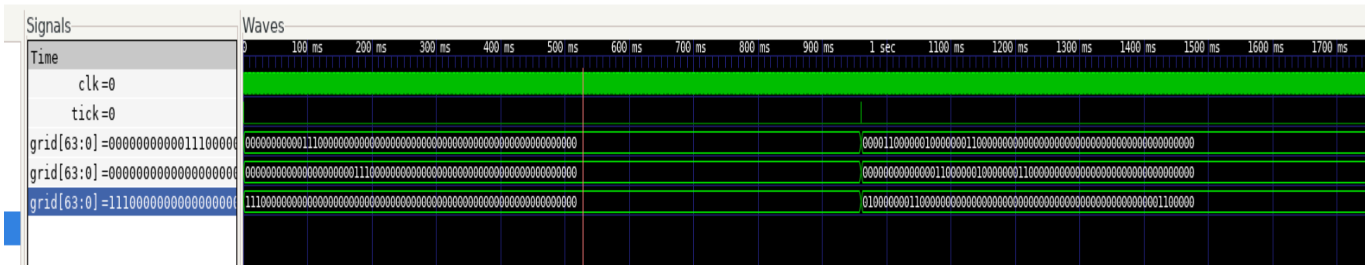
The implemented circuit is a hardware version of Conway's Game of Life on three independent 8×8 grids, corresponding to red, green, and blue color channels. Each grid is initialized with a different pattern, which exists perpetually while moving (gliders), with some occasional overlap between patterns (where more than one led for that pixel is lit up, changing color). The circuit operates on a 12 MHz clock, with a clock divider generating a 1 Hz tick to update the grids. Each tick triggers the evaluation of all cells in each grid simultaneously according to the standard Game of Life rules: a live cell with two or three live neighbors survives, a dead cell with exactly three live neighbors becomes alive, and all other cells die or remain dead. I implemented this using one game of life module, which was instantiated three times, for ease of understanding of code and simple testing.

The interesting part of this design is the optimization of the Game of Life computation. Rather than evaluating each cell individually using nested loops and arithmetic, the neighbor states are precomputed using bitwise shifts and wrap-around logic. This reduces the resource usage, allowing for multiple games to run on the same FPGA without issues.

Simulation results over a two-second period show the correct operation of the system. The tick signal pulses high once per second, triggering simultaneous updates across the three grids. The binary outputs of the red, green, and blue grids for both seconds show the correct readings and changes, confirming proper implementation of the Game of Life rules.



```
clk=0
tick=0
grid[63:0] =00000000000111000000   0000000000011100000000000000000000000000000000000000000000000000   0000110000001000000011000000000000000000000000000000000000000000
grid[63:0] =00000000000000000000   0000000000000000000001110000000000000000000000000000000000000000   0000000000000001100000010000000110000000000000000000000000000000
grid[63:0] =11100000000000000000   1110000000000000000000000000000000000000000000000000000000000000   0100000001100000000000000000000000000000000000000000000001100000
```