Digital Logic Design Final Project

Oklahoma State University

FPGA Game of Life Implementation

Author
Garrett Page

Lab Partner
Bryan Struble

Instructor
Dr. James Stine

March 21, 2023

Contents

1 Introduction

In this lab, I designed an implementation of John Conway's Game of Life for a Zynq-7000 FPGA board. The lab ties together all of the concepts that we learned in class throughout the semester, from clocks, to combinational logic, to finite state machines. I had to create our own testbench capable of debugging the project as I went along, which was a challenge we had not been assigned in any previous labs.

1.1 Design Specifications

The baseline design for the project required a simple eight by eight Game of Life grid that uses a finite state machine to transition the game's state to the next state in sequence. It also required a pause and reset switch that set the game back to its initial state. The modules to update the game's state to the next state were provided to us in a datapath.sv file, making the primary task to build the finite state machine implementation and testbench.

1.2 The Design

The design that I completed was similar to the required baseline design, however I added a few improvements, including a cursor and toggle button that lets you modify the state of the game while the game is paused, a pleasant change in color scheme from the dreadful blue and red to a more modest yellow and orange, and improved flexibility of grid size.

1.3 Testbench and Debugging

The testbench includes three debug methods: waveform, display output, and a custom renderer that renders each frame as a .tga image file that is converted to a .png file and can be viewed in any image viewer. A setup generator was designed to easily and visually create custom setups for testing that could be changed, and another utility allowed the size of the grid to be changed from eight by eight to any other desired size. The HDMI file provided did not provide a way to add more grid size without changing hundreds of lines of pre-existing code, so only the eight by eight grid was synthesizable, however a few simple changes to the hdmi file would allow any grid size to be synthesizable.

2 Baseline Design

The game of life is a set of grid cells that can be placed on any arbitrary sized grid. The baseline design required an eight by eight table of grid cells. Each cell can be either alive or dead during a generation, and the game forever cycles through generations, with each generation being derived from a set of rules applied to the last generation. The rules of the game are as follows:

- Any cell with fewer than two live neighbors dies.
- Any cell with two or three live neighbors lives to the next generation.
- Any living cell with more than three neighbors dies.
- Any dead cell with exactly three live neighbors becomes alive.

The FPGA implementation had to apply these rules in parallel using combinational logic. Each generation would be cycled through over time and displayed on the screen. Flipping the reset switch would stop the evolution of the game and reset it back to its original state.

The design, project files, and example output can be found at the first GitHub reference at the end of this report.

2.1 Changes to the Baseline Design

The implementation included a few non-trivial changes to the design specifications. Instead of using a constant and predetermined setup for the game of life to cycle through, I instead started with an empty grid and used a new mode to provide a user-defined setup before the simulation runs.

2.1.1 Control Mode

The design contains a new mode called control mode. When the pause switch is enabled, control mode provides a built-in cursor which can be moved by pressing a left, right, up, or down button. A fifth button was added to toggle the cell selected by the cursor between alive and dead. With the pause switch enabled, the game does not cycle to the next generation, allowing the state of the game to be modified however the user wants before restarting the simulation in the desired state. This modification required a more complex design strategy and implementation.

2.2 Testbench Design

The testbench provided three different modes of testing. The waveform was the primary method of debugging the internal finite state machine, however both a text output and image output were also created to verify and visualize the Game of Life simulation as it progressed through testing. Because of the new control mode, no fixed initial setup could be used, so a setup had to be generated just like it would be inputted into the device during a live simulation in control mode – using the left, right, up, down, and toggle buttons to input a user-defined setup. The testbench had to follow these same principles when setting up a test simulation. Due to these restrictions, I made a few utility scripts to generate setups for simulation.

2.2.1 Image Generation

The testbench has an option to enable image generation by a custom-made renderer. A renderer output module detects when the game state is updated by the simulation and creates a formatted .tga image file depicting a grid with the alive cells highlighted and dead cells unhighlighted.

2.3 Utility Scripts

I created three Python utility scripts to provide flexibility and make testing the more complex system easier and more efficient.

- **gendatapath.py** takes a grid-size as input (ex. 8 for 8x8) and automatically generates the datapath module used to compute the rules for a grid of the given size.
- **gensetup.py** is used to create a testbench setup. It takes a grid-size as input and builds a text file with a grid of 0s and 1s that the tester can modify. The modified grid will be used to auto-generate simulated left, right, up, down, and toggle button presses that a real user would input into control mode to acquire the defined setup.
- **convertrenders.py** takes the output .tga files from the custom renderer and converts them into .png files for increased portability.

These scripts are tied together through a run.bat file in charge of both setting up and running a simulation. The run.bat script takes two inputs: grid size and number of test generations before quitting. It runs all three utility files when necessary as well as the run.do file to run the testbench through the specified number of generations before exiting.

2.4 Attempt at 3D Rendering

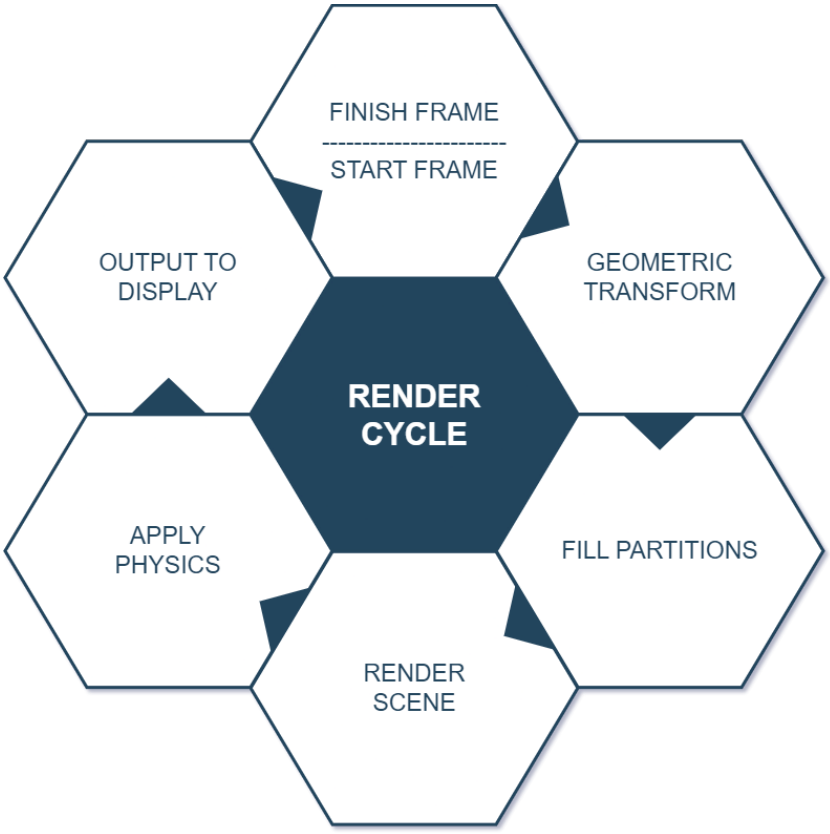
I did make an attempt at creating a more complex 3D game of life which does deserve an honorable mention. The first step was creating a 3D rendering engine strictly in

SystemVerilog, a large challenge on its own. I was able to complete a rendering engine capable of generating a working 3D rendered video, however I stopped short of converting the game of life logic into 3D geometry due to time constraints, instead settling for a single 3D rotating cube.

2.4.1 A Short Summary

The rendering engine contains one central finite state machine which loops through a sequence of rendering steps once per frame.

Figure 2.1: Render state diagram



- **START**
  - Resets the partition map
  - Updates and increments the frame count by one.
- **GEOMETRIC TRANSFORM**
  - Transforms the scene, which is a class storing a collection of models containing a color value and triangle vertices, into a set of cached screen-coordinate models used to compute the pixel buffer.

- **FILL PARTITIONS**
  - Instead of triangle-by-triangle rasterization, the system takes a raytracing approach and separates the screen into partitions of equal size. These partitions are then boundary checked with each model, and a reference to each triangle crossing the partition is stored for the rendering process.
- **RENDER**
  - The render step loops through all partitions and renders the geometry within each partition at every pixel.
- **PHYSICS**
  - The engine does a physics pass that modifies position values based on a velocity and rotational velocity value associated with each respective model. This is how the frame-by-frame rotation is achieved in the example.
- **DISPLAY**
  - This is the final important step, sending the pixel buffer generated in the RENDER step to the tgadisplay function which uses the image renderer created in the baseline design to output each frame to a .tga file.
- **FINISH**
  - Checks whether the current frame has reached the number of frames requested to be processed and ends the simulation once the last frame has been rendered.

2.4.2 Math

The rotation math is quaternion and matrix based, and the sin, cos, and tan functions required for generating rotation quaternions and matrices were derived from fifth level Taylor expansions. Other math involved includes a point collision function used to detect whether a pixel intersects a triangle's screen coordinate, a matrix class including perspective projection and translation functions, quaternions and quaternion-matrix conversion, four-dimensional vectors, and more mathematical utility functions. The non-synthesizable *real* data type was used for testing purposes, although it could be replaced with a synthesizable fixed-point precision 8x24 data type capable of running the math on the fpga.

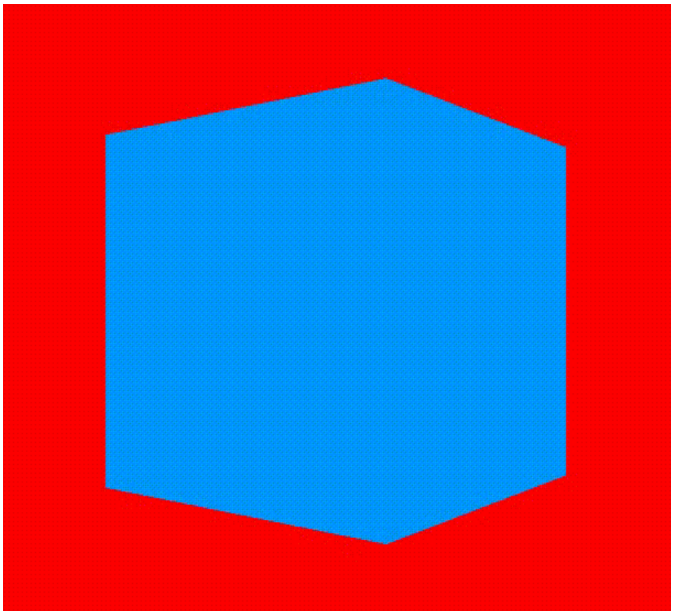
2.4.3 Limitations

Converting all geometry in a scene requires surprisingly few division operations, making the rasterizing process mostly parallel over relatively few clock cycles. Once the geometry is converted into screen coordinates, each partition would be rendered one after the other using a sliding-window technique, however, memory limitations due to a shortage of block rams required external storage to render the depth and pixel buffer, preventing the creation of a full pixel buffer that could be rendered to an HDMI output. The Zynq-7000's capability to pull this off at any acceptable frame-rate is questionable at best.

2.4.4 3D Rendering Results

While the renderer did not make it to the point of including the Game of Life itself, it is still worth including in this report. The rendering engine was going to be made synthesizable, however the device memory and area limitations as well as time constraints prevented this endeavor. The engine currently runs purely in low-level SystemVerilog code built from scratch and includes no outside code, either copied or imported from any external source. The rendering engine code and example video can be found at the second GitHub reference at the end of this report.

Figure 2.2: Single example frame generated by 3D renderer



3 Design Details

The implemented Game of Life design required a primary finite state machine and a second control flip-flop working in tandem. The FSM controlled the step and update of the simulation process, updating the grid of each generation to the next generation after a period of time. The flip-flop activates only during control mode and handles button input that controls the cursor and cell toggling features. Only the FSM or the flip-flop can be active at once because both modules update the state of the grid, with the active module overwriting the grid for the other.

3.0.1 Types

A *types.sv* file holds two packed structs which define the input and output of the system. Each data structure is responsible for strictly input to or output from the top level gameoflife module so input and output can be cleanly separated and transferred in two separate structures.

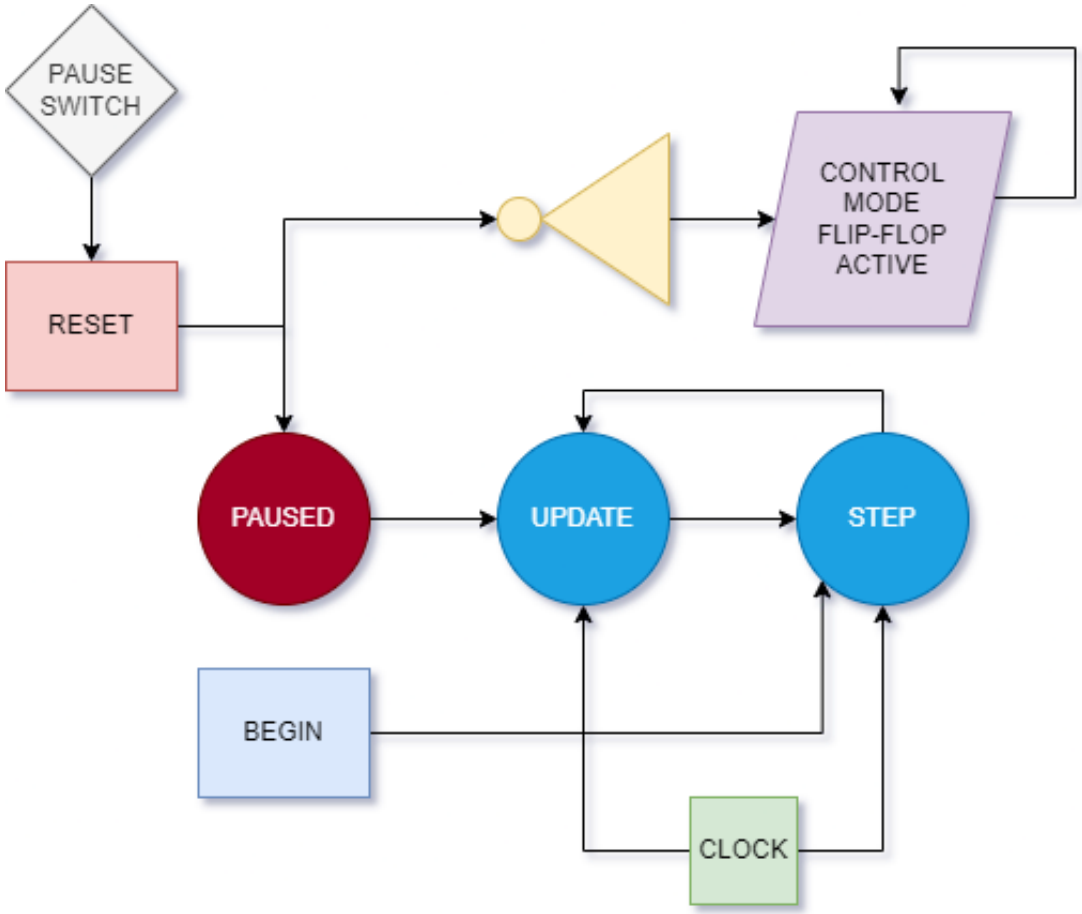
- The **golmachine** struct defines the output structure of the system: the grid, evolved grid, generation count, state, and nextstate.
- The **golcontrols** struct defines the input structure of the system: the pause switch, left, right, up, down, and toggle buttons, and the reset switch.

3.0.2 Simulation FSM

The FSM in charge of running the simulation when unpaused contains three simple states:

- **PAUSED** is the reset state and is only activated when the pause switch is on. The next step is **UPDATE** when unpaused.
- **STEP** updates the current grid to the evolved next-generation grid and increments the generation number by one. The next state is always **UPDATE** unless paused.
- **UPDATE** toggles the update signal which tells the testbench to output the current state of the game as either an image and/or a text-based grid in the output window. The next state is always **STEP** unless paused.

Figure 3.1: Schematic of FSM and flip-flop design



3.0.3 Control Mode

Control mode is controlled by a single flip-flop that updates the control state when the pause switch is active. It contains a set of multiplexers and conditional combinational logic that moves the invisible cursor and toggles cells between alive and dead when corresponding button input is detected.

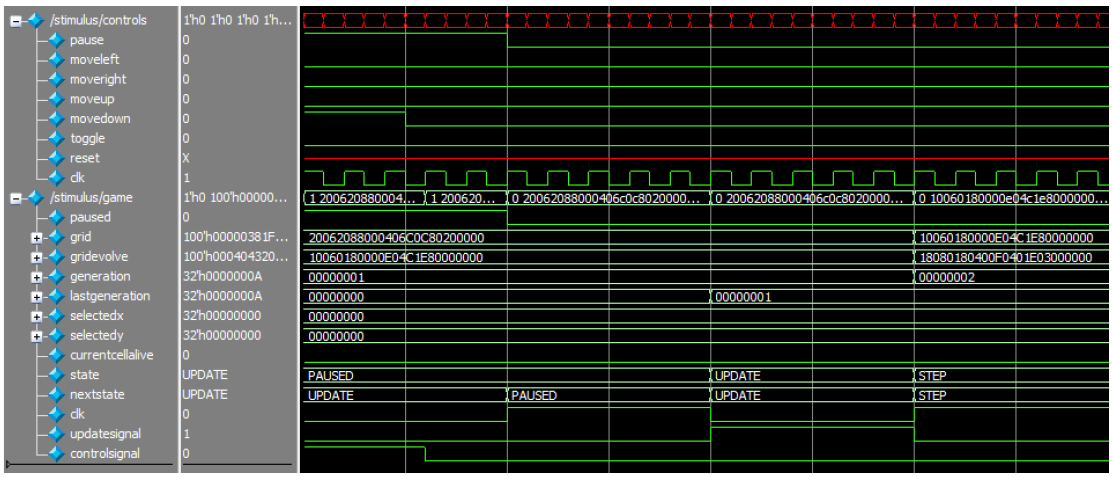
4 Testing Strategy

A more complex system requires a more complex testing strategy. The strategy involves multiple types and methods of testing, including a waveform for debugging, raw text grid output to the command window to test the Game of Life simulation, and an image processor to more easily visualize how the simulation will look each frame once rendered. The test setup that was created for the following test outputs is configured to use a 10x10 grid instead of the original 8x8.

4.0.1 Waveform

The waveform included only two packed structures: golcontrols and golmachine, displayed as controls and game respectively. Packing all the input and output into two separate structures allows the waveform to include new values as they are added while keeping clutter out. Only the most necessary values are depicted in the waveform. Within the output structure – listed as 'game' on the waveform – the most important values used to debug were state, nextstate, grid, gridevolve, pause, and updatesignal. Each of these values are used internally to move the simulation to the next generation. The controls structure includes all user input to the top-level gameoflife module. This includes the pause switch, up, down, left, right, and toggle buttons, and reset switch. The controls structure also includes the 125MHz system clock.

Figure 4.1: Waveform



4.0.2 Raw Text Grid Output

Testing custom grid setups calls for a better method of visualizing the grid layout. The solution was to output each generation as a set of 0s and 1s on a grid, with 0 representing a dead cell and 1 representing an alive cell. The test output is formatted in a way that is simple to verify.

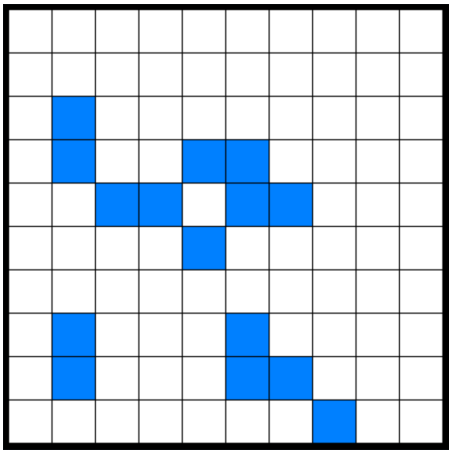
Figure 4.2: Text output for test generation 1

```
# 0  0  0  0  0  0  0  0  0  0
# 0  0  0  0  0  0  0  0  0  0
# 0  1  0  0  0  0  0  0  0  0
# 0  1  0  0  1  1  0  0  0  0
# 0  0  1  1  0  1  1  0  0  0
# 0  0  0  0  1  0  0  0  0  0
# 0  0  0  0  0  0  0  0  0  0
# 0  1  0  0  0  1  0  0  0  0
# 0  1  0  0  0  1  1  0  0  0
# 0  0  0  0  0  0  0  1  0  0
# ----------------------------------------------------------
# Paused: false;        Generation:          1;
# ----------------------------------------------------------
```

4.0.3 Image Output

While text output is easy to read and verify, it is boring and not aesthetically pleasing, so it was clear that a custom renderer to render each generation to an image file was absolutely necessary. An example of the output render for the same test generation as the last image is shown below.

Figure 4.3: Image output for test generation 1



5 Evaluation

5.0.1 Performance and Area

The final design implements the full evolution datapath of the Game of Life in parallel so it can be updated in a single clock cycle. However, the area requirements increase exponentially as the size of the grid gets larger. At some point, the FPGA will run out of space for combinational logic and reach a maximum grid size. At that point, to continue creating larger and larger grids would require multiple clock cycles that each compute a section of the larger whole. The performance could be maximized by creating the largest possible parallel blocks and applying them over multiple clock cycles for exceptionally large grids. For the implementation, the performance and area were nearly optimal for the purposes, as no extra area was required for other functionality so the full power of the FPGA could be dedicated to the Game of Life.

5.0.2 Power Consumption

As a whole, the Game of Life implementation uses only one FSM and flip-flop block to run both the simulation and control mode. Switching between them is accomplished by flipping one single bit. The state of the grid, however, is copied each clock cycle to the inactive module, which is detrimental to power consumption. Optimizing the system to only copy the state of the grid between simulation and control mode when they are switched would be by far the greatest power consumption improvement possible. Alternatively, going even further to change the underlying finite state machine system, it would be possible to combine both the simulation FSM and control mode flip flop into one module that requires no copies of the grid that need to be shared and updated between functions. This would theoretically be the most optimal version of the design.

5.0.3 Vivado Implementation Details

The design requires six buttons, but only four buttons are available on the fpga board, so the toggle and reset buttons are mapped to switches. Both switches behave as buttons, where flipping them on and off would be equivalent to a button press. Furthermore, for control mode to be active, the pause switch must be flipped on. Attempting to change the state of the grid while the simulation is active will do nothing. Within Vivado, the controls were mapped to the following:

- **Pause** is switch 7 (on is paused, off is unpaused)
- **Toggle** is switch 6 (behaves as a button)

- **Reset** is switch 0 (behaves as a button)
- **Cursor Down** is button 3
- **Cursor Up** is button 2
- **Cursor Right** is button 1
- **Cursor Left** is button 0

5.0.4 Summary

The final design includes a working Game of Life implementation that was synthesized and run on a Zynq-7000 series FPGA board. The features include both simulation mode and control mode. The datapath can be size configured at compile time, and the testbench suite has multiple output modes used to test new features and debug.

6 References

[1] Game of Life GitHub Repository

https://github.com/GarrettTP/fpga-game-of-life/

[2] 3D Renderer GitHub Repository

https://github.com/GarrettTP/system-verilog-3d-renderer/