Imperial College London

Mathematics Accelerator - JABBAL

ELEC50015 – Electronics Design Project 2

17th June 2024

Submitted to Dr Edward Stott

by

Benjamin De Vos	02217428
Anderson Lo	02238821
Lolézio Viora Marquet	02217500
Ajay Samaranayake	02213324
Ching Bon Tang	02207717
Joshua To	02267753

Department of Electrical and Electronic Engineering

2nd Year EEE and EIE

Word Count: 9298

Abstract

Conway's Game of Life, a well-known cellular automaton consisting of an infinite size grid of "alive" and "dead" cells. These cells evolve based on the number of "alive" Moore neighbours they have, where the next state of each cell is determined by its current state and the state of its current 8 neighbours. Calculating the next state for large grid sizes is computationally expensive but can be parallelised well. Therefore, we present an implementation of Conway's Game of Life for a native resolution grid (1280x720), using custom FPGA logic to accelerate the next state calculations along with modern computer vision techniques to devise an inclusive and intuitive user interface (UI). Our system generates frames at 60Hz, with potential for a higher frame rate with better monitors. Moreover, users can input any initial grid of "alive" cells through simple hand gestures, and each current generation can be studied with an intuitive pause function.

Table of Contents

List of Tables

List of Figures

1. Introduction

The motivation for this project was “to create an educational tool that can visualise a mathematical function” [1]. A user must be able to directly interface with the visualisation of a mathematical function, which therefore should be calculated in real-time and shown in video format. The PYNQ Z1 FPGA Development Platform was provided as a hardware accelerator, for which custom logic is developed to interface with a user input and an output monitor.

1.1 Initial Mathematical Function Choice

The requirements for the mathematical function was that it must have educational applications, be visual in nature, interact intuitively with user inputs, and be easily parallelisable making use of hardware acceleration. Therefore, the following possible functions were identified in Table 1:

Table 1 - Proposed functions

Function	*Advantages*	*Disadvantages*
Julia Set Fractal for corresponding point on Mandelbrot Set Fractal	• Visually stunning. • Each pixel colour is independent of all others. • Infinitely recurring, intuitive user zoom.	• Commonly implemented. • Lacks educational value.
Photorealistic image with ray tracing	• Visually stunning. • Intuitive, videogame like user interface. • Real-world, educational use.	• Not easily parallelised. • Complicated real-world physics. • Requires graphic design skills.
Fourier Series Drawing	• Creates complex drawings. • Easily parallelised.	• Requires drawing skills for shapes. • More advanced user interface. • Little real-world educational use.
Collatz Conjecture	• High educational value. • Parallelable.	• Difficult visualisation. • Little user interface.
Conway's Game of Life (GoL)	• Easily measurable performance. • Visually stunning. • Educational with real-world applications. • Easily customisable. • Easy user interface. • Possibility to rush to simple implementation and scale up (AGILE).	• Potentially challenging to parallelise. • Requires extensive FPGA memory.

The list was narrowed down between GoL and Fractals, but GoL was eventually chosen due to its scalability, meaning that more advanced features could be implemented after a simple implementation was working, while we felt Fractals would be a more of an "all or nothing" type of project.

1.2 Conway's Game of Life (GoL)

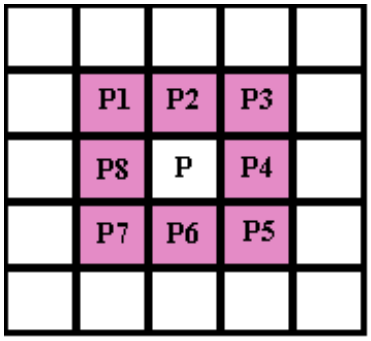
1.2.1 Rules

GoL consists of an infinitely sized grid, which we restrict to the native resolution of our output monitor, 1280 x 720 pixels, with one cell per pixel. The initial location of "alive" cells is known, and the next state of each cell is calculated as follows:

1. The number of "alive" Moore neighbours (See Figure
2. 1) of the cell is counted.
3. If the cell is "alive", the following rules are applied, which emulate under and overpopulation:
    a. If the cell has 2 or 3 "alive" neighbours, it survives in the next generation.
    b. Else, it becomes "dead" in the next generation.
4. If the cell is "dead", the following rules are applied, which emulate reproduction:
    a. If the cell has exactly 3 "alive" neighbours, it becomes "alive" in the next generation.

Figure 1 - Moore neighbourhood [2]

    b. If not, it remains "dead" in the next generation.



    c.

1.2.2 Real-world Applicability and Educational Goals

Conway's Game of Life is a Turing complete simulation, meaning that all logic gates required to build a computer can be simulated by alive and dead cells [3]. However, its applications extend beyond the realms of computer science.

In particular, the reproduction/overpopulation/underpopulation mechanism can be used to show population growth of organisms in an ecosystem, bacteria, or cancer cells. For example, a modified version was able to accurately describe the predator-prey cycle system commonly seen in hare and fox habitats [4].

While the applications are potentially limitless, generating a large grid is computationally expensive and may require access to large, expensive CPU resources. Our goal is,

therefore, in alignment with Imperial College London's goal of excellence in science and engineering for the benefit of society [5], is achieved through the use of the inexpensive PYNQ-Z1 platform. This platform and our efficient implementation of GoL, as well as its remote interface, would mean that it could be deployed in regions of the world with little access to powerful computers, and help develop talents and solutions in regions lacking computing infrastructure.

1.3 System Overview

Our system consists of three separate pieces of hardware - a laptop, the PYNQ board and the monitor. A user interface runs on the laptop and allows the user to initialise the first generation. It also sends signals to the board for starting and stopping the visualisation. The FPGA performs calculations related to finding the next generation and is also responsible for generating the visualisation of the changing states via an HDMI out port connected to a monitor.

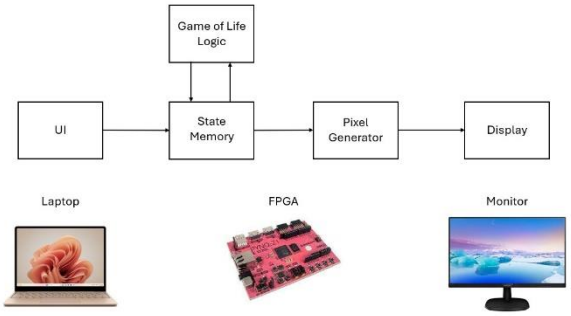


Figure 2 - Project Overview

1.4 The PYNQ-Z1 Field-Programmable Gate Array (FPGA)

The PYNQ-Z1 is a new, powerful hardware platform which gives us the possibility to exploit Xilinx Zynq All Programmable System-on-chip (APSoC) as it directly interfaces with an ARM A9 CPU capable of running a web server hosting a Jupyter Notebook [6]. This means the system's user Interface can directly communicate with this Web Server, and by extension, the programmable logic, which hosts our hardware accelerator.

The programmable logic (PL) comprises of [6]:

- 630KB of Block Random Access Memory (BRAM)
- 13,300 Logic Slices of 4 Look Up Tables (LUTs) each, giving 53,200 LUTs
- 17,400 LUTRAMs of Distributed Memory

Particularly, this limits the number of operations that can be carried out in parallel (since each uses some LUTs), the amount of data that can be stored in LUTRAMs, and the number of frames that can be stored inside the PL's Block RAM, which influences our implementation of our hardware accelerator on the platform.

2 Algorithm and parallelisation

2.1 Theory

The requirements of the project state that "the computational throughput of the accelerated implementation shall exceed that of a CPU-only alternative" [1].

Therefore, a purely software-based version capable of simulating GoL using Python and C++ must be created. This acts as a performance benchmark to compare our final solution against, on a single thread.

Parallelisation, on the other hand, consists of dividing a large task into multiple independent subtasks that are executed concurrently, by multiple threads [7]. As expected, this can achieve considerable speedup, potentially linearly to the number of threads if the task is "embarrassingly parallel".

How long an algorithm will take to run is known as its time complexity [8], denoted in Big O notation. Intuitively, how long an algorithm will take to run depends on how big of a problem it is trying to solve, denoted by the variable n. In the case of GoL, n can refer to the width or height of the grid.

The goal, therefore, of the algorithm development, is to have a solution work in as low of a power of n as possible, in $O(n)$ notation.

Various approaches can be taken to achieve this, and precise specifications must be devised to determine which approach is optimal for the implementation on the FPGA.

2.2 Design Specification

Precise specifications for both the CPU benchmark be designed based on the overall requirements of the project of being able to run at 60Hz at a native resolution of 1280x720.

Table 2 - Design specification for the CPU benchmark

Specification	Quantitative measurement	Test
Must be time efficient (not intentionally slow)	Must run in at least $O(n^2)$ time complexity.	Use analysis techniques to measure time complexity.
Must be memory efficient	Must run in at least $O(n^2)$ memory complexity.	Use analysis techniques to measure memory complexity.

The CPU benchmark must, therefore, be developed to meet the specifications outlined above.

2.3 Experimental Method and Development

The simplest, naive algorithm for GoL consists of storing the grid as a 2D array and iterating through each "alive" cell. For each cell, the number of "alive" neighbours is counted, and the next state is determined, as seen in Figure 3

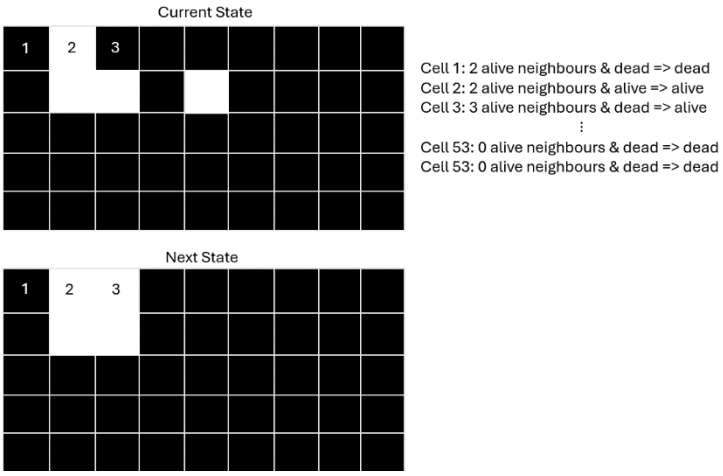


Figure 3 3 - Naive cell-by-cell algorithm

This algorithm has worse time complexity of $O(n^2)$ since it requires iterating through every cell, and then checking the values of 8 adjacent cells.

To solve this, looking at Figure 3 shows that it is inefficient to check for the cells in the last 2 rows, as they are all dead and, therefore, have no chance of becoming alive.

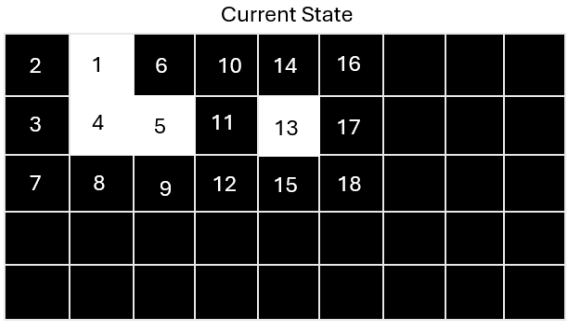


Figure 4 4 - Efficient neighbour counter

Since a cell's next state is only dependent on its neighbours at a distance of 1 cell, the most efficient method is to only check each alive cell, and each of its neighbours, keeping track of how many alive neighbours each cell has. Exploring each tree of neighbours ensures that every cell that could potentially stay alive, become dead, or become alive, is checked, as seen in Figure 4. Starting at cell 1, the number of alive neighbours for this cell is counted and stored. Then, this is repeated for its alive neighbours, and the tree of alive neighbours is followed. In this case, cells 1, 4 and 5 are checked in this way. When these cells and their neighbours are checked, and it is known how many alive neighbours each one has, the program moves to the next alive cell tree, at cell 13. Then, the next state is calculated for each of the visited cells. In this way, every alive cell can be determined to be alive or dead in the next state, but also every cell neighbouring any alive cell is checked, so that it can become alive in the next state (as is the case with cell 6). However, none of the cells that have no contact with alive cells are checked, since it is impossible they could become alive, and by default remain dead. In this way, only 18 cells are visited and checked for this example, as opposed to the 45 cells visited using the naïve algorithm, meaning the performance is more than doubled for this example. However, this is entirely

dependent on the number of alive cells, and if the entire grid was alive, the performance would be the same. As such, the time complexity is $O(m)$, where m is the number of alive cells. In the worst case, $m = n^2$, but in the best case, $m = 1$.

However, this implementation does not make use of parallelisation, and this solution was explored to reach a more performant CPU benchmark.

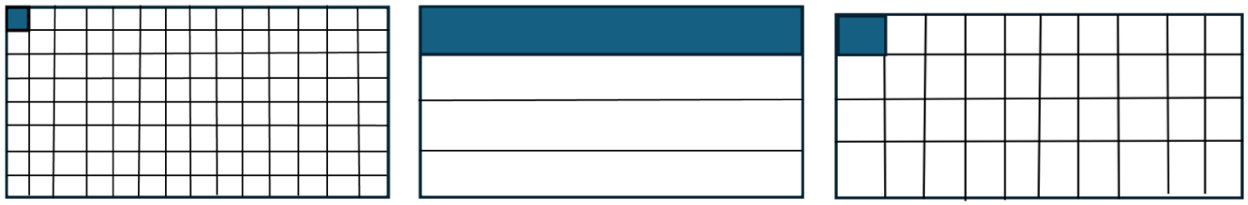


Figure 55 - Diagram displaying different parallelisation methods, with shaded box representing the work of a single thread.

The naïve method of parallelisation consists of, similarly to the naïve method without parallelisation, of computing each cell individually, as seen in Figure 5. The speedup can occur by spawning 1 thread per cell, so that the next state of every cell is calculated at the same time. However, obvious flaws with this method arise:

1. For large grids, there are not enough CPU threads on a computer to calculate every cell at once. Tasks must therefore be queued while waiting for a thread to be available.
    a. The computer used for testing has 16 threads, and so the calculations could only be done for 16 cells at a time.
2. The reconstruction cost is enormous, as each result of each thread must be collected and appended back to the 2D array storing the grid.
3. There is significant overhead associated with spawning each thread

A significant improvement that solves the first issue is to count the number of available threads and divide the grid into that number of rows (containing $\frac{720}{16} = 45$ rows each). This has the advantage of spawning only 16 threads, improving on issue number 3 as well. Each thread can then apply the naïve method of iterating through each cell to determine its next state. The more efficient method shown in Figure 5 above is unfortunately not possible when dividing the grid into sub grids, since the entire tree of a given alive cannot be followed past the boundaries of its sub grid. Edge cases between sub grids are handled by including edge rows in both

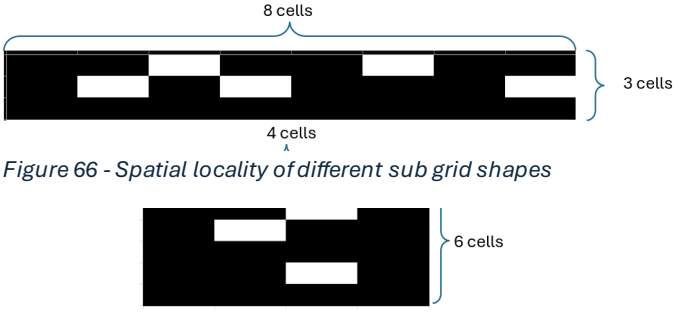


Figure 66 - Spatial locality of different sub grid shapes

adjacent sub grids, so they can calculate the accurate next state. Issue number 2 is also improved upon since there are now only 16 results to reconstruct.

However, this method can be improved further by considering the use of spatial locality. When dividing the grid into square sub grids instead of rows, more of a given cell's neighbour are visited, on average, as seen in Figure 6. In this case, both grids possess 24 cells, but in the top shape, only the state of cells in the middle row is checked 8 times (as they have 8 neighbours in the grid). In the bottom shape, however, there are a total of 4 rows where they are checked 8 times. In CPU implementation, retrieval of frequently used data is stored in cache, which is located in close proximity to the CPU and decreases the time it takes to retrieve data [9]. When the state of a cell is retrieved from memory, it is then stored in cache. For the bottom shape in Figure 6, since the state of middle rows is accessed many times, each operation benefits from the close availability of the states of these cells when stored in cache. This is not the case for the top shape, as there is only 1 middle row, and the state of cells must be retrieved from memory more often.

Each of these solutions was implemented in Python and the execution speed was measured.

2.4 Testing and Results

Initial testing involved ensuring the next state logic was correct, by visualising the grid and checking each frame. Then, the execution speed was measured on both Python and C++ implementations. Python has the advantage of giving easy thread control, but as it is not a compiled language, it is significantly slower, and so a C++ implementation was also derived to test the true speed of the CPU benchmark.

The testing was conducted by timing the time it takes to generate 1000 generations, for increasing grid sizes.

Grid Size	Naive C++ method (no parallelisation)	Alive cell tree Python method (no parallelisation)	Parallelisation : Spawning 1 thread per cell	Parallelisation : Splitting grid into rows	Parallelisation : Splitting grid into blocks
20x20	0.0732	0.983	7.66	1.51	1.56
40x40	0.238	1.22	27.3	3.50	3.31
100x100	1.334	3.02	145	15.6	14.8
1280x720	122	49	603	3023	1896
1920x1200	186	215	Timed Out	8400	3335

Table 3 - Time taken in seconds for each method to produce 1000 generations for various grid sizes.

Two important things stand out from this testing:

1. the C++ implementation can run at a frequency of $\frac{1000}{12224} = 8.19Hz$ for a native resolution grid as needed from the FPGA. This means that custom hardware will be needed to meet the desired frequency of 60Hz for a grid of that size, and 8x speedup is necessary.
2. The parallelised method of splitting into blocks is the most efficient for larger grid, more than twice outperforming the row method for a 1200x1920 grid, owed to the spatial locality of this method.

The C++ implementation passes the required specifications outlined in Table 2, since the time complexity of this implementation is $O(n^2)$, and the state of each cell is stored in a 2D array of dimensions 1280 x 720, giving a memory complexity of $O(n^2)$ as well.

It must be noted, however, that an implementation in Verilog will be very different than in software. It will differ in 3 main ways:

1. The Programmable Logic (PL) does not benefit from spatial locality and a cache system.
2. Reconstruction costs are negligible since thread outputs can in theory all be outputted concurrently.
3. Memory management is particularly important due to the limited amount of LUTRAMs and LUTs.

Due to Reason 1, the square sub grid method may not be more efficient than the row method in Verilog. In theory, due to Reason 2, the 1 thread per cell method could be the most efficient, though Reason 3 will limit the feasibility of this method.

2.5 Evaluation

Efficient algorithms have been devised and a clear idea of how they can be developed in Verilog is apparent. However, parallelising the Set method that only checks alive cells and their neighbours. One possible way to do this would be to identify all the distinct alive cell trees, and spawn threads for each tree. In this way, there would be no overlap and each thread could fully follow the entire trees. For some starting patterns, this could be very efficient. However, this method would also mean that a different number of threads is used for different starting grids, and extensive logic would be needed to manage this. Furthermore, identifying alive cell trees in Verilog would very difficult as well.

3 Custom Hardware

3.1 Theory & Design specifications

Efficient algorithms were designed in the previous section. However, as mentioned in the Testing and Results section, implementing an algorithm in software varies significantly with implementing an algorithm in hardware. The implementation is limited by the physical build of the PYNQ-Z1 board, and its specifications outlined in Section 1.4.

Therefore, the specifications of the custom hardware can be seen in Table 4:

Table 4 - Design specification for the Verilog implementation

Specification	Quantitative measurement	Test
Must be synthesisable	Written in Verilog HDL and can generate bit stream on Vivado.	Test whether the bit stream can be generated.
Must be parallel	At least 2 separate threads compute the next state for separate cells	Count the number of threads in the design.
Must not exceed available memory	Must use less than 630KB of Block RAM, 17,400 LUTRAMs.	Look at Vivado Status Report estimations.
Must not use more than number of available logic elements	Must use less than 53,200 LUTs.	Look at Vivado Status Report estimations.
Must be able to take in user input	User input is passed through to the PL	Test whether a user defined input can be seen on the FPGA output.

These specifications require significant designing due to the ambitious goals on a limited platform.

3.2 Experimental Method and Development

As seen in section 1.3 System Overview, the system comprises of a UI, the PYNQ-Z1 board, and an output. The UI must pass the grid inputted by the user to set the initial grid, as explained in Section 4.3.1.3 below. This initial grid is then used to calculate the next state following GoL logic. The next state is then outputted through to the monitor, to be displayed for the user. To store the initial grid, memory management must be devised for the board.

3.2.1 Block Random Access Memory (BRAM)

The first significant design challenge arises when storing a large grid, due to the limited memory on the FPGA. There exist 3 possibilities:

1. Store it in Distributed Memory using LUTRAMs
2. Store it in DDR3 RAM

3. Store it in Block RAM

The first option is the simplest since it only requires the creation of a 2D array in Verilog which can easily be indexed and updated directly. However, immediately it is clear there are not enough LUTRAMs to store a 1280x720 grid, and more storage is required.

The DDR3 RAM, with 512MB of storage, can store large amounts of data, but is not optimised to store data for the PL, and a large number of clock cycles are required to retrieve data from it.

Therefore, we opt to use Block RAM. At 630KB of RAM, we are able to store at least 2 full grids, since 1 grid takes up $1280 \times 720 = 921{,}600\ bits$, we require $1{,}843{,}200\ bits$, and we have $5{,}040{,}000\ bits$ of storage.

This means an initial module must be created that can retrieve the data from passed from the PS in the register file and write it to the BRAM.

The ability to store two separate grids in BRAM is crucial, as after calculating the next generation of a cell, we cannot immediately overwrite its current state with its future state, as we will need its current state for the calculation of future states of other cells. Thus, two distinct RAMs are needed– one to read the current generation from and another for writing the next generation to. The dual port configuration of each RAM means a cell can be read simultaneously to perform calculations on and be used in generating a visualisation.

At any given time, one RAM will the current state of the grid, and the other the next state of the grid. When it comes time to update the frame, the two RAMs switch positions. The next generation becomes the current generation, and the previous generation is overwritten as the next generation is calculated. Earlier implementations involved having fixed current and next generation memories, so when it would be time to move to the next frame the entirety of the next state would have to be copied to the current state memory, a costly process.

3.2.2 Writing an initial state to memory

The initial grid is passed through the register file from the PS. As explained in Section 4.3.1.3 below, the maximum allowed bid width for registers is 32 bits, and so each row of the initial grid is split into forty 32-bit words. The values of the 40 registers are concatenated into a single line, which forms the 1280-bit word stored in memory. This means an entire row can be written every cycle, requiring 720 cycles to write the entire initial grid to memory. A flag is used to indicate when initialisation of memory is completed, so that this logic is never executed again. The initial state is always written to BRAM1, but a

system of flags is used to read from BRAM1 and write the next state to BRAM2 when calculating the next state, represented as a state machine.

3.2.3 Hardware State Machine

Once the BRAM has been initialised, a flag ensures the state machine alternating between displaying the grid and the next state calculation is entered. This is done by clocking the Pause flag, so that as the Pause flag changes state, the next state is calculated, and a further change caused the grid to be displayed. When the Pause flag is kept constant, the grid continuously displays. This occurs when the user pauses the execution of the next state logic.

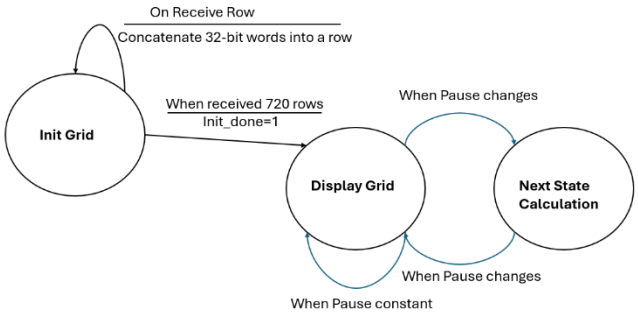


Figure 7 - Hardware State Machine

A change in pause also flips which BRAM is being read and which is being written, so that the Display Grid state displays from where the next state was written to, and the next state calculation reads from the location that has just been displayed.

3.3.2 Displaying Current State

The base version of pixel generator worked by iterating through each pixel in a 620x480 frame and assigning them an RGB value calculated from its position within the frame. Similarly, to generate a frame we iterate through every pixel in a 1280x720 frame and assign values based on the corresponding state stored to the corresponding location in memory.

We have hardware concurrency in mind when designing the video output. Fully utilising the dual port BRAM with one port allocated to obtain the line buffer for game logic, while the other one is allocated for video readout. We have BRAM A and B, with one being read and the other being written to in the same cycle, and we designed it to avoid any potential memory clashes for read and write operation. During the video readout, a row counter is established, and the row number is sent to the mode_selector for correct wiring to the BRAM for read based on the configuration in BRAM role switching. Row is then stored as a 1280bit register and we iterate through each element to assign it for RGB states. The rate of frames generated per second is controlled by the 'PAUSE' switching signal created by Python logic. This allowed for greater flexibility for the user to control the frame rate to speed up or slow down the observation, allowing them to be educated and visualise the evolution of the cells generation through their own input.

3.3 Calculating the next state

The next generation is calculated one row at a time which is split into 3 different processes. When a new frame is requested, it starts a counter that goes through the addresses of each row of the current generation. This address is then sent to the module line buffer, which fetches the relevant row and its two adjacent neighbours. These three rows are sent to a final module that computes the next generation of the row and writes the result to the appropriate memory address.

3.3.1 Row counter

The row counter block only starts when we want a new frame. When this happens, it sets the address of the row we want to calculate, to zero. It waits for a valid signal from line buffer to be true and only then begins to iterate through all 720 rows that make up one generation.

3.3.2 The Line buffer

The line buffer outputs a row and its immediate neighbours – named *top*, *middle* (which corresponds to *calc_row*) and *bottom*. There are two cases where *top* and *bottom* respectively are not stored in the BRAM holding the current generation – the very first and last row of a generation. When this is the case the top and bottom row will be filled with zeros, as in our model the grid is surrounded by dead cells.

Reading from BRAM takes one clock cycle. Ideally, we want the line buffer to take in the address of a row and then output the contents of three rows in the same clock cycle. Typically, as we move down row by row, *top* will become *middle*, *middle* will become *bottom,* and a new value fetched from memory will be written to bottom. Shifting the output like this means we only perform fetch a new row from memory one time each iteration. We prepare the fetch address output to update this value the next sequence – this is how line buffer has the right memory fetched to write at any given time. The process is outlined in the diagram below. This is possible for all rows but the first, where top will be filled with zeros, but we will have to wait two clock cycles to fetch content for middle (the first row) and bottom (the second row), during which time valid will be set to zero.

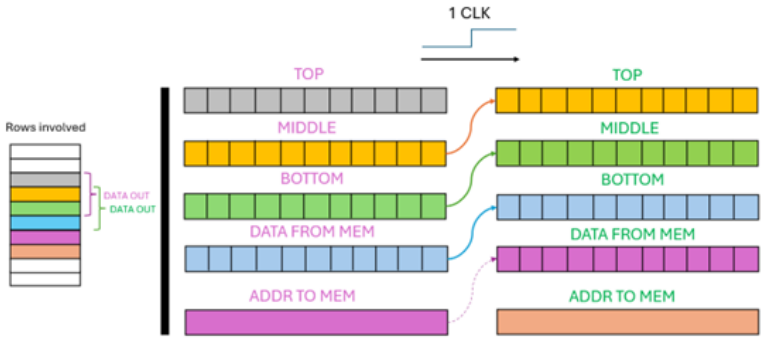


Figure 8 - Line buffer

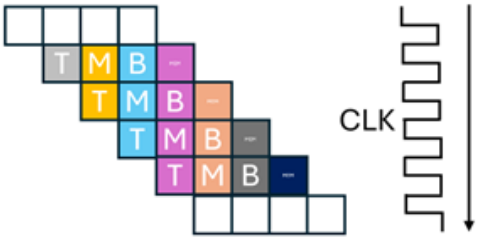


Figure 9 - Each clock cycle, the lines being read is moved along.

3.3.3 Calculating the next state of a row

Once received, the three rows are padded with zeros at each end to handle edge cases (because the grid is surrounded by dead cells). Every cell in a row and its Moore neighbours are sent to a computation unit. As the next state of every cell in a row is calculated in parallel, each cell has their own unit, totalling up to 1280 units. The results of all these units are then concatenated in a single register which is then written to the appropriate address of the BRAM designated to storing the next state memory. Flags from modules higher up in the hierarchy control a write enable to prevent erroneous lines being written into memory.

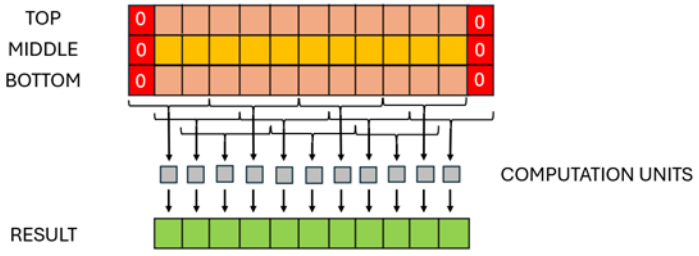


Figure 10 - The next state of each row. note the bits padded on either side of the input rows.

3.3.4 Calculating a Single state

Each computation unit receives a cell and its neighbours. Standard game of life rules can be applied quite easily to determine the next state of a single cell.

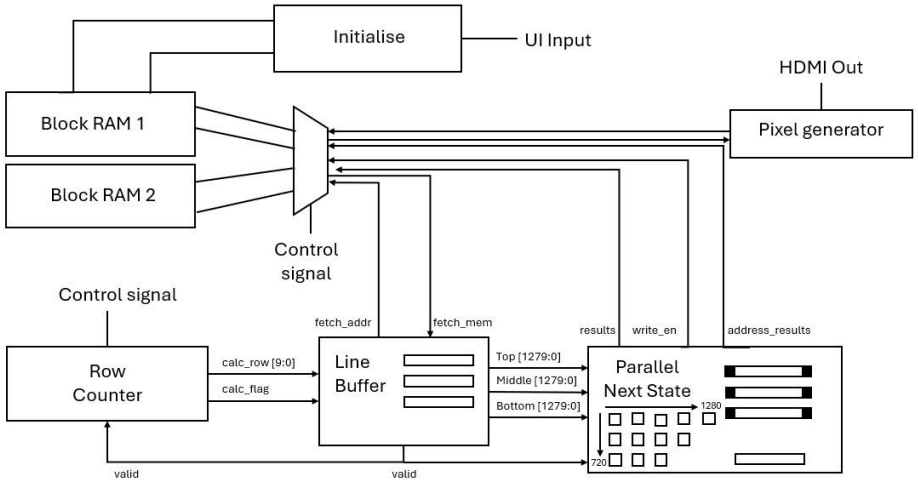
3.4 Memory Management and Allocation

Originally in our design, we divided each state into 3 block RAMs, initially using 6 block RAMs for 2 states. This is due to the limitations of Verilog since the register cannot exceed one million cells. This allowed us to have dual port I/O in every block RAM and achieve a high level of parallelism by accessing 3 rows of the matrix at a time. This design was the most ideal case for the optimisation of a 1920x1080 grid. However, with further testing, it was found that the PYNQ board did not have the sufficient onboard block RAM and would have to utilise DDR memory which is significantly slower so therefore we decided to lower our native resolution to 1280 x 720 so that it was possible to fit a grid within the block RAM. This decision allowed us to create a minimal viable product for the highest memory throughput using block RAM.

In our block RAM for the 1280 x 720 grid, we created a design to use 2 block RAMs, one for each state. We had to utilise the true dual port native block RAM, and decided to exclude the primitive register as it will introduce a delay of 1 cycle and was not ideal in our system. With a bandwidth of 1280 bits for each port, we were able to read an entire row of the matrix, maximising the processing power with the data hungry CPU processing cluster. During the operation, BRAM_A will be read by the line buffer for further calculations and the other read port will be used for the pixel generator video outstream, maximising the dual port potential in this case. On the other hand, the BRAM_B will ingest the new row results created by the 1280 CPU cluster and the whole row could be written to the RAM in a single operation as we need to concatenate all the results together for the 1280 bits word. In the following frame generation, the role of RAM_A and RAM_B is reversed by a MUX channel selector hence avoiding the need for a temporal block memory for transferring the data between the two blocks. This avoided the computationally expensive data migration from one block to the other and saved around approximately 720 cycles for the transfer. This method comes from having a lower memory allowance while outperforming data shifting to other register. This led to a near 100% BRAM utilisation in our synthesis report, optimising every tile for cell state calculation, once again maximising the full potential of the 1 cycle delay high throughput block memory.

3.5 Hardware System Overview

Figure 11 - Hardware system overview



The basic components of the hardware we created are as follows, seen in Figure 11.

- Memory to store current and next states
- Logic to write an initial grid to memory
- Logic to read a state and create a visualisation.
- Logic to find the next generation of a given row and write it to memory
- Logic dictating control sequencing – when to start and stop the calculation of a next state and choosing what RAM to read from and write to

4. User Interface (UI)

4.1. Theory

The requirements of the project state that "human-computer interaction" must "allow a user to manipulate and understand the visualisation" [1]. This links to the project's goal of being an educational tool and gives many possibilities for developing an impactful UI.

Good UI design is characterized by 6 key rules [10]:

1. Intuitive – users should immediately understand the UI with minimal instructions.
2. Familiar – users should have been exposed to a similar design before
3. Responsive – the UI must respond to user inputs with negligible delay.
4. Consistent – the UI must respond in a predictable way.
5. Minimalistic – the UI must show as few elements as it needs to reach its goal.
6. Inclusive – the UI must be usable by all kinds of people in a non-discriminatory way.

However, the nature of GoL trivially gives rise to other important aspects of the UI. The user must be able to set the initial grid, so that the tool can be used educationally, to study the effects of changing some initial cells. Another significant requirement, as per the project specification, is to be able to adjust parameters while the simulation is running.

4.2 Design Specification

The 6 key rules described in the previous section were then refined and expanded in Table 5, with measurable metrics and devised tests so the UI's impact can be measured in the testing phase.

Table 5 - Design specification of UI

Specification	Quantitative measurement	Test
Must be intuitive	Must be understood and correctly utilised in under 10 seconds.	Give the system to 5 users and time how long it takes them to draw the desired grid after explanations.
Must be familiar	Users have used or know of at least 1 similar UI before.	Ask each user to name what this UI system reminds them of. Record their response.
Must be responsive	Response time to input under 0.3s, the average human reaction time [11]	Measure the delay time between when the user pauses and when that pause is displayed on the monitor.

Must be consistent	No noticeable change in the UI's setup.	Trivial
Must be minimalist	The number of different pieces of information on screen cannot be greater than 7, the maximum number of items short term memory can store at a time [12].	Count the number of distinct information pieces that can be seen on the screen.
Must be inclusive	The UI must differ from standard, non-inclusive UIs in at least 2 ways.	Count the number of ways the UI is inclusive.
Must be able to set the initial grid	Trivial	Trivial
Must be able to interact with the evolution of the grid	Trivial	Trivial

4.3 Experimental Method and Development

4.3.1 Design of the solution

From the specifications above, it was decided that the user must be able to achieve two tasks:

1. Set the initial grid of the GoL, to be able to study the effects of changing the "alive" starting cells and understand how the grid evolves through the generations.
2. Pause the simulation, so that the current generation could be studied in more detail.

This automatically fulfils the last two specifications but remains significantly open ended in how this should be achieved. Table 6 contains detailed analysis of possible solutions to implement this behaviour:

Table 6 - Proposed UI solutions

UI method	*Advantages*	*Disadvantages*
Remotely connected smartphone application, where an initial grid can be drawn, with a button to pause the simulation	• Intuitive and familiar. • Gives significant freedom of choice of initial grid. • Easily add more advanced features.	• Difficult to implement directly since phones are not convenient developer environments. • Difficult to operate for people with low fine motor skills (not inclusive) since the screen is small.

		• Potentially less responsive since wireless connection through AWS.
Simple Python app with keyboard inputs for drawing and pausing	• Simple to implement. • Minimalist • Easily add more advanced features	• Not familiar • Keyboard inputs are not intuitive, will take some adjustment to place correct live cells. • Difficult to use for people with low fine motor skills (not inclusive)
Computer Vision technique to detect hand movements, with pen motion to draw alive cells, and hand movements to pause simulation	• Highly inclusive, uses universally recognizable hand signals (for people from diverse cultures or who have never used a computer before. • Highly inclusive, people with low fine motor skills still able to draw on a large canvas (the air in front of them). • Minimalist. • Familiar as similar to VR/AR technology. • Intuitive, day to day signals.	• More difficult to implement. • Potentially less responsive. • Less reliable.

From the above analysis, the 3rd option of developing an intuitive UI using modern computer vision techniques to detect the position of the hands in the air was chosen, due to its intuitiveness and inclusiveness. Common hand signals such as “open hand” and “thumbs up” are commonly used [13] in a vast number of cultures as a means of communication, in addition to speech. The project could, therefore, be used by people with little knowledge of English, no formal education, or no access to computers. Moreover, people (particularly children) with fine motor skills issues, might struggle with holding a pen, drawing on a touchscreen, or operating a computer keyboard [14]. Broad, general hand signals are typically not an issue making the solution more accessible and so

particularly effective as an educational tool for children. This means the inclusive aspect of the specification is met.

4.3.2 Development of the solution

To achieve its goals, the UI must, therefore, permit the user to "draw" in the air, where they would want the cells to be initially alive. Then, to be able to communicate that they are finished drawing, and that the simulation should start. Finally, that, while the simulation is being executed, it should pause at the current generation.

Two approaches could be used to achieve this:

Table 7 – 2 different UI approaches

Approach of hand signal detection	*Advantages*	*Disadvantages*
Train a deep learning model to detect specific hand movements ("thumbs up", "pinched fingers", "open hand") based on a dataset of annotated training images.	• Model specific to the problem making it more efficient. • More robust for given hand signals.	• Requires significant training time and resources. • Not easily modifiable. • Might struggle with detecting a "pinched fingers" situation.
Train a deep learning model to detect joint locations on the hand (fingertips, articulations, etc), and use the respective distances between these joints to determine the current hand signal.	• Easily modifiable. • Possibility to measure precise distance between index and thumb fingertip locations for "pen down".	• Less robust in signal detection. • Dependent on hand size and distance to camera.

The 2nd option was chosen in large part due to the possibility of modifying the hand signals based on feedback from early-stage users, which significantly decreases the development time and makes it a more versatile solution. Its disadvantage of being dependent on hand size and distance to camera (since the distance between pixels is used) also means it can be fine-tuned after in-the-field testing, giving it more potential for success.

Three main control signals will, therefore, be used, as seen in Table 8.

1. Pen down, to draw the location of alive cells. Achieved by making one's index and thumb fingertips touch
2. Thumbs up, when the user is finished drawing, to communicate that the simulation should start. Achieved by wrapping all fingers and extending the thumb

3. Hand open, when the user wishes to pause the simulation. Achieved by extending all fingers. Closing the hand would then restart the simulation.

These different hand positions must, therefore, be determined using computer vision.

4.3.2.1 Hand Land Marker

Google's MediaPipe module [15] was trained on over 30,000 real and synthesized images of hands with varying backgrounds, to detect the location of 21 landmarks on a human hand. These landmarks can be seen in Figure 12. Using Computer Vision Zone (CVZone)'s own library [16], these landmarks can be easily accessed in an OpenCV video feed, and their coordinates used in further calculations, according to the logic outlined in Table 8:

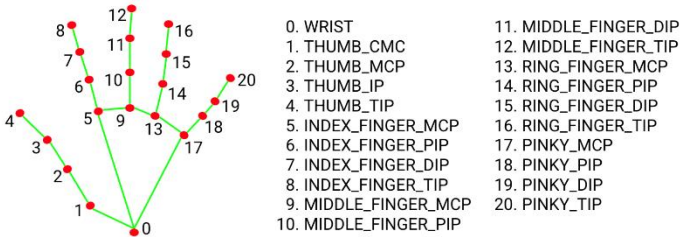


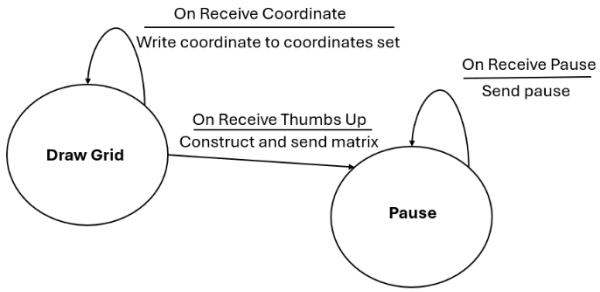
Figure 12 - Hand landmarks and their respective locations [15]

Table 8 - UI logic

Hand sign that must be detected	Relevant hand landmark numbering (Figure 12)	Respective location
Pen down	Thumb tip (4), index tip (8)	The pixel distance between these two landmarks must be below 8 pixels.
Thumbs up	All landmarks except the wrist (0)	The coordinates of the 4 landmarks corresponding to the thumb must be higher in the vertical distance than all other landmarks.
Open hand	All fingertips (4, 8, 12, 16, 20) and wrist (0)	The coordinates of all 5 fingertips must be above a certain vertical threshold distance of the coordinate of the wrist.

4.3.2.2 UI State Machine

Since these signals are communicated by the user at very distinct times, and should never happen concurrently, a state machine is used to ensure different signals are not sent to the FPGA at the same time, following what can be seen in Figure 13:



13- State machine of the UI Application

The "Pen down" coordinates detected by the computer vision model are store in a Python set, which automatically removes duplicates.

The "Thumbs Up" signal then acts as an internal flag to indicate that all coordinates have been received, and the coordinates set is used to populate a matrix with 1s for these coordinates, and 0s elsewhere. This matrix is then sent to the PS on the PYNQ-Z1, and the state changes to the "Pause" state. This logic ensures the PS either receives the initial matrix or the pause signal, but never both, so that appropriate decode logic can be implemented.

This behaviour is especially important since the size of the matrix (921,728 bytes) far exceeds the size of the TCP buffer of the PS (4096 bytes), and must be correctly split into chunks which are then decoded by the PS.

4.3.1.3 Processing System (PS)

The PS initially receives the matrix in chunks of 4096 bytes through a TCP connection, which are then reconstructed into a Python 2D array. The Boolean matrix must then be converted in hexadecimal format so it can be loaded into the register file in the PS. This is the crucial interfacing, since the PL can have direct access to this register file, and so a data transfer can occur.

An important limitation that was solved was that the bit width of the registers is a maximum of 32 bits, and a matrix of dimensions 1280 x 720 must be sent through. The logic shown in Figure 14 is repeated for each row of the matrix.

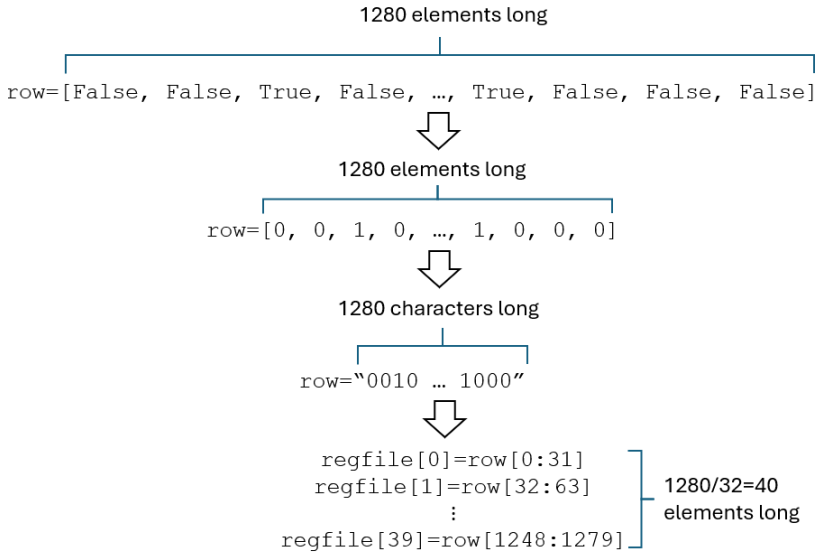


Figure 14 - Row by row logic of loading the matrix into the register file

It consists of splitting the rows into blocks of 32 bits, each joining elements together in a string which is converted to hexadecimal format so it can be loaded into the registers.

A further 3 registers are used to load control flags, as explained in the Custom Hardware section above.

The PS also controls when the next frame should be displayed on the output monitor. This mechanism uses the PAUSE flag to control the mode switch and triggering the calculation of the next state and video read out from the BRAM. Python programmed to iterate the PAUSE flag in the regfile from high to low with an equal time duration in each state, creating a positive and negative edge repeatedly through the AXI controller. The sensitivity list picks up edge change and direct the pulse change to the PL. This allows a flexible change in display frame rate by simply changing the PAUSE signal pulse rate, allowing the user to visualise the impact of their input in terms of cells generation with a speed increase.

This entire system must then be tested for failure, to ensure that it will reliably function when end users use the system.

4.3.3 Failure Modes and Effects Analysis (FMEA)

To ensure the reliability of the system and low risk of failure when presented to users, FMEA was conducted on the user interface, and problems were subsequently fixed and mitigated. Table 9 contains an overview of the potential failures that were identified.

Table 9 - FMEA

Process purpose	Potential failure mode	Severity	Potential causes of failure	Occurrence	Process control	Detection	RPN	Recommended actions
Detection of hand presence in the frame	System crashes	9/10	Hand has moved out of detectable frame	8/10	Extensive user testing	3/10	216	Include a try-catch block for when this occurs. Add appropriate placeholder values in place of the values that are supposed to be measured
Detect when finished drawing	Unable to continue drawing	7/10	"Thumbs Up" detected too early	7/10	Extensive user testing	2/10	98	Include an option to confirm "thumbs up" Include a cooldown that must be awaited before "thumbs up" possible
Classify hand signals	Wrong signal detected	9/10	Model unable to accurately detect location of landmark. Distance thresholds not set correctly	1/10	Extensive user testing	5/10	45	Adjust thresholds based on model performance in real testing scenarios. Use a more advanced model trained on more/better data
Send whole matrix in multiple chunks	Some chunks lost, corrupted or not extracted correctly	8/10	Too much noise on communication channel, TCP end-to-end connection lost	3/10	Error messages when user testing	1/10	24	Send the number of chunks that must be received before sending the chunks, so that the right number are received. Decrease the buffer size to ensure more reliability.

The recommended actions were adjusted based on extensive testing conducted on the final product.

The following actions were taken following the FMEA:

1. A number of Try, Except blocks were incorporated so that if the connection fails, neither the server nor client crash and the connection can be reestablished.

2. The number of chunks of the matrix is sent first, so that the receiver can check the correct number of chunks is received.
    a. This is on top of the TCP protocol that already possesses a system of ACKs to recover lost frames.
3. Thresholds for the pixel distances were also adjusted, tested for people with varying hand sizes so that the system is as inclusive as possible.
4. A 10 second cooldown was added for the "Thumbs Up" signal, so that users have 10 seconds to get used to the user interface without mistakenly showing a thumbs up before being able to draw the frame.
    a. This number was determined based on the specification of the UI needing to be understood in under 10 seconds.

4.4 Testing and Results

The impact of the actions taken as a result of the FMEA must now be measured, as well the entire UI as a whole against the design specifications, to ensure that the system implemented is a "success" and meets the goals of the project.

The testing devised in Table 5 in the Design Specifications section was conducted, with the results shown in Table 10

Table 10 - UI specification test

Specification	Results
Must be intuitive	Average time of 5.5 seconds to understand the functioning
Must be familiar	All users mentioned the UI is similar to the Xbox 360 Kinect.
Must be responsive	Measured response time averaging 30.32ms
Must be consistent	The UI is the same every time it is used.
Must be minimalist	Only 4 elements are provided. The hand position, its zoomed in version, and the two dots corresponding to the thumb and index fingertips.
Must be inclusive	The UI is particularly usable for people with little experience with computers since they can use their fingers instead of a mouse/keyboard. Users with limited motion skills can also precisely draw a starting grid.
Must be able to set the initial grid	The initial grid is set by the user.

Must be able to interact with the evolution of the grid	The user can pause the grid.

A testing user can be seen in Figure 15:

Figure 158 - A random user testing the UI

4.5 Evaluation

Clearly, from Table 10 above, the UI passes all the design specifications and is, therefore, a success. Significant challenges were faced when integrating the UI with the PS, due to the format of the registers (needing hexadecimal) and the limiting 4096 bytes buffer.

The following improvements are proposed for the UI, if time had allowed for the development of a more advanced system. These features would also require significant changes to the hardware architecture:

1. Add more control signals for the user
    a. Ability to zoom into the grid
        i. To enhance the user's ability to study the grid, being able to zoom in specific locations would allow for the user to see specific behaviour in detail
    b. Ability to slow down and fast forward the next state generation
        i. 60Hz remains very fast and unique generation changes are difficult to visualise, so the ability to slow it down further would help in understanding more specific behaviour
        ii. Also, being able to fast forward to more interesting patterns would be beneficial
2. Add the ability to see the grid being drawn in real time
    a. A significant limitation of the current UI is that it's impossible to visualise what the grid will look like until after the "Thumbs Up" signal is shown, and the grid is displayed on the FPGA

        i. Extending the Python application to include a grid visualisation would be greatly beneficial.
3. Give the user the option to choose the starting grid from given options on top of being able to draw it.
    a. Well known patterns in the GoL can produce stunning visualisations, and giving the user the ability to explore these could make the system more approachable by a large audience.

5. Conclusion

5.1 Team management

The work for the project was efficiently split between team members with daily meetings to go over progress, goals, and more extensively integrate the different parts of the project. This significantly increased what was possible with the project. Crucially, significant time and two team member's time for 2 weeks was used to debug Vivado issues and ensure development could continue despite often unclear and unspecific error messages.

A Gantt chart, seen in Figure 16, was written at the beginning of the report and then expanded throughout the development of the project, so that each team member could always refer to it to know what to focus on and the deadlines they needed to hold themselves accountable to. This also helped with the integration, since parts of the project were completed in a timely manner.

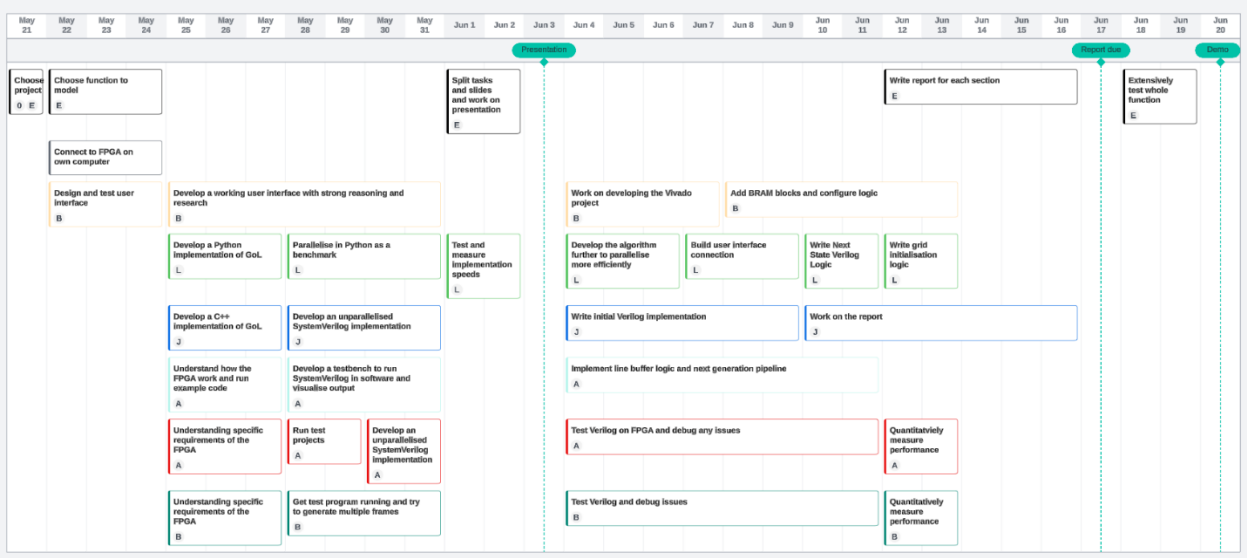


Figure 16 - Gantt Chart

5.2 Final Remarks

The original goal of the project was to use the create the visualization of a mathematical function in real time that would be non-trivial to generate without utilizing hardware acceleration on the FPGA. We were successful in generating an FPGA implementation of Conway's Game of Life, which as per project requirements demonstrated significant performance increases compared to standard CPU only C++ programs - we were able to achieve 60 fps with a 1280x720 grid, whereas the equivalent program only ran at 8FPS. The

FPGA could support even faster framerates, but we were limited by the refresh rate of our monitor. It takes 722 clock cycles to calculate the next state of a grid and the clock linked responsible for this is set to a frequency of roughly 150MHz. This means our implementation is theoretically capable of generating a frame in 4ms which is equivalent to 200,000 FPS!

Unfortunately, we could not implement everything we planned out. As covered before our UI had multiple extra features that increased user interactivity. We were also looking forward to using colour to represent different characteristics of alive cells but to thanks to memory limitations and the approaching project deadline we had to put this ambition aside.

Development was mostly slowed by significant challenges with Vivado, which provided unclear messages and required significant messages, as well as had a very large learning curve meaning initial weeks had to be used to learn how to use it and how we could build our custom hardware. Moreover, defining precise project goals required significant research into what exists and what is possible with FPGAs, so that the project could have real-world applicability and use cases, meets its educational goals but also be possible to complete given the short project timeframe and the new technology being used.

The expertise needed to complete this project went beyond content covered in our modules, ranging from using Vivado, writing synthesisable Verilog code, using computer vision to detect images, and organising how different software and hardware modules must interact through an ingenious and complicated use of flags.

However, this expertise enabled us to create a project with tangible results that could be used by a wide range of users, and contribute to a number of fields. Moreover, we know posses numerous transferrable teamwork, project management, and technical expertise, having achieved our goal of creating an educational tool using a hardware accelerator that would not otherwise be possible to use.

6. Bibliography

[1] E. Stott, “edstott/EE2Project,” *GitHub*, Jun. 14, 2024. https://github.com/edstott/EE2Project/tree/main.

[2] A. Ghuneim, *Moore Neighborhood*. 2000. Available: https://www.imageprocessingplace.com/downloads_V3/root_downloads/tutorials/contour_tracing_Abeer_George_Ghuneim/Figure1.gif

[3] N. Loizeau, “GOL computer,” Aprill 22, 2024 https://www.nicolasloizeau.com/gol-computer

[4] D. A. Faux, P. Bassom, “The Game of Life as a species model”, *American Journal of physics*, vol. 91, issue 7, July 2023

[5] Imperial College London, “Overview, Values and Principles” *Imperial College London* https://www.imperial.ac.uk/human-resources/procedures/work-location-framework/overview-and-values/#:~:text=Imperial%20College%20London's%20mission%20is,for%20the%20benefit%20of%20society.

[6] Digilent, Inc “PYNQ-Z1 Reference Manual - Digilent Reference,” 2023 *digilent.com*. https://digilent.com/reference/programmable-logic/pynq-z1/reference-manual

[7] Heavy AI “What is Parallel Computing? Definition and FAQs | HEAVY.AI,” 2024 *www.heavy.ai*. https://www.heavy.ai/technical-glossary/parallel-computing#:~:text=Parallel%20computing%20refers%20to%20the

[8] G. L. Team, “Time Complexity Algorithm | What is Time Complexity?,” *GreatLearning*, Aug. 24, 2023. https://www.mygreatlearning.com/blog/why-is-time-complexity-essential/

[9] B. Lutkevich, “What is Cache (Computing)?,” *SearchStorage*, Aug. 2021. https://www.techtarget.com/searchstorage/definition/cache

[10] Anon, “Key Characteristics of Good UI Design – According to 8 Experts,” *Studio by UXPin*, Mar. 16, 2022. https://www.uxpin.com/studio/blog/good-ui-design-characteristics/

[11] M. Carroll, “How Fast Is Realtime? Human Perception and Technology,” *PubNub*, Nov. 13, 2022. https://www.pubnub.com/blog/how-fast-is-realtime-human-perception-and-technology/

[12] S. Mcleod, “Short Term Memory | Simply Psychology,” *Simplypsychology.org*, May 10, 2023. https://www.simplypsychology.org/short-term-memory.html

[13] Vanessa Van Edwards, “60 Hand Gestures You Should Be Using and Their Meaning,” *Science of People*, Aug. 21, 2015. https://www.scienceofpeople.com/hand-gestures/

[14] Anon, “How to Identify Issues with Fine Motor Skills,” *CoordiKids*, Sep. 08, 2020. https://www.coordikids.com/issues-with-fine-motor-skills/

[15] “Hand landmarks detection guide | Edge,” *Google for Developers*. https://ai.google.dev/edge/mediapipe/solutions/vision/hand_landmarker

[16] CVZone, “CVZone,” *GitHub*, Jan. 12, 2023. https://github.com/cvzone/cvzone

7. Appendix

GitHub link: https://github.com/lolzio5/JABBAL