Project Report

Michael Henson
School of Electrical and Computer Engineering
Oklahoma State University
Stillwater, Oklahoma
michael.a.henson@okstate.edu

Callee Webb
School of Computer Engineering
Oklahoma State University
Stillwater, Oklahoma
callee.webb@okstate.edu

I. INTRODUCTION

This project was a test of sorts. Students were given less at the start to complete this design than usual, providing ample opportunity to be creative as well as testing us to see what we have learned thus far.

All of our past labs were useful during this project, as the designing and implementation process included creating a test bench and a finite state machine, among other fun features.

II. BASELINE DESIGN

The basic design for this project was an imitation of *Conway's Game of Life*. We were given code to start with and we had to add onto it to make it work.

Conway's Game of Life is a cellular automaton. This means that a user initializes the seed and then the program continues without user interference.

We accomplished the most basic form of this program by using a finite state machine to determine whether the game should be running or not. Once the user flips a designated switch, the program will run indefinitely.

Conway's Game of Life adheres to 4 rules, as seen in Figure 1, that determine the state of the cells.

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

Figure 1 (Rules from Wikipedia)

The seed input by the user will determine the outcome of the game. For every cycle of the program, every cell will be compared with the rules above to determine if the cell should be dead or alive. Once every cell has been checked, the entire grid will update to the next generation and the cycle will repeat.

III. DETAILED DESIGN

With the basic outline of the game completed, we implemented additional elements to improve it.

Mainly, we created a finite state machine in order to control the state of the game, instead of having the program run continuously.

We also increase the grid size in order to display more complicated seeds; our other additions include extra seeds and colors to switch between.

Our FSM has three states, plus a failsafe fourth.

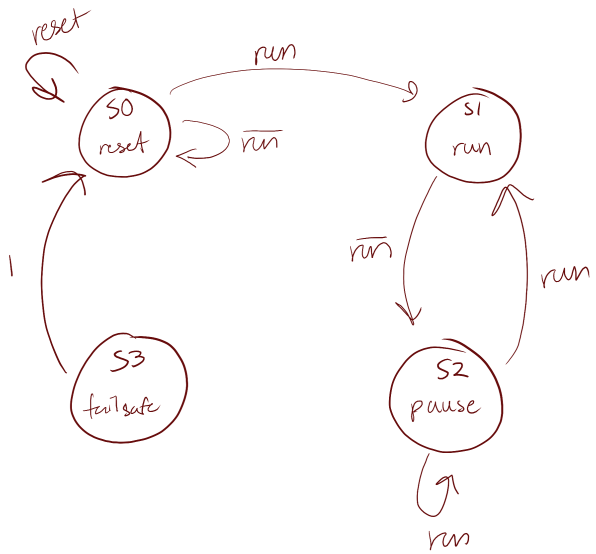

Figure 2 (Simplified FSM)

State zero of our FSM is the reset state; When the game is in this state, it will revert the grid to its initial seed. State one handles running the game. When in this state, the grid will evolve until it either cannot evolve anymore or until it is paused.

This brings us to State two; the pause state. When in this state, the clock is stopped and the grid will freeze on its current generation. State three is irrelevant, and was only implemented as a failsafe. In the case it is somehow reached, this state will set the current state to zero.

As for increasing the grid size, we wrote two programs in c++ to automate the process of rewriting the blocks of code used to update the grid; one for the code that updates the grid itself, and the other for the HDMI module that will display the grid.

Seeing that the grid has hundreds of tiles, making and using programs to generate new code was more efficient than typing it out by hand, which would have left plenty of room for error. Increasing the grid size also included changing the bit size of any port or variable that held a seed.

Seeds! One of our other, much less complicated implementations was a switch case in vivado that allows the user to flip between different seeds, so that hard-coding a new one each time is not an issue. We included two unique seeds: one is a Christmas tree, which I thought was appropriate given the season, and the other is multiple flip-flopping bits of threes.

Alongside multiple seeds, we also implemented cases for different color combinations. Red and green is included to fit our Christmas tree seed.

Implementing color changes was slightly more complicated, as it required creating new input logic in the given HDMI module, and then figuring out how to assign them to a switch. Despite the difficulties, though, users are able to change colors mid-evolution.

IV. TESTING STRATEGY

Usually, ModelSim is our most effective tool to test our designs. However, it was not so useful for this project, and was really only used to ensure there were no errors in our code. We attempted to display the grid and its evolutions with ModelSims waveforms, but that was not very visible or reliable. Instead, we created an output file to display our grid with ones and zeros to have a visual confirmation that our code was working as intended. Figure 3 displays the grid using ModelSim waves.

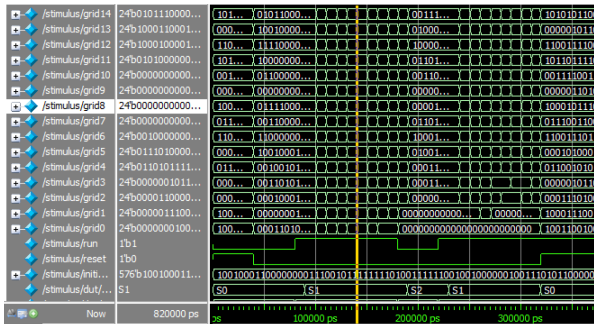

Figure 3 (ModelSim Testing)

As for our main testing strategy, after verifying there were no errors in our code and seeing that our grid was evolving as intended, we put the design into Vivado in order to debug easily. Of course, this meant that there was a delay in testing something new, but our output file got us close enough that it was worth the short waits. Being able to clearly see what was happening made debugging fairly simple.

For example, at one point, we forgot to change the size of a port. We figured out that it was a port size issue because our grid was visually off; only half of the grid seemed to be working. This indicated that we had smaller numbers than we needed somewhere, so we were able to diagnose and address the problem.

V. EVALUATION

Finally, after finalizing the design in vivado, we ran the implementation to analyze the impact of the design on performance, power, and area. We determined that the design runs at 125 Watts, with 42 nanoseconds, and a 36% Utilization.

To review our main design elements and fun features, we have: pause, reset, and run states, an upgraded grid size, multiple seeds and switches to change them, and multiple colors that utilize switches as well. Our grid size was upgraded from 16x16 to 24x24, which is 256 bits to 576!

This project was a lot of trial and error, but we are very happy with our end result. We have made exceptional progress throughout our labs, and this project is the product of our efforts.