Introduction and Explanation of Functionality
Emergent phenomena occur when a standard set of simple instructions are followed a large number of independent actors interacting within an environment. Random initial conditions result in unique outcomes, but similar behaviors may be evident as the system progresses. This project seeks to examine such phenomena by implementing and expanding on John Conway's historical computing construct 'the game of life'. While the original game involves only one species and a fixed instruction set, the system described here includes two species with adjustable properties. The game is implemented on a Xilinx Spartan-3 FPGA on a Digilent board, with output of a 80 x 478 two-color grid to a 640 x 480 VGA monitor.

'Game of Life' standard rules:
For a populated space:
< 2 neighbors: cell dies
2 or 3 neighbors: cell survives
> 3 neighbors: cell dies
For an empty space:
3 neighbors (exactly): cell is born

The system includes two species, green trees and blue tree squirrels. The green trees follow the standard 'game of life' instructions, with the exception of the addition of adjustability to the number of bordering cells required to give birth to a new cell (from 1 to 4). Too few trees, and they are exposed to harsh winds. Too many trees and they block out the light. The blue squirrels follow a new set of rules based on both tree and squirrel populations. Squirrels need to be near a tree to survive, and need to be in pairs and near enough trees to multiply and raise their young.

Green tree rules:
For a populated space:
< 2 neighbors: cell dies
2 or 3 neighbors: cell survives
> 3 neighbors: cell dies
For an empty space:
(adjustable from 1 to 4) neighbors: cell is born

Blue tree squirrel rules:
For a populated space:
Any tree neighbors: cell survives
No tree neighbors: cell dies
For an empty space:
2 or more squirrel neighbors + (adjustable from 1 to 4) tree neighbors: cell is born

The edges of the active grid can be considered as hard walls or cliffs. No cells can be born beyond these borders, and therefore nothing beyond the borders contributes to a cell's neighbor count.

Instructions for Use
Upon startup, the system loads a fresh grid of random green and blue pixels to the active 80 x 478 grid visible on-screen. All output is displayed on-screen. Inputs may be adjusted on-the-fly. Controls are the following:

Button 2: Refreshes the grid (at any time)
Button 0: Activates fast forward (maximum speed) if being played
Switches 7-6: Control blue squirrel mating tree requirements (1 + input)
Switches 5-4: Control green tree seedling requirements (1 + input). default: "10"
Switch 0: Activates game play to calculate future generations

Overall Architecture
The system uses the wishbone databus and provided interconnect (wb_intercon.vhd), SRAM memory (sramctl.vhd) and VGA display (wb_vga640x480.vhd) modules. New functionality is contained a single module (wb_game.vhd). This module directly processes button and switch control inputs and interfaces with the SRAM via the wishbone bus to update the video memory.

Diagram of Wishbone Modules

Description of Module Function
The Game Controller module is comprised of an SRAM interfacing wishbone master front-end, with a semi-parallel back-end processor, both contained in one large state machine.
Upon reset, the module writes zeros to all video memory addresses. Then it reads random data values from consequetive memory addresses and ouputs data to the green and blue components of the active grid pixels. Upon repeated hard resets (btn3) or grid updates (btn2), consequetive memory addresses are loaded for random data. The system then waits for the inter-generation delay time and begins to process data. The SRAM master grabs lines from data, places this data into temporary input caches for green and blue species, waits for data processing, then writes output lines from the cached green and blue output rows. Three horizontal rows are cached at a time for green and three for blue. The back-end processor calculates the number of active green and blue neighbors for each cell in parallel across each horizontal row from the display, considering the previous, current and next row from the caches. When the next generation for the row has been calculated, the caches are rotated, and the system again waits for the inter-generational delay to pass, before the cycle repeats.

State Machine Flow

Summary of Design Challenges and Problems
The central design focus was to determine how to best implement the system within the limited space of the 200k Spartan-3 FPGA. It became clear early that a fully parallel implementation would severly limit grid size, since each pixel requires several adders and comparitors. A serial implementation would be slow and cumbersome or require a buffer, given the repeated interfacing to the SRAM. Because the data is stored as horizontal lines, it makes sense to process data in semi-parallel by row. All of the lines of a row are loaded into memory and compared in parallel to the preceding and following rows. Even with this architecture, the process is limited to 80 horizontal pixels. Moving beyond 72 horizontal pixels required increasing the synthesis and mapping effort levels and moving some look-up-tables to on-board bram to clear up space.

Properly formatting the output values to update the SRAM for each following generation was also challenging. While the pixels in each line are arranged from high to low, lines are displayed in reverse on screen from how they are represented in the literature. In the celebration that the system was providing output, and within the randomness of the generated patterns, this fault was not immediately apparent. Looking closer, it was clear that the columns were reversed and shifted. Troubleshooting included outputing binary incremented value patterns across each row to test for patterns. One the lines were reoriented, it was clear that cells on one side of each column had been shifted to the other side of the previous column. After scouring the code, this was initially remedied by changing the offset value within the provided VGA module. I assumed that the shift had not been recognized when displaying characters, due to the unused space between them. The real reason become clear after fixing the next problem.
Over longer time scales as populations reduced, it became apparent that new cells were being born from the void on the left side of the grid. This problem was due to a misalignment of the caches, so that they were offset to one side, rather than centered. Each cache is created with an extra space to be left blank on either side of the space reserved to record the grid pixels. This allows the cache data of the leftmost and rightmost cells to be processed by the neighbors process without modification. While these were shifted (by not adding +1) to the cache value, errant bits were being introduced into the process.
Finally, there were some challenges when attempting to write new rules for the interaction between species. It was difficult to modify the rules without catastrophic effects to either or both species. I decided to leave the trees independant of squirrels, and the squirrels reliant on trees, and themselves for reproduction. Instead of drastically modifying the algorithms, I chose to introduce user adjustability of one value for each species. This allows interaction with the model, while preserving species for extended time periods in most cases.

Reported Specifications as Synthesized and Routed:

Timing Summary:
- Minimum period: 13.617ns (Maximum Frequency: 73.438MHz)
- Minimum input arrival time before clock: 8.763ns
- Maximum output required time after clock: 28.788ns

Logic Utilization:

Number of Slice Flip Flops:	1,749 out of	3,840	45%
Number of 4 input LUTs:	3,524 out of	3,840	91%

Logic Distribution:

Number of occupied Slices:	1,863 out of	1,920	97%
Total Number of 4 input LUTs:	3,693 out of	3,840	96%
Number used as logic:	3,524		
Number used as a route-thru:	169		
Number of bonded IOBs:	97 out of	173	56%
Number of RAMB16s:	12 out of	12	100%
Number of MULT18X18s:	1 out of	12	8%
Number of BUFGMUXs:	2 out of	8	25%