University of Trento
Master's Degree in Physics

A Game of Life

Student: *Guglielmo Grillo*

Course: *Laboratory of Advanced Electronics [145230]* – Teacher: *Prof. Leonardo Ricci*
Date: *June 24, 2022*

1 Conway's Game of Life

Conway's *Game of life* is a zero-player game designed by John H. Conway in 1970. It is one of the most famous cellular automatons due to the simplicity of its rules and the wide research done by mathematicians and enthusiasts around the world. Furthermore, Conway's Game of Life is Turing complete and for this reason it can compute, albeit inefficiently, anything that can be computed with any regular programming language.

1.1 Rules

The universe of this model consists of an infinite square grid where each cell can be in one of two states: either dead or alive. In this project, "dead" states are encoded as zeros and represented with white cells while the "alive" cells are encoded with ones and represented with black squares.
The evolution over the discrete-time is regulated by the state of the nearest cells in the eight directions, here labeled with the cardinal directions:

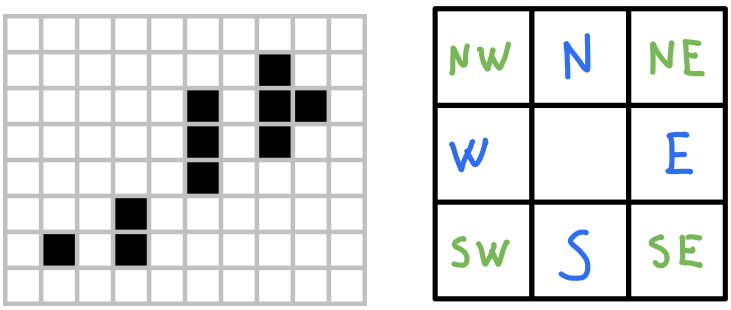

Figure 1: Left: Example of the game state; Right: the convention used to label the neighbors.

1. If a cell has less than two alive neighbors, that is zero or one alive neighbor, it dies.
2. A cell with exactly two alive neighbors will survive the generation.
3. A cell with exactly three alive neighbors will be born if dead or survive if already alive.
4. Any cell with more than three alive neighbors will die due to overpopulation.

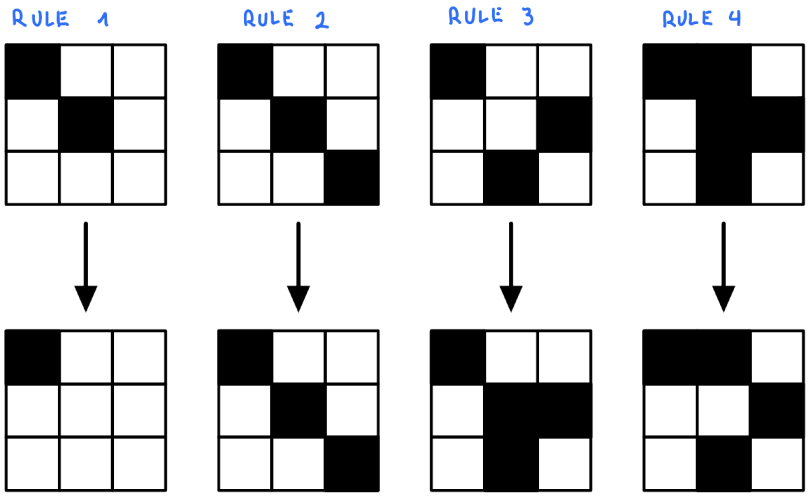

Figure 2: Example of applications of the rules to the middle cell: the first cell dies of loneliness, the second one keeps living, the third is born from the surrounding three while the last one dies of overpopulation.

1.2 Archetypal patterns

These simple rules give rise to a multitude of stable patterns, most of which are yet to be discovered. Some examples of configurations that will be simulated in this project are:

1.2.1 Still life

Still life forms are constant patterns that do not show any evolution over time. Usually, they support the evolution of more complex systems or are the leftovers of unstable patterns. In the context of this project, they provided a way to reliably test the behavior of the module $CELLULAR\_AUTOMATA$ independently from the evolution over time.

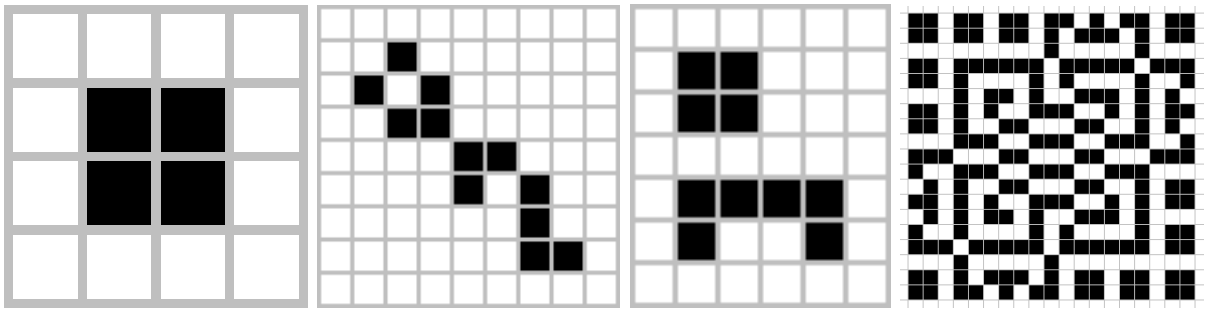

Figure 3: From left to right: the *Block*, the *Boat tie eater head*, the *Block on table*, an example of strict still life, and an example of maximum-density pattern in a 19×19 grid.

Examples of nontrivial still life are *strict still life*, configurations stable only if a supporting island is present, and maximum-density still life, configurations that aim at having as many stable living cells as possible in a fixed $n \times n$ square. The last ones are testing grounds for constrained optimization algorithms.

1.2.2 Oscillators

Oscillators are patterns that repeat themselves periodically after a fixed number of generations called the period. They are the simplest life form that shows an evolution between generations. Due to these characteristics, they were essential to test the first time-dependent build of the projects without regard for the actual evolution. If the pattern was displayed twice then the evolution was stable.

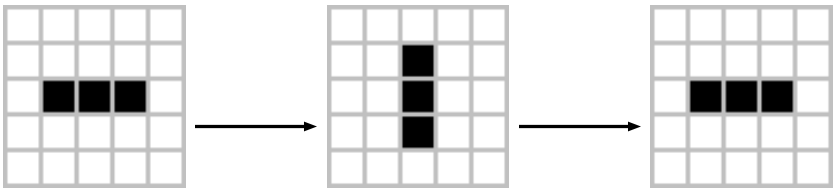

Figure 4: *Blinker*, a period 2 oscillator.

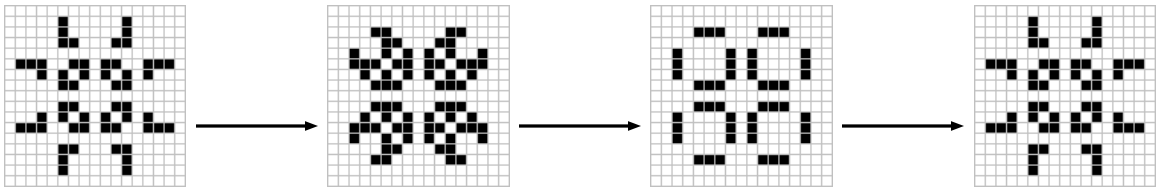

Figure 5: *Pulsar*, a period 3 oscillator.

1.2.3 Spaceships

Spaceships are patterns that repeat themselves after a fixed number of generations displaced by a non-zero amount. They are the most complex patterns that can be reliably simulated with periodic boundary conditions and the resources available. Spaceships are the basis of more complex patterns and allow for the transmission of information between one configuration and the other. Spaceships are characterized by an average speed expressed in terms of c, the maximum speed allowed by the simulation. This maximum speed corresponds to

$$c = \frac{1 \text{ cell}}{\text{unit of time}}$$

and is due to the fact that the rules provide only nearest neighbors coupling. Note that the diagonal directions are still considered one cell distant. Formally, if a pattern moves of x cells in the $\hat{x}$ direction, y cells in the $\hat{y}$ direction in n time unit, the average velocity is defined as:

$$v = \frac{\max\{|x|, |y|\}}{n} \, c$$

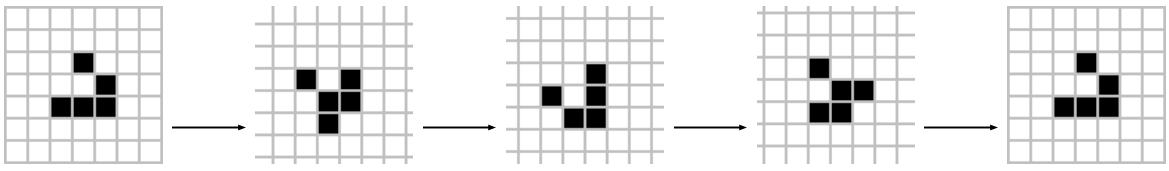

Figure 6: *Glider*, a period 4 spaceship with speed $c/4$.

2 Motivations

Conway's game of life was chosen for the exam project because it allows making use of the intrinsic parallelism of an FPGA while still being relevant in physics simulations[3]. As each cell evolves based on the state of the surrounding cells, at each time step it is necessary to communicate the state of each cell to its neighbors. Only after the communication step is completed, each cell will update its status according to the rules. On a traditional CPU this will take a number of clock cycles that scales as, at least, $\mathcal{O}$(number of cells). On an FPGA each cell can communicate its status in parallel with the others giving rise to a time complexity independent of the number of cells.
Recently, there has even been an interest in applying this intrinsic parallelism to machine learning, see, for example, [5].

3 Design and Code

3.1 AGameOfLife.v

The module *AGameOfLife* is the entry point of the project. Its main purpose is to link the singles *CELLULAR_AUTOMATA* with periodic boundaries and keep track of the VGA output based on the data it receives from the other modules.
The coupling is done via generate blocks [7]. This fictions construct instructs the Verilog preprocessor to generate a predetermined number of modules that differs only by an arbitrary parameter. In the same step, the initial conditions are set using an hard coded register of dimension $L2 = L \times L$ called *wb_initial_status*.
This module also drives the three color channel of the VGA interface via a series of *if-else*s:

```
if( (wb_status[ `L*( (wb_vga_y >> wb_zoom ) % `L) +
    ( (wb_vga_x >> wb_zoom) % `L) ] == 1'b1) & (wb_vga_x<479) ) begin
        // If the cell is alive draw a black pixel
  end else begin if( (wb_vga_x[wb_zoom] ^ wb_vga_y[wb_zoom]) & (wb_vga_x
   <479)) begin
    // Draw a gray square in the background
  end else if( ~(wb_vga_x[wb_zoom] ^ wb_vga_y[wb_zoom]) & (wb_vga_x<479)
   ) begin
    // Draw a white square in the background
  end else begin
    // In any other case set the color to black.
  end
end
```

Some words are due on each of those *if*s:

```
if( (wb_status[ `L*( (wb_vga_y >> wb_zoom ) % `L) +
    ( (wb_vga_x >> wb_zoom) % `L) ] == 1'b1) & (wb_vga_x<479) ) begin
        // Set each channel using  4 bit colors
        // according to the cell's state
  end
```

This *if* is the most complex one and has to be interpreted in the following way:

- *wb_vga_y » wb_zoom*: by right shifting the x and y coordinates of *wb_zoom* bits it is possible to assign *wb_zoom* pixels to the same cell.
- *('L*y%'L)+(x%'L)*: the use of the modulus operator % allows to map the two variables $x \in [0, 799]$ and $y \in [0, 524]$ to the range $[0, L2 - 1]$ with periodic boundaries conditions. Notice that the Xilinx's allows the use of the modulus operator only with a power of two. This is probably because the modulus operator is directly translated into bit-shift operations. The subsequent equality checks the status of the cell.
- *wb_vga_x<479*: will draw only on a 480×480 region. This is a pure aesthetic choice.

```
if( (wb_vga_x[wb_zoom] ^ wb_vga_y[wb_zoom]) & (wb_vga_x<479)) begin
  // Draw a gray square in the background
end
```

This if checks whether the sum *w_vga_x+w_vga_y* is even or odd via the value of the least significant bit (LSB) of the two registers. As an even number is produced by the sum of either two even or two odd numbers, a simple *XOR* on the LSB is sufficient to check if their sum will be even or odd.

3.2 Conway.v

The module *CELLULAR_AUTOMATA* is contained in the file *Conway.v* and simulates the behavior of a single cell. It receives the state of its neighbors in input and outputs its state updated at each positive edge of *clk_in*.
It is important to highlight a peculiar behavior of the module: it is possible to force a state via the flag *set_state* and the input *initial_state*. However, the state will be updated only on the next positive edge of *clk_in* to avoid having surrounding cells go out of sync. As the switch *SW*[3] controls both the initial conditions and the most significant bit (MSB) of the time evolution speed, when it is turned on the evolution will slow down for a couple of seconds and then reset. This behavior was kept to allow for both a reset of the state and a way to pause momentarily the evolution.

3.3 VGA_DRIVER.v

The communication with the monitor was implemented via a VGA cable and follows the standard *VGA 640x480 at 60Hz*. The VGA cable consists of six types of pins:

- Three color pins that are driven by four-bit registers each;
- *VGA_VSYNC* and *VGA_HSYNC* pins that communicate to the monitor that an end of a line is reached and it is time to reset the vertical or the horizontal position. The standard specifies a *negative polarity*: the default state of *VGA_VSYNC* and *VGA_HSYNC* is HIGH while a line ending is signaled with a logical LOW;
- Ground pins.

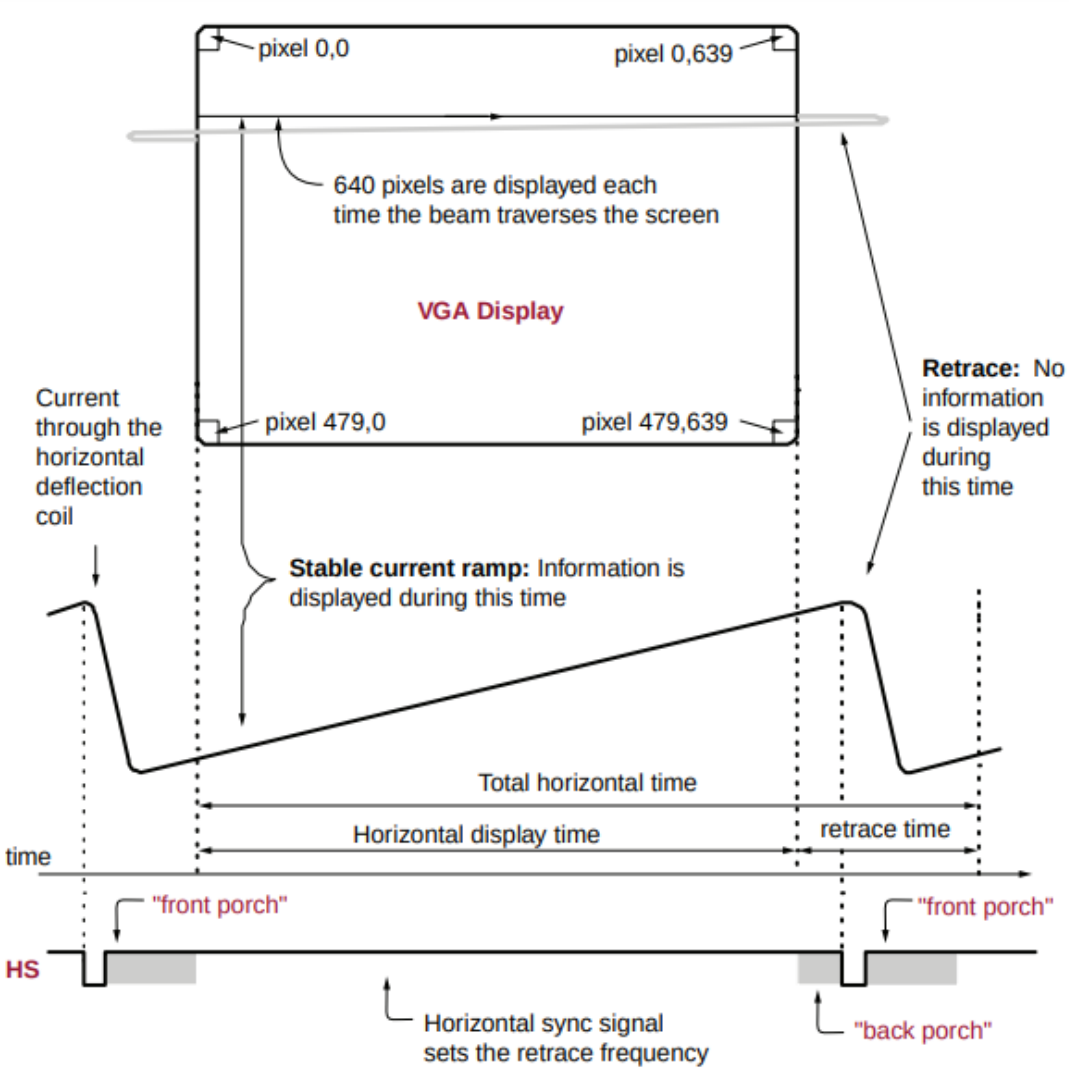

Figure 7: Summary of the VGA timings. Image from the *The Spartan®-3A/3AN FPGA Starter Kit's user manual.*

The behavior of the standard is, as one can see in *Fig* 7, based on cathode ray tubes (CRTs) and assumes an electron gun with non-instantaneous speed. The electron gun starts pointing at the pixel $(0, 0)$ and moves one pixel to the right at each tick of VGA_CLOCK. Once the pixel $(0, 639)$ is reached, the gun needs time to point back to the left. On the driver's side, this is equivalent to having 160 phantom pixels. These pixels are subdivided in the following way:

- 16 Front Porch pixels that serve as a buffer to let slow guns complete their task;
- 96 Sync Width pixels where VGA_HSYNC goes LOW to signal the actual end of a line;
- 48 Back Porch pixels before the actual beginning of the new line.

This process is repeated for each line. When the gun reaches the line 479 a similar process is repeated with a vertical front porch (10 pixels), sync height (2 pixels, VGA_VSYNC is LOW), and a vertical back porch (33). During this process, a total area of $800x525$ pixels was covered. Of these only $640x480$ are actual RGB pixels. Notice that $800 \times 525 \times 60Hz =$

$25.2MHz$, which is the optimal clock frequency.
It is important to highlight that during the "phantom pixels" the driver expects LOW levels on all the three colors. Not doing so results in a desync of the image.

3.3.1 VGA_CLOCK_480p

This module is a frequency halver that converts the onboard $50MHz$ clock into a $25MHz$ clock using a single flip flop. The *VGA 640x480@60Hz* standard requires a $25.2 \pm 0.125\ MHz$ clock while the frequency halver provides $\sim 25MHz$. All the monitors tested seems compatible with this discrepancy, but it may cause some problems with stricter monitors.

3.3.2 VGA_DRIVER_480p

This module was adapted from [1] and manages all the timings required by the standard while converting it into (x, y) coordinates. The module provides a rst_pix pin that resets the coordinates to zero. Despite no practical application of this pin being found, the pin is left in the module for future uses.

3.3.3 VGA_ZOOM_KNOB

This module uses the signals ROT_A and ROT_B to decode the state of the Rotary Push-Button Switch and update the register that controls the zoom of the VGA output. An internal representation of the rotary shaft encoder via a cam is presented in figure 8. What actually matters are the timings: a rising edge on ROT_A while ROT_B is LOW indicates a clockwise rotation. A rising edge on ROT_B while ROT_A is LOW indicates a counterclockwise rotation.

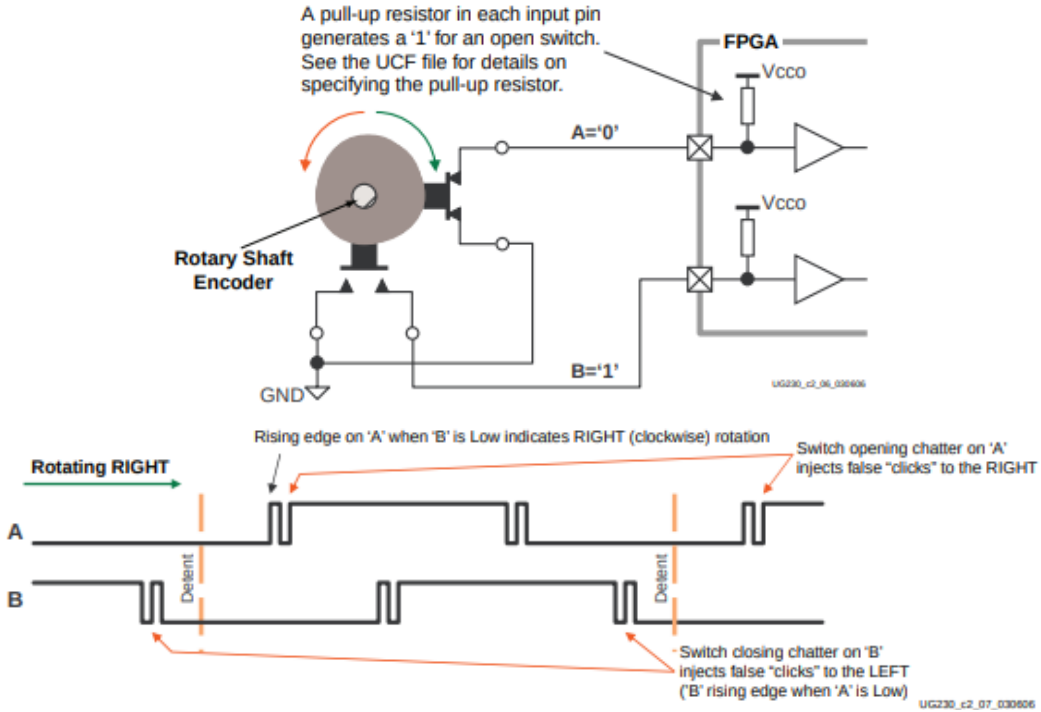

Figure 8: Summary of the VGA timings. Image from the *The Spartan®-3A/3AN FPGA Starter Kit's user manual.*

To avoid the bouncing effect a mono-stable multi-vibrator connected to an activation pin is used. Overflows and underflows are avoided using a conditional statement.

3.4 PatternConverter.py

In the current iteration of the project, the patterns are encoded as binary strings where *1*s encode living cells and *0*s encode dead cells. The reference website [4] uses a different encoding where living cells are represented as *O*s and dead cells as . (dots). Furthermore, LifeWiki doesn't have any restrictions on pattern sizes and uses a newline as a row separator. *PatternConverter.py* allows converting patterns in the LifeWiki format into strings compatible with the project.

```
0 python3 PatternConverter.py pattern.txt
```

The Python programming language[6] was chosen because it allows obtaining minimum viable products (MVP) in no time.

4 Conclusions and possible improvements

At this stage, the project is more like a proof of concept and more improvements are possible. Some of them are listed below.

4.1 Serial communication to set patterns from a PC

The Spartan®-3A/3AN FPGA Starter Kit supports cable connection via both Ethernet cable and two RS-232 Serial Ports. Neither of these cables was available for the project so I didn't manage to implement any kind of communication with the computer. Despite that, once some sort of communication is established, the presence of the pin *set_state* and the register *initial_state* should allow for an easy set of the received pattern.

4.2 Remove constraints on the squared grid

Right now there are two hard constraints on the grid design: the grid has to be squared and the side L has to be a power of two to allow the modulus operator % to work properly. The first constraint could be removed by introducing the constants Lx and Ly and changing $L2$ to $A = Lx \times Ly$.
The second one, instead, could be removed by introducing two independent counters that keep track of the current value of *x_pattern* and *y_pattern*. The counters should respectively reset when $x == Lx - 1$ / $y == Ly - 1$ or $VGA\_HSYNC == 0$ / $VGA\_VSYNC == 0$.
This improvement was not implemented because it would have added complexity to the project without providing any real benefit or new feature.

5 References

The majority of the images and patterns were retrieved from LifeWiki[4], en.wikipedia.org[2] and Project F[1].

References

[1] *Beginning FPGA Graphics - Project F.* https://projectf.io/posts/fpga-graphics/. Accessed: June, 20, 2022.

[2] *Conway's Game of Life - Wikipedia.* https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life. Accessed: June, 20, 2022.

[3] *Lattice gas automaton.* https://en.wikipedia.org/wiki/Lattice_gas_automaton. Accessed: June, 20, 2022.

[4] *LifeWiki.* https://conwaylife.com/wiki/. Accessed: June 24, 2022.

[5] *Machine Learning FPGA Applications - Intel® FPGA.* https://www.intel.it/content/www/it/it/products/docs/storage/programmable/applications/machine-learning.html. Accessed: June, 20, 2022.

[6] *The Python programming language.* https://www.python.org/. Accessed: June, 23, 2022.

[7] *Verilog Generate Block - chipverify.com.* https://www.chipverify.com/verilog/verilog-generate-block. Accessed: June, 21, 2022.