ECE241 Final Project Report

Conway's Game of life

Prepared by: Noah Poplove, Viktor Kornyeyev on December 4, 2017

Introduction

The goal of this project was to effectively implement John H. Conway's Game of Life on a DE1-SoC board with Verilog code. Our motivation was to investigate and create a project related to how FPGAs are able to perform computations with large amounts of data. At first this interest manifested itself into a signal processing direction, where we focused on implementing audio filters. After some trouble manipulating the audio controller modules for our project, we decided to analyze a different format of large data -- a 2 dimensional grid of states.

Conway's Game of Life takes place on a 2 dimensional board of cells that are each specified by x and y coordinates. Each cell is in either an alive (1) or dead (0) state. Each cell interacts with its eight neighbours, which are the cells directly adjacent to it. The game naturally evolves under the following rules, where at each specified clock cycle, the next iteration is given by:

(1) A live cell with less than two live neighbours dies due to underpopulation
(2) A live cell with more than three live neighbours dies due to overpopulation
(3) A live cell with two or three live neighbours lives until the next generation
(4) A dead cell with exactly three live neighbours becomes a live cell due to reproduction

Building upon these standard rules, we implemented "periodic boundary conditions" as a way to increase the overall lifespan of the game, as well as produce more interesting results. Periodic boundary conditions develop a topology whereby cells passing through one side of the board, reappear or interact with cells on the opposite side of the board. In the context of the Game of Life, this amounts to a "wrap-around" effect whereby cells on the edges of the board interact with other cells on the opposite side(s). Thus, each cell is able to interact with 8 neighbours.

An example of the above rules (without periodic boundary conditions) is shown in **Figure 1**. Given an initial state, processing these computations to identify the live and dead cells in the next generation requires inspecting every cell's neighbours. For a 32x32 board running at 4Hz, that's 32,768 comparisons a second.

As our guiding principles, we set out to create an interactive, visually stimulating, and enjoyable experience

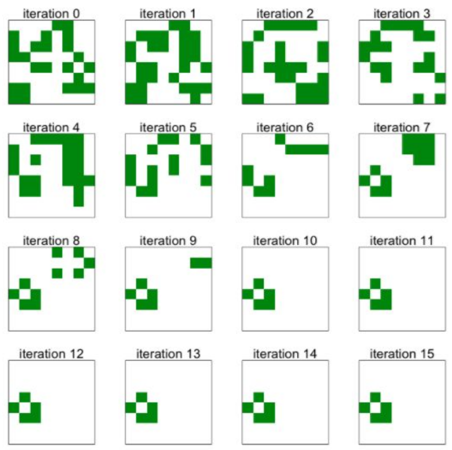

Figure 1

for users. To accomplish this, we implemented the following tasks:

- Display the grid data on the VGA
- Display a background on the VGA
- Create a number of interesting preset starting grid states that the user can interact with
- Utilize the switches to let users input a starting grid state
- Provide feedback data about which coordinate(s) the user is modifying
- Compute the next grid states given a previous state based on previously defined rules
- Continuously show the number of live cells on the HEX displays on the DE1-SoC as the game evolves

The Design

We decided to write all of our modules into one file, “GameOfLife.v”, so we could more easily make iterations of our code without having to check through multiple files to make sure inputs/outputs match when they are changed in other modules. The below block diagram (**Figure 2**) is a top level view of our code organization:

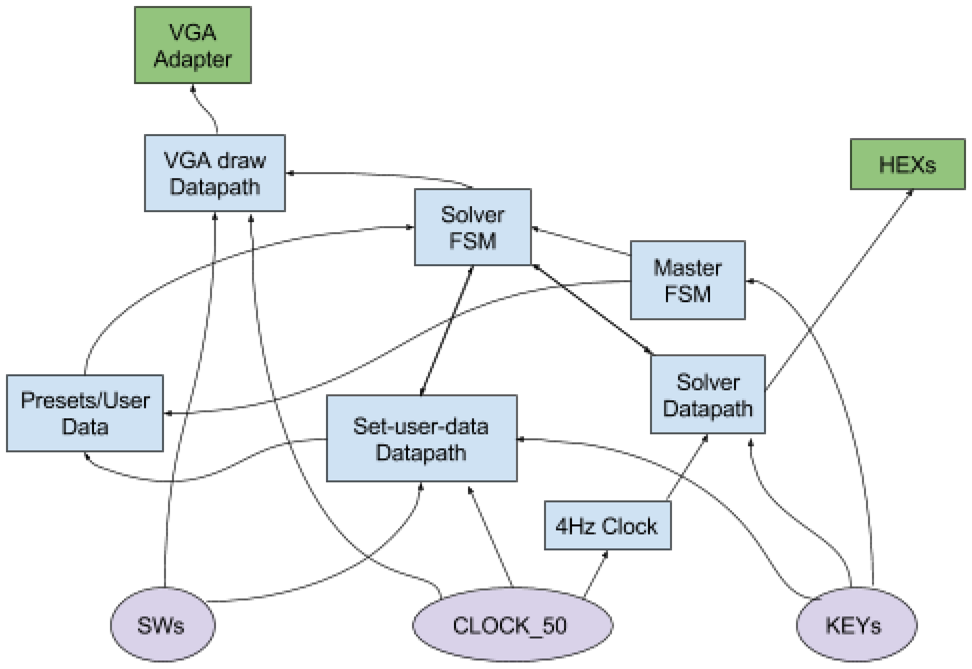

Figure 2

The top level module “GameOfLife” contains declarations of all wires, registers, presets and parameters, as well as some sequential code and module instantiations necessary to model the game. The following are descriptions of each block from the diagram.

Master FSM - currentGameStateFSM()
The main FSM operates on two inputs provided by KEY[3] and KEY[2]. KEY[3] toggles the state of state variable "go", which is what enables or disables the solver doing its computations. KEY[2] toggles the state of "preset", which sets whether the user will input the initial grid, or if the grid is one of the 8 presets declared in our code.

Solver Datapath - nextStateSolver()
This is the primary "solver" code, which produces and updates the next state for a given input array based on the previously defined rules. Running at "clock" speed (4Hz in our implementation) it continually updates the output array as long as the state variable "go" is enabled. The number of live cells is constantly sent to be shown on the HEX displays, as shown in **Figure 3**.

Solver FSM - gameOfLife()
This FSM depends on the state of the state variable "go". When "go" is low, the FSM checks the value of "preset", and either sets the output-grid to the selected preset, or sets the grid to an empty grid that can be customized with SW's and KEY[1] through the Set-user-data datapath.

Set-user-data Datapath - userSetLocation()
This datapath enables the user to update the array (i.e. create a live or dead cell at a chosen location) through use of the on board switches and KEY[1].

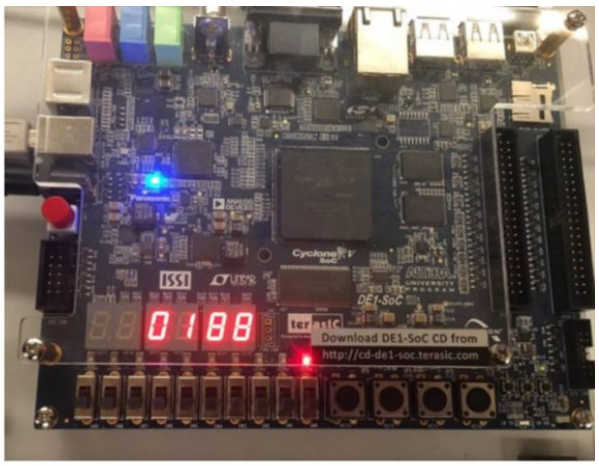

Figure 3: Mid-game view of board displaying number of live cells

Presets/User Data - gameOfLife()
This block consists of all the preset wires that are assigned in the top level module, as well of the grid from the Set-User-Data datapath. These are sent to the Solver FSM, where the selected one is sent to the actual solver module.

VGA draw Datapath - fill()
The top level module contains an always block that loops through x and y to be sent into the VGA adapter. The fill module compares any xy-coordinate with the current grid given by the solver module. Then it either outputs to the current xy-coordinate the color white for dead, or red/pink for alive. It also outputs to the VGA adapter the current x and y coordinates inputted on the SWs (eg. X: 12, Y:04), as well as the current preset (PRE# where # is the preset number, or USR for user input). The background .mif is also declared here.

Success Report

Reflecting upon our previously defined guidelines (as specified in the Introduction), we believe we have effectively implemented all our tasks.

Firstly, we were able to display both the grid data and background effectively and neatly on the VGA. Next, we were able to create and implement a set of interesting presets, as well as let the user input their own design into the grid. To make it easier and more intuitive to use, we provided an interface to do so through onboard switches and keys. Additionally, we provided real time feedback through the HEX displays and on screen text so the user would have a better idea of the state of the game and what they are inputting.

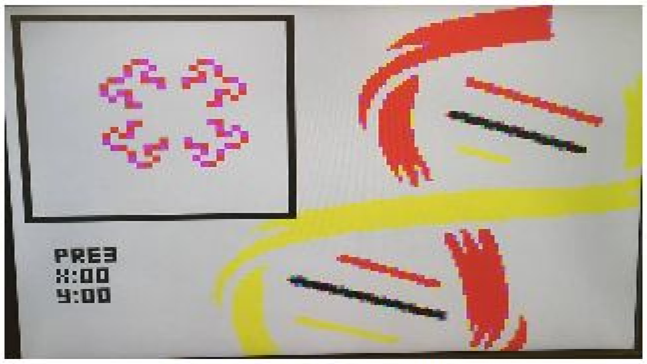

Figure 4: Preset #3

At the core of the Game of Life, lies the next state solver which we implemented first with standard boundary conditions, then with periodic boundary conditions. To provide actual in-game feedback to the user, we configured the onboard HEX displays to display the number of live cells at any given time.

Overall, we succeeded in creating modules to input and preset arrays of cells, and a datapath to find the next state of the cell life, all while colorfully displaying the contents and providing feedback to the user.

What We Would Do Differently

Looking back, some of our methodology for writing our code was based more in C/C++ style programming, as opposed to hardware description programming. By doing so it took us longer than expected to finish some modules that required extensive logic behind them such as the state-solver module. We initially used nested for-loops to check each cell's neighbours in the state-solver, but we could not get the code to give the correct output until we approached the code using a hardware datapath structure.

The grid used to store the cell states was made using an m×n length array register and wire. While this simplified the coding logic, we quickly learned that as the array size increased, so did the amount of FPGA resources it took up. In turn, our compile times became unmanageably long. By the time we were increasing our grid size, we had already wrote most of the computational code, but if we were to go back we would use RAM to store the states of the grid instead of the huge arrays.

Lastly, we only used Modelsim to test a few simple modules at the beginning of our project. Some of our reasons for not using it were: difficulty reading the VGA output variables and relating them to the grid positions of cells, tediousness of writing large .do files, and the amount of modules we would have had to test in modelsim. Instead of running simulations, we just compiled scaled down versions of our code (i.e. smaller array parameters) each iteration and tested the inputs and outputs on the FPGA and VGA display. We ran into numerous problems when testing our code on the hardware, including having some iterations having no visual output to the display. These problems were frustrating, and making them worse was the increasingly long compile times for our code as we added more modules. If we were to start all over again, we would more extensively utilize modelsim for each module so as not to deal with the aforementioned problems.