ECE532: Conway’s Game of Life

Final Design Report

Noah Poplove
Zhe (Ryan) Yin
Kingston (Ting Ray) Wang

Table of Contents

1. Overview

1.1 Background

Conway's Game of Life is a 0-player cellular automata simulation game. The game consists of a set of cells in a two-dimensional grid of squares, where at each time step or "generation", the cells are born, continue to live, or die, based on certain conditions. An initial state is set by the player, who can then observe how it evolves over time based on the following rules:

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.

The Game of Life is known for generating shapes and patterns out of random initial states. Some patterns are simple, while some states have been identified which consist of up to millions of cells and act as logic gates (AND, OR, etc.) or even replicate themselves after a certain number of generations.

1.2 Motivation

Our team is interested in how a two-dimensional grid and a few simple rules can result in random squares turning into symmetry and the range of complexity that patterns can take. Many software implementations of Game of Life can be found today, especially since its straightforward rules make it an accessible project for beginner/intermediate programmers. Our team was able to find some hardware implementations of the Game of Life, however they were all limited to one FPGA board, limiting the size of the grid and the computing power to one board.

Given that the game's grid can be infinitely large, we wanted to design an implementation of Conway's Game of Life that is scalable to multiple FPGAs that work together to compute each generation for large grids. The idea of splitting a large grid into multiple smaller grids would also act as proof of concept for any game where a large grid/map must be split up efficiently, particularly in MMORPGs.

1.3 Goals

At the start of our project, our goal was to implement the Game of Life in hardware such that it was scalable to multiple FPGAs and could simulate large grids (more than 100x100). We planned to use one PC/DESL machine and two or more FPGAs. The DESL would send data to one FPGA through UART, and that FPGA would communicate with the other FPGAs over the DESL lab's shared network.

Specifically, our goal was to split up the Game of Life grid evenly between multiple FPGAs, where each FPGA computes its grid's cell states within the game and then communicate shared edge cell states over the network (example in Figure 1).

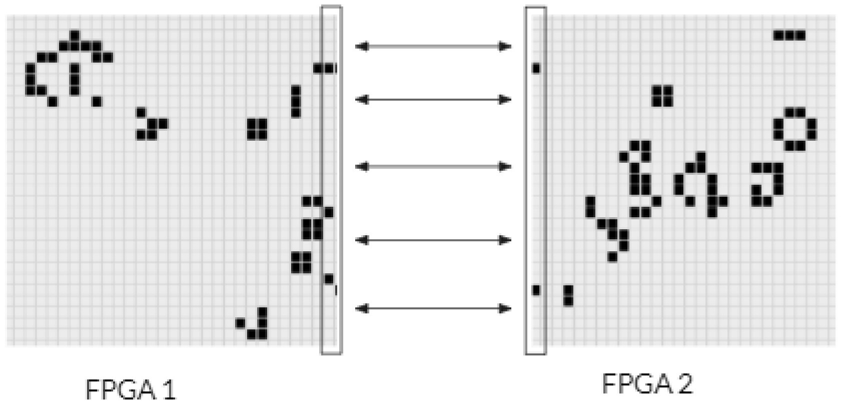


Figure 1: Illustration of two FPGAs communicating their edge states

Originally, we intended to implement each grid section in an FPGA's BRAM. Our hardware would iterate through each cell in BRAM and compute its next state based on surrounding cells. Other important aspects that were part of our goals were:

- Ability to start/pause the game and input grid states from a DESL machine
- Having a visual output of the grids to show to live evolution of the game

We had some secondary goals that we planned to attempt if we completed our main goals, but unfortunately we were not able to implement them. These goals were:

- Having a VGA interface as a GUI that could display the grid
- Functionality to store/load grid states to a microSD card

1.4 Block Diagram

A high-level block diagram of the system is shown in Figure 2. Our original proposed block diagram can be found in Appendix A (this includes BRAM blocks and microSD card modules). A more detailed block diagram of the hardware in each individual FPGA is shown in Figure 3. The functionality of each block is described in the later sections.

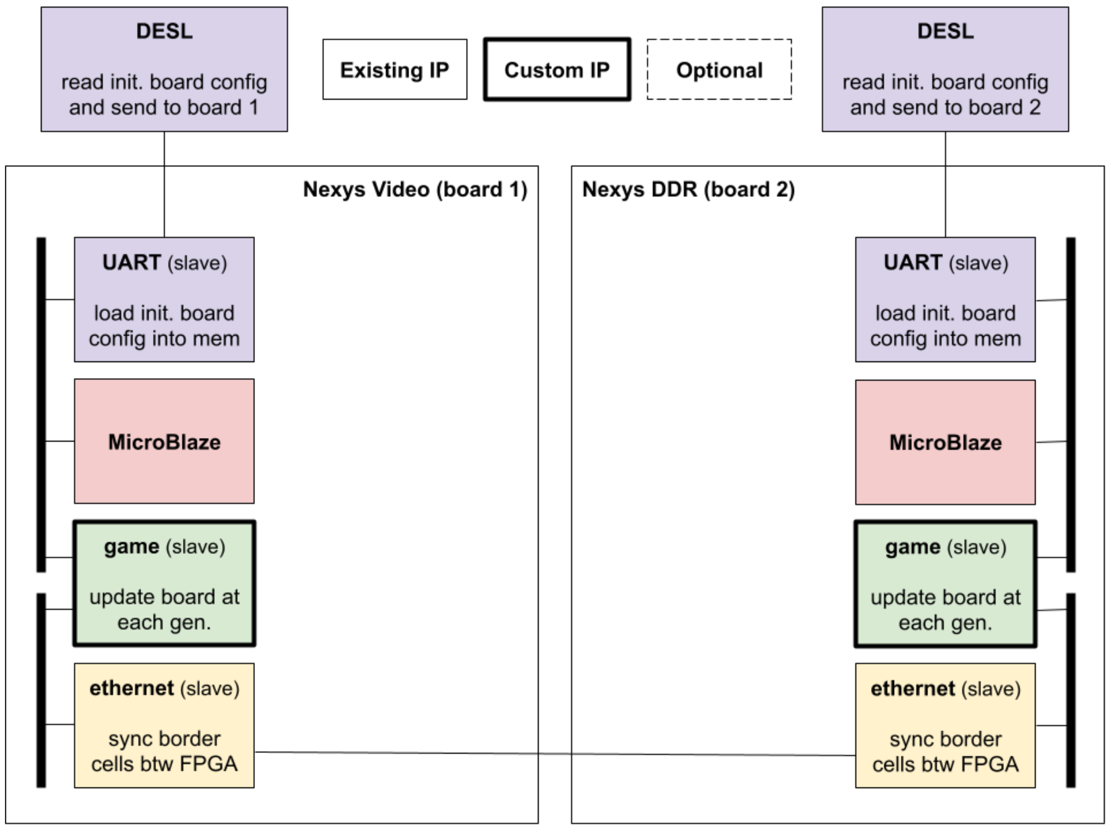

DESL
read init. board config and send to board 1
Existing IP
Custom IP
Optional
DESL
read init. board config and send to board 2
Nexys Video (board 1)
Nexys DDR (board 2)
UART (slave)
load init. board config into mem
MicroBlaze
game (slave)
update board at each gen.
ethernet (slave)
sync border cells btw FPGA
UART (slave)
load init. board config into mem
MicroBlaze
game (slave)
update board at each gen.
ethernet (slave)
sync border cells btw FPGA

Figure 2. Overall Block Diagram

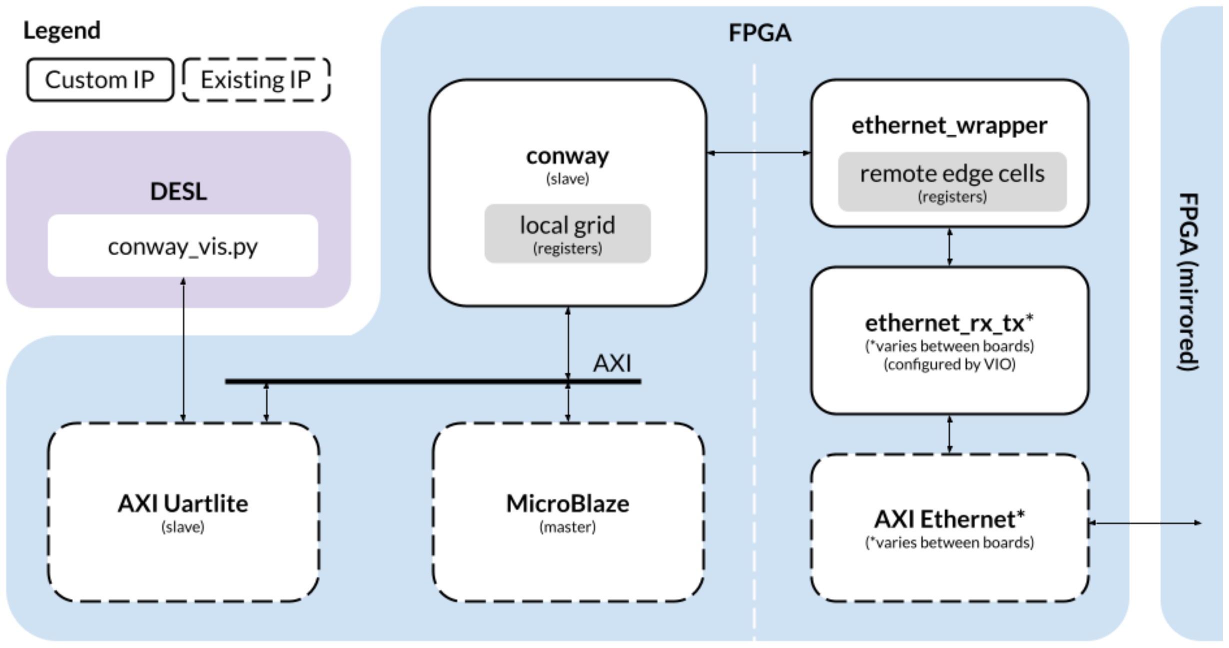


Figure 3. Detailed Block Diagram for a single FPGA

1.5 Brief Description of IP

Module	**Function**	**Type**	**IP**
Microblaze	Multi-purpose controller that controls the system	Hardware	Existing
UARTLite	Interface for sending data from MicroBlaze to the UART port via serial-to-USB cable	Hardware	Existing
System Controller	C program that runs on Microblaze to control the entire system (change FSM state, send grid input, receive grid output) and communicate through UART with the interfacing software	Software	New
Interface Program	Python script to communicate with System Controller through UART. This visualizes the output grid on the DESL machine.	Software	New
Game of Life	Computes Game of Life cell states. Sends/receives grid data from Microblaze and sends/receives edge cell data from ethernet module	Hardware	New
ethernet _rx_tx	Receives and transmits ethernet frames; interfaces with AXI Ethernet* IPs and ethernet_wrapper	Hardware	New
ethernet _wrapper	Receives and transmits remote and local edge cells between FPGAs; interfaces with ethernet_rx_tx	Hardware	New

2. Outcome

2.1 Results

We encountered difficulties in some key parts of our project which resulted in us having less time to integrate the parts of our design with each other. We were able to complete each individual part of our design and meet their respective acceptance criterias (See Appendix B). These completed design aspects include: the Game of Life game logic, FPGA-FPGA communication of edge states, and live visualization of the grid.

Due to time constraints from the aforementioned issues (described in more detail in Section 4), we were unsuccessful in integrating these parts into a complete design. We effectively ended with two separate designs, capable of the following:

1. A design that uses two FPGAs running the Game of Life and communicating their edge cells to each other using our ethernet modules. Displays output to Xilinx SDK terminal
2. A design that uses one FPGA running the Game of Life. Communicates with Python interface script to visualize live grid states in a new window on the PC.

Both versions of our design are capable of receiving user input of the initial grid and displaying some form of the grid's evolution over time. The Python visualization can be seen in Figure 4.

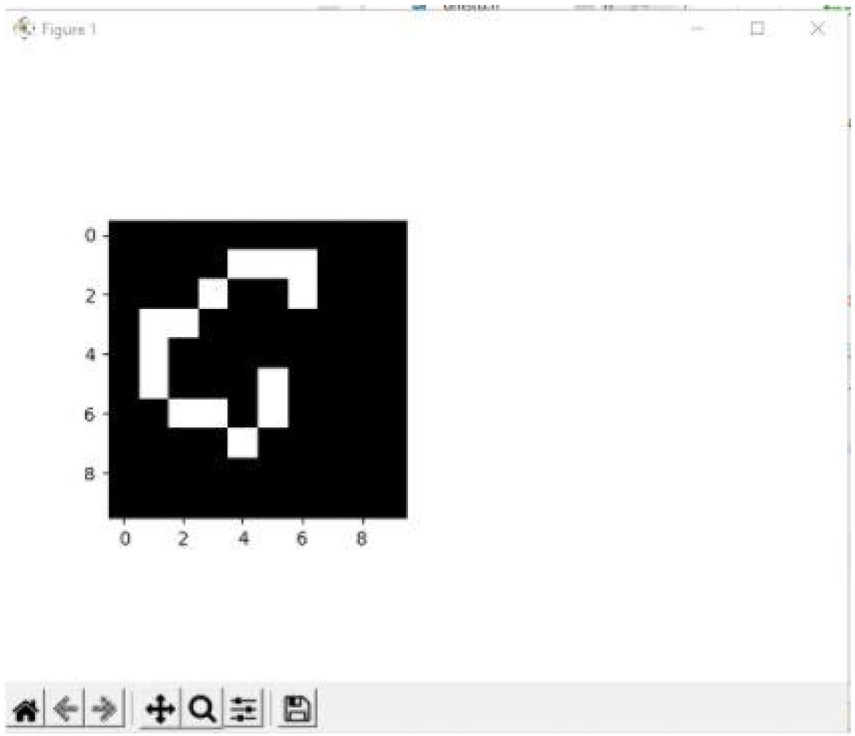


Figure 4. Visualization of Conway's Game of Life

Our Game of Life logic works as intended, however we did not end up implementing a large or customizable board size. Our hardware currently uses a 10x10 board - this was what we originally intended on using in order to easily verify the correctness of its functionality.

Our FPGA-FPGA communication works as intended and allows the FPGAs to communicate their edge states and compute each state correctly. Our ethernet modules include the necessary synchronization that allows the two boards to compute each grid state at the correct time.

2.2 Future Improvements

The first and most important future improvement would be to complete the integration of our modules into one complete design. Beyond this, there are a few potential future improvements that we have considered:

- Add functionality for 3+ FPGAs
    - This would require expanding the synchronization logic to account for each board in use. Some of this has been implemented in ethernet_wrapper.
- Customizable board size and game running speed
    - The main bottleneck in our design is the FPGA-FPGA communication over the ethernet. The retransmission timeout feature, as it's currently implemented in our ethernet modules, are generous in that frames are not re-transmitted after 131,072 cycles even though network round-trip latency is much shorter, at roughly 7000 cycles. The timeout (and also the total number of shared edge cells) could be fine-tuned to increase game running speed.
- VGA GUI or improved Python GUI

3. Project Schedule

#	Proposed Milestone	Weekly Accomplishments
1	Create testbench code (for testing custom hardware correctness)	Mainly worked on Project Proposal Report. Wrote our behavioral testbench code and simple visualization code to display cell states to the TCL console.
2	Implement custom hardware on one FPGA to compute Game of Life progression DESL should be able to communicate with FPGA using UART + microblaze + stdin to send initial conditions	Started hardware implementation of Game of Life logic. Wrote some functions for the system controller module to communicate through microblaze using memory-mapped registers.
3	Implement connection between FPGAs with TCP/UDP with microblaze/LWIP DESL should send and receive grid states and display the grid on the terminal	Completed logic for Game of Life (parallel implementation). Started making progress on ethernet modules for FPGA-FPGA communication (RX and TX implemented)
4	Implement connection between DESL and FPGAs using UART without microblaze Implement connection between FPGAs without network (directly through hardware)	Added FSM to Game of Life module for control of game functionality and stepping through the game one generation at a time. Mostly completed C program for Microblaze system controller, which can now send/receive grid states and display to SDK console. RX and TX unified.

5	Implement GUI using VGA	Tested and debugged FPGA-FPGA communication. Started integrating FPGA-FPGA communication modules with the Game of Life module.
6	Add functionality to store and load grid states from microSD card	Made progress integrating modules together. Made progress in integrating the FPGA-FPGA networking functionality with the game itself. Added functions to Microblaze controller C code for better displaying grid states, sending/receiving data from Game of Life module.
-	N/A (week from Milestone 6 and Demo)	Created visual interfacing Python script that communicates with Microblaze controller through UART. Modified Microblaze controller C code for UART connection with visualizer script.

Compared to the proposed milestones, our weekly accomplishments were behind schedule starting from Milestone 2. We underestimated the work that would be required for building and debugging both the game's IP block as well as the FPGA-FPGA communication. Integrating our parts together took especially long, which we believe is partially due to each of us working remotely due to Covid-19 this year, resulting in us being in different time zones.

One important delay was that we implemented UART connection after Milestone 6 when it was planned to be done for Milestones 2 and 4. This was because we incorrectly thought that interacting with our design through the Vivado SDK terminal was sufficient as a UART implementation.

4. Description of IP Blocks

4.1 MicroBlaze

The main function of the microblaze processor was to facilitate communication between the Conway block and our DESL interface for sending/receiving grid states and control signals. Our C program for controlling the Microblaze processor used it to read and write to memory-mapped registers corresponding to the Conway block.

4.2 Conway

Note: this module was previously named Game_of_Life.

This module contains main FSM datapaths, the most important of which is the logic for the Game of Life itself. FSM states are:

1. Reset - set board cells all to 0 (dead)
2. Write - write new grid values sent from DESL
3. Read - output the grid through to DESL
4. Step - run computations for one generation of the game
5. Run - run computations indefinitely

Our implementation of the logic for each cell was changed from what we initially planned. Firstly, instead of using BRAM to hold cell states on each FPGA, the cell states are stored in arrays of registers where each cell state takes up one bit of a register. The main reason for this approach is that we did not end up using larger boards, in which case it would have likely been more efficient to use RAM to hold the grid values. The generated RTL for our Conway block using a 10x10 grid can be seen in Figure 5. This design uses significantly more resources than an iterative approach and limits the board size that we would be able to use.

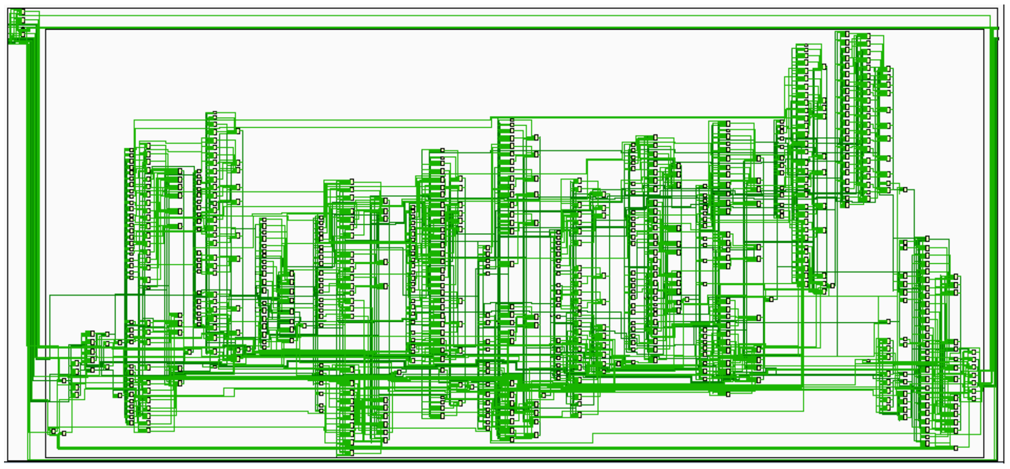


Figure 5. Conway block RTL design

In terms of the computation logic, instead of iterating through the grid (like in all software approaches and many hardware approaches), our module generates the logic for every cell so that all of their “next states” are computed at the same time. We recognize that this approach has benefits and drawbacks. On the plus side, computation of each generation takes significantly less time. However, as we increase the size of the grid, we would start using a large amount of the FPGA’s resources/LUTs. While our parallel implementation works well on one FPGA, it is likely not necessary when using multiple FPGAs due to the time delay associated with communicating edge values across the network. If given the chance to change our approach to this section of the Conway module, we would implement the cell states in BRAM and compute next-state values using some combination of parallel and series logic. For example, it could generate logic that computes just a 5x5 chunk of the grid in parallel, then iterate through the whole grid in 5x5 sections.

4.3 Ethernet

The FPGA-FPGA communication through Ethernet is implemented in two hardware modules:

1. **ethernet_rx_tx**:
    a. receives and transmits ethernet frames
    b. interfaces with AXI Ethernet* IPs and ethernet_wrapper
2. **ethernet_wrapper**:
    a. receives and transmits remote and local edge cells between FPGAs
    b. interfaces with ethernet_rx_tx

4.3.1 ethernet_rx_tx

Note: this module was previously named ethernet_tx_hub.

This module, together with AXI Ethernet* IPs, implements the physical link layer of the networking stack. Implementation differs between Nexys DDR and Video boards:

- **Version 1** implements the programming sequence for AXI Ethernet Lite MAC detailed in p. 27-31 of the corresponding Product Guide [2], specifically the following sections:
    - Transmit Interface – Software Sequence for Transmit with Ping Buffer
    - Receive Interface – Software Sequence for Receive with Ping Buffer
    - Ethernet MAC Address
- **Version 2** implements the programming sequence for AXI4-Stream FIFO detailed in p. 38-39 of the corresponding Product Guide [1], specifically the following sections:
    - Table 3-1: Programming Sequence for TX and RX in Store-and-Forward Mode

In both versions of this module, an FSM:

- Resets or powers-up the AXI Ethernet* IPs:
    a. **Version 1 only**: program the correct ethernet MAC address
    b. **Version 2 only**: enable transmit and receive complete interrupts
- Waits for incoming or outgoing frames to arrive:
    - If transmit valid from ethernet_wrapper goes high, transmit a frame. ethernet_wrapper provides the destination address, the length of the frame and the payload.
    - If receive status bit (Version 1) or receive complete interrupt (Version 2) from goes high, receive a frame. ethernet_wrapper is provided with the source address and the payload. Frames with destination addresses not matching the device's own address are not passed along.

This module can block ethernet_wrapper but can't be blocked by it. The interface between this module and ethernet_wrapper has both valid and ready for transmit, but only valid for receive.

4.3.2 ethernet_wrapper

Note: this module was previously named ethernet_adapter.

This module implements the network layer of the networking stack. The module, using interfaces provided by ethernet_rx_tx:

- Transmits local edge cells (**reliably**) to neighbouring FPGAs
- Receives remote edge cells from neighbouring FPGAs

The payload in each ethernet frame transmitted or received contains:

1. The data - edge cells
2. The type - poll or ack
3. The sequence number

The module implements retransmission timeout to re-send frames not acknowledged after a set number of cycles. It decides, based on the type and sequence number of incoming frames, whether to accept or reject (not acknowledge) a frame and whether to update remote edge cell registers. Duplicate frames are dropped and neighbouring FPGAs could be stalled.

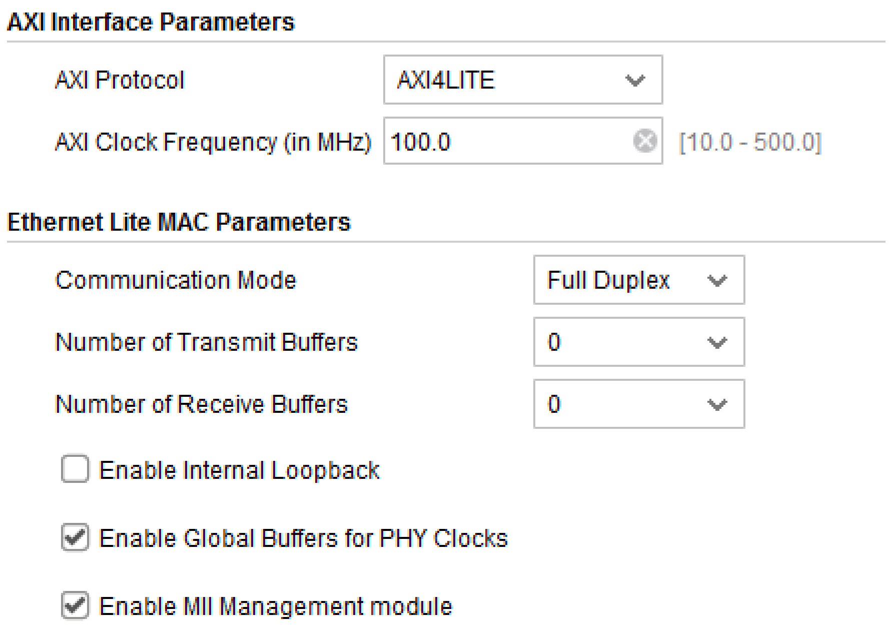
Hence, during each round of synchronization:

1. The sequence number is incremented:
    a. The module updates remote edge cells upon receiving the first frame with the same sequence number as its own sequence number.
    b. The module acknowledges any frame with a sequence number smaller than or equal to its own sequence number.
1. local edge cells are transmitted reliably
2. The sequence number is incremented again:
    a. The module acknowledges any frame with a sequence number smaller than its own sequence number.

The addresses of neighbouring FPGAs are programmed through the VIO.

4.3.3 AXI Ethernet Lite MAC

This pre-existing IP core implements Ethernet MAC on the Nexys DDR board. It has memory to buffer packets, and it provides receive and transmit interrupts. “Number of Transmit Buffers” and “Number of Receive Buffers” are both set to 0.



AXI Interface Parameters

AXI Protocol: AXI4LITE

AXI Clock Frequency (in MHz): 100.0 [10.0 - 500.0]

Ethernet Lite MAC Parameters

Communication Mode: Full Duplex

Number of Transmit Buffers: 0

Number of Receive Buffers: 0

☐ Enable Internal Loopback

☑ Enable Global Buffers for PHY Clocks

☑ Enable MII Management module

Figure 6. Customize Block window for AXI Ethernet Lite MAC

4.3.4 AXI 1G/2.5G Ethernet Subsystem

This pre-existing IP core implements Ethernet MAC on the Nexys Video board.

4.3.5 AXI4-Stream FIFO

This pre-existing IP core interfaces with AXI 1G/2.5G Ethernet Subsystem to allow the write or read of packets written and read using the AXI4-Lite interface. It provides interrupts for error and status conditions.

4.4 System Controller

This module is a C program that runs on Microblaze and controls the design through the FSM within the Conway module, as well as interacting with the Python interface controller through a UART serial channel. It also visualizes grid output through the SDK terminal.

The Microblaze module interacts with the Conway block through memory-mapped registers, so this controller takes input through the SDK terminal and controls the game by writing to the Conway-mapped addresses in memory. We wrote a number of functions to do certain actions by writing to the appropriate addresses, including:

1. write_row(): overwrites the values of one row of the grid
2. write_grid(): overwrites the entire grid with the inputted values
3. write_edge(): writes values to the register(s) holding the edge value (for testing)
4. run_step(): write to address that changes FSM in order to run one generation of computations
5. print_grid() and print_grid_hex(): prints grid to SDK terminal in either binary or hex, respectively
6. send_uart(): send values to the Python interface program through UART
7. rec_uart(): receive input values through Python interface program through UART

4.5 Interface Program

This module is a Python script that takes input from the user to run/pause and set grid values, as well as visualize the live grid in a separate graph window. The script takes inputs through the DESL console and interacts with our hardware through a UART serial channel.

For UART communication, we used Python's Serial library, which provided the necessary functions for read/write to the serial stream (as well as checking how many r/w bits were waiting, etc.). We used blind synchronization between the Microblaze system controller's C program and the interfacing script, where each module waits a set amount of time before reading from the serial channel. Ideally we would have implemented a more robust synchronization method for this UART connection, but we unfortunately were limited by time constraints and did not necessarily need that due to the relatively slow speed at which we expected to progress through the game (around 1 generation per second).

For visualization, we used Python's MatPlotLib library, which allows for plotting all sorts of graphs and numerical visualizations. The library's ImShow() function was used to plot a grid of black and white squares corresponding to array values.

5. Design Tree Description

G8_ConwaysGameOfLife

- **README.bd**
- **src** - Vivado project files
  - **conway_ddr** - the project targeting Nexys DDR boards
    - **bd** - block diagrams
      - **block_main.bd** - block diagram
    - **hdl** - Verilog files
      - **conway** - RTL for Game of Life logic
        - **Conway_v6_S00_AXI.v** - module definition for conway
      - **ethernet** - RTL for ethernet modules
        - **ethernet_adapter.v** - module definition for ethernet_wrapper
        - **ethernet_tx_hub.v** - module definition for ethernet_rx_tx
        - **m_axi_interface.v** - AXI master interface for ethernet_rx_tx
    - **sdk** - files for the MicroBlaze
      - **test_uart.c** - C code for controlling the progression of the game
  - **conway_video** - the project targeting Nexys Video boards (same structure as conway_ddr, but with different block_main.bd and ethernet_tx_hub.v)
- **pc** - files for the PC
  - **conway_vis.py** - Python code for UART connection and visualization on PC
- **doc** - reports and presentations
  - **final_report.pdf** - this report

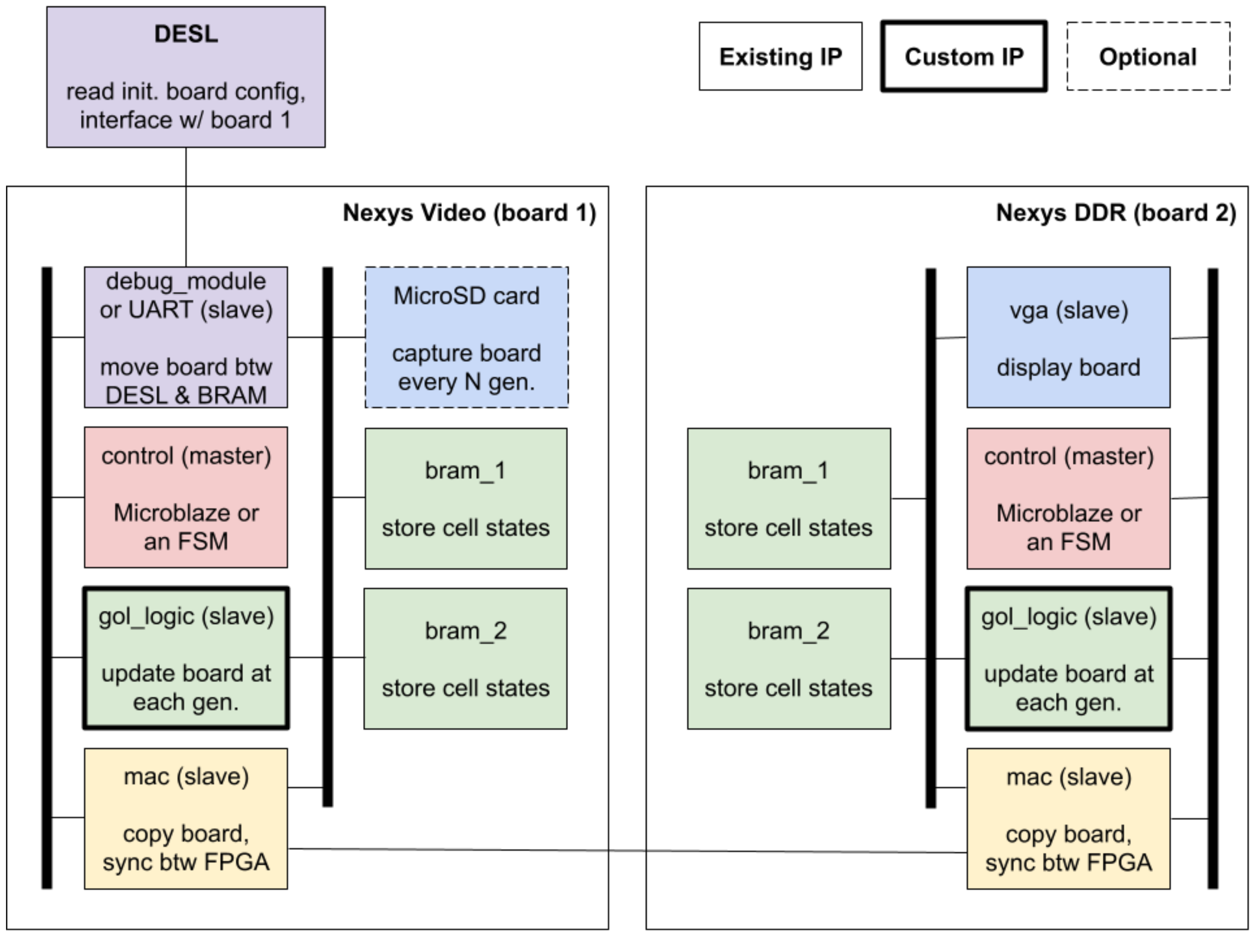
6. References

[1] AXI4-Stream FIFO v4.2 LogiCORE IP Product Guide
https://www.xilinx.com/support/documentation/ip_documentation/axi_fifo_mm_s/v4_2/pg080-axi-fifo-mm-s.pdf

[2] AXI Ethernet Lite MAC v3.0 LogiCORE IP Product Guide
https://www.xilinx.com/support/documentation/ip_documentation/axi_ethernetlite/v3_0/pg135-axi-ethernetlite.pdf

7. Appendices

Appendix A: Original Block Diagram



Appendix B: Original Functional Requirements

#	Functional Requirement	Acceptance Criteria
1	Correctly simulates a grid according to Conway's Game of Life rules	Behavioral simulation passes testbench by outputting matching data
2	Accepts user input	User is able to change the size and initial conditions of the grid (adding/removing squares and patterns), specified through a file

3	Visual output of grid generations	A visual interface that shows the live evolution of the grid over time
4	**Constraint**: Utilizes resources from multiple FPGAs	Game grid is split between FPGAs to compute
5	**Constraint**: FPGAs communicate with each other	FPGAs send border cell data to each other for each generation