Both the Keyboard / Display and the Timer tools use MMIO.

A list of the addresses of all MMIO registers can be found on the website.

Keyboard & Display MMIO

Keyboard MMIO Registers

Keyboard Control: Enables keyboard interrupts.

Keyboard Data: Contains the ASCII value of the last keypress.

Display MMIO Registers

Display Control: Indicates whether or not the display can be written to.

Display Data: When a character is stored here and the display is ready, the character is written to the display.

Timer MMIO

Timer MMIO Registers

Time: Contains the time given in the timer tool. Stored in milliseconds.

Timecmp: When the value in this register is less than or equal to the value in the Time register, a timer interrupt occurs.

Writing an Interrupt Handler

An interrupt handler is similar to a regular function.

The interrupt handler is run during the execution of the program. Thus the interrupt handler cannot appear to change any of the registers.

Registers cannot be saved to the stack, since the stack pointer may be corrupted. For example, consider an exception thrown for a missaligned load using the stack pointer.

The `uscratch` register contains a pointer to memory that can be used by the handler.

This memory can be used to store the original values of registers that are used by the interrupt handler.

CMPUT 229

Simulator

Game of Life Simulator

Input to the Simulator

The inital state of the cell grid is given by the input file.

This file is parsed and given to your solution as a cell grid buffer.

Stepping the Simulator

At each step of the simulation, the cells in the grid are updated according to the Game of Life rules.

We cannot use a single cell grid, since if we update the cells in place, the neighbour counts may change during the update.

This may cause the simulation to compute the next state incorrectly.

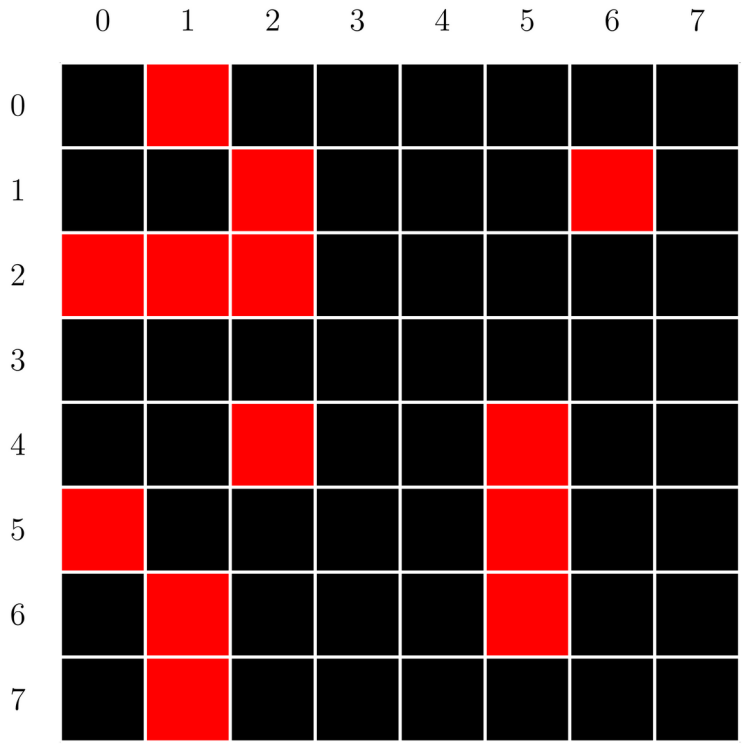
To solve this issue, we can use two buffers. One buffer contains the current state. During the simulation step, the results are written to the second buffer.

Cell Grids

Format of the Cell Grid

The state of the simulation is stored in a byte array, referred to as a “grid”.

The bit `0` represents a dead cell, while living cells are represented by the bit `1`.



	0	1	2	3	4	5	6	7
0	0	1	0	0	0	0	0	0
1	0	0	1	0	0	0	1	0
2	1	1	1	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	1	0	0	1	0	0
5	1	0	0	0	0	1	0	0
6	0	1	0	0	0	1	0	0
7	0	1	0	0	0	0	0	0

Cell Grids in Memory

How are Cell Grids Stored?

Cell grids are stored in row-major order.

This grid:

	0	1	2	3	4	5	6	7
0	0	1	0	0	0	0	0	0
1	0	0	1	0	0	0	1	0
2	1	1	1	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	1	0	0	1	0	0
5	1	0	0	0	0	1	0	0
6	0	1	0	0	0	1	0	0
7	0	1	0	0	0	0	0	0

Memory layout:

Address	Binary Value	Hexadecimal
0x10001000	00000010_2	0x02
0x10001001	01000100_2	0x44
0x10001002	00000111_2	0x07
0x10001003	00000000_2	0x00
0x10001004	00100100_2	0x24
0x10001005	00100001_2	0x21
0x10001006	00100010_2	0x22
0x10001007	00000010_2	0x02

Wrapping the Grid

How Wrapping Works

When travelling off the edge of the simulation grid, the simulation wraps around to the opposite edge.

Any point `(x, y)` can be mapped to a grid with `r` rows and `c` columns using the following formula:

`(x mod c, y mod r)`

However, in this lab, we are only concerned with cells that are in the grid, or are one cell outside (including the corners).

Your solution only needs to handle these cases.

Printing to the Display

Printing to the MMIO Diplay:

The MMIO display cannot be printed to using `ecall`.

Instead, we provide the `printCell` function, which can print a single character to a given row and column of the display.

The characters to use for living cells, dead cells, and the cursor are given in the `ALIVE_CHAR`, `DEAD_CHAR`, and `CURSOR_CHAR` variables, respectively.

printCell

Description:

Print a character to the MMIO text terminal at (row, col).

Parameters:

`a0`: Character to print as an ASCII byte.

`a1`: Row to print the character at.

`a2`: Column to print the character at.

Return Value:

N/A

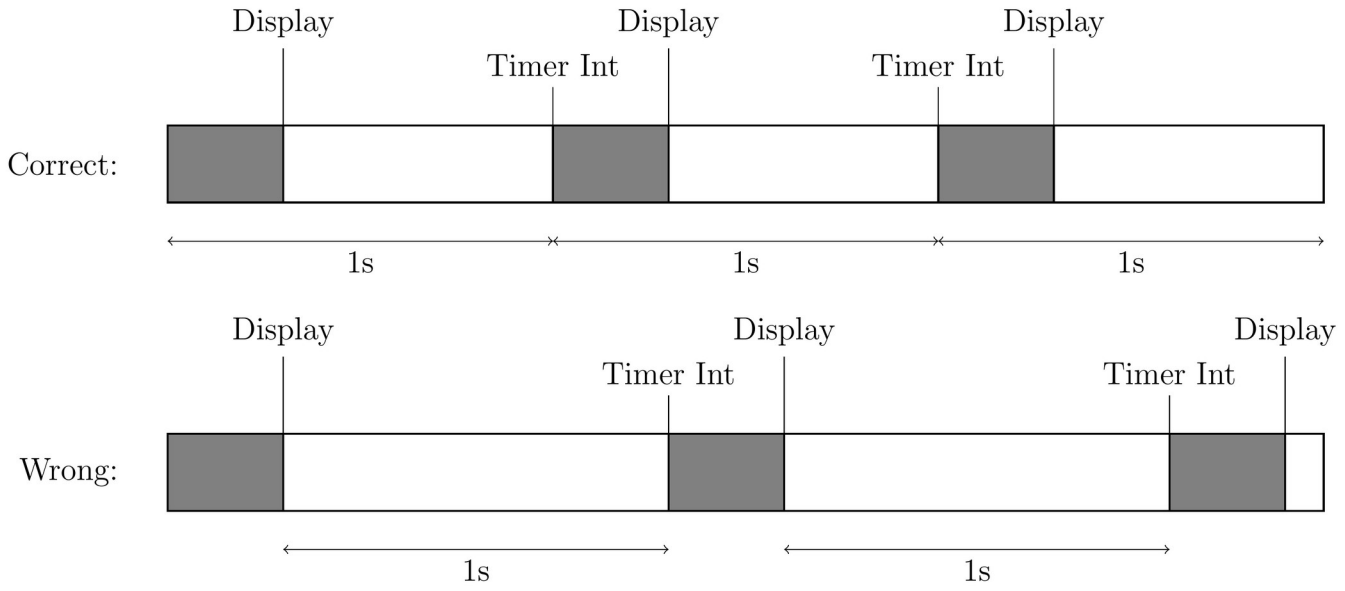
Timing

How Timing Works:

When the simulator is in the “running” state, an iteration of the simulation is performed once per second.

The timer must be reset before beginning to update the simulation.

Otherwise, the time between updates may be more than one second.

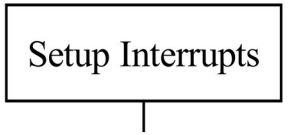


Controlling the Simulator

Simulation Controls:

`w`	Move the cursor up one character, wrapping if needed. Then update the display.
`s`	Move the cursor down one character, wrapping if needed. Then update the display.
`a`	Move the cursor left one character, wrapping if needed. Then update the display.
`d`	Move the cursor right one character, wrapping if needed. Then update the display.
`j`	Set the cell at the cursor to the living state, then update the display.
`k`	Set the cell at the cursor to the dead state, then update the display.
`t`	Toggle the simulation between the “running” and “paused” states.
`space`	Step the simulation and update the display. Only works when in the “paused” state.
`q`	Exit the simulation.

Flow of the simulator



CMPUT 229

Assignment

Overview of the lab

Overview:

In this lab, you must implement a series of funtions to create a simulator for the Game of Life.

You are required to implement all of the following functions.

gameOfLife

Description:

Entry point for the Game of Life simulator.

Parameters:

`a0`: Number of rows in the grid.

`a1`: Number of columns in the grid.

`a2`: Pointer to grid buffer A (Initializied with the input state).

`a3`: Pointer to grid buffer B.

Return Value:

N/A

displayGrid

Description:

Display the given Game of Life grid to the MMIO terminal.

Parameters:

`a0`: Number of rows in the grid.

`a1`: Number of columns in the grid.

`a2`: Pointer to grid buffer to display.

Return Value:

N/A

updateGrid

Description:

Perform a single Game of Life simulation step. Reads from one buffer and writes the result to another.

Parameters:

- `a0`: Number of rows in the grid.
- `a1`: Number of columns in the grid.
- `a2`: Pointer to the input grid buffer.
- `a3`: Pointer to the output grid buffer.

Return Value:

N/A

getCell

Description:

Get the value of a given cell. If the location is out of bounds, wrap around.

Parameters:

`a0`: Number of rows.

`a1`: Number of columns.

`a2`: Row of the cell.

`a3`: Column of the cell.

`a4`: Pointer to the grid buffer.

Return Value:

`a0`: Value of the cell.

setCell

Description:

Set the value of a given cell. If the location is out of bounds, wrap around.

Parameters:

`a0`: Number of rows.

`a1`: Number of columns.

`a2`: Row of the cell.

`a3`: Column of the cell.

`a4`: Pointer to the grid buffer.

`a5`: New value of the cell.

Return Value:

N/A

handler

Description:

Interrupt handler for the Game of Life simulator.

Parameters:

N/A

Return Value:

N/A

CMPUT 229

Testing

Testing your Solution

Included tests:

We have provided some inputs for you to test your solution with.

These inputs are stored in the “Tests” directory as `*.txt` files.

Format of the test file:

Make sure the test file has the correct format (refer to the website).

RARS may need the full path to the test file.