A Modular Single Cell for Conway's Game of Life

Jordan Lewis

Abstract

The cellular structure of Conway's Game of Life provides an excellent challenge in electronic design: to create a self-contained circuit that will perform the logic of a single cell, and display the result. These circuits will have the property that they can be connected in an orthogonal grid as in the Game of Life itself, allowing for a large simulation of the Game of Life simply by connecting many of these modular units.

1 Introduction

The Game of Life is a cellular automaton created by mathematician John Conway. It is describable with only four simple rules, yet the right starting states can generate surprisingly complex results. It is Turing-complete, and has been the subject of considerable study, both focusing on algorithms for simulating the automaton and on the varied kinds of patterns that can emerge from running it.

1.1 Rules

As a cellular automaton, the Game of Life runs on an infinite grid of two-dimensional squares, each of which can either be alive or dead. The function that decides what state a cell will be in on the next turn, given the current states of its 8 neighbors and itself, can be described with the following pseudocode.

GAME-OF-LIFE($n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8$, *cur-state*)
    $n = \sum_{i=1}^{8} n_i$
    if *cur-state* $== 1$ **and** $n \subset \{2, 3\}$
        return 1
    elseif *cur-state* $== 0$ **and** $n == 3$
        return 1
    elseif $n < 2$ **or** $n > 3$
        return 0

1.2 Goal: Implementation of a Single Cell

GAME-OF-LIFE gives us enough information to design logic that will output the state of a cell, given the state of its eight neighbors and its own state. It seems, therefore, that it should not be too much trouble to design a circuit that will implement this logic. By adding some extra machinery to store and display the one bit of liveness for a single cell, and a switch to set the liveness of the cell, we will have created a stand-alone cell for the Game of Life that can be connected with others of its ilk to form a larger simulation.

2 Design

2.1 Logic Design

The central problem in designing this circuit is the logic necessary to implement GAME-OF-LIFE. We could build a truth table that shows every combination of inputs and the resulting output, as below:

n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	cur	next
f	f	f	f	f	f	f	f	f	f
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
f	f	t	f	f	f	t	f	t	t
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
t	t	t	t	t	t	t	t	t	f

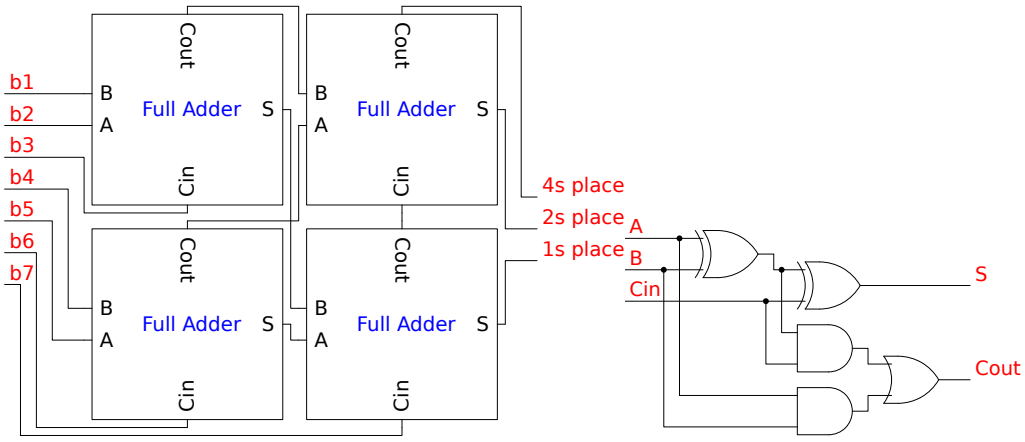
Clearly, listing the whole table would be of dubious value: since we have nine inputs, the table would have 2^9 rows. Synthesis from the canonical sum of products would need a prohibitively large number of gates, and Karnaugh maps begin to lose their ability to organize truth tables after only about four inputs.

GAME-OF-LIFE is not unstructured, though: at its core, it is simply trying to calculate the number of live adjacent neighbors. This is the same as the *Hamming weight* of the 8-bit word formed by the n_i arguments. It is a simple matter to calculate the Hamming weight of $2^n - 1$ bits by chaining together a bunch of full adders, as in the 7-bit Hamming weight calculator in Figure 1a. GAME-OF-LIFE requires calculating the Hamming weight of an 8-bit word, however. Creating a 15-bit Hamming weight calculator would work, but there are a few optimizations we can make to reduce the complexity of the circuit.

1. Since our function kills a cell if has four or more neighbors, we can safely not worry about combining both 4-bit sums with another full adder.

2. Since we only need 8 bits of the 15 bits available, we can replace the low-order full adders with half-adders. This is more clearly illustrated in Figure 2a.

Once the Hamming weight network is complete, we are left with the Hamming weight of the 8-bit neighbors word, represented with two 4s places, one 2s



(a) 7-bit Hamming weight calculator

(b) Full adder

Figure 1

place, and one 1s place. In addition, we must deal with the *cur-state* input bit. It is simple to compute the rest of Game-of-Life given these inputs, however:

Finish-GOL($4s\text{-}place_2$, $4s\text{-}place_2$, *2s-place*, *1s-place*, *cur-state*)

not-overcrowded = $4s\text{-}place_1$ **NOR** $4s\text{-}place_2$
at-least-3 = *2s-place* **AND** *1s-place*
2-and-alive = *2s-place* **AND** *cur-state*
alive-if-not-overcrowded = *2-and-alive* **OR** *at-least-3*
return *alive-if-not-overcrowded* **AND** *not-overcrowded*

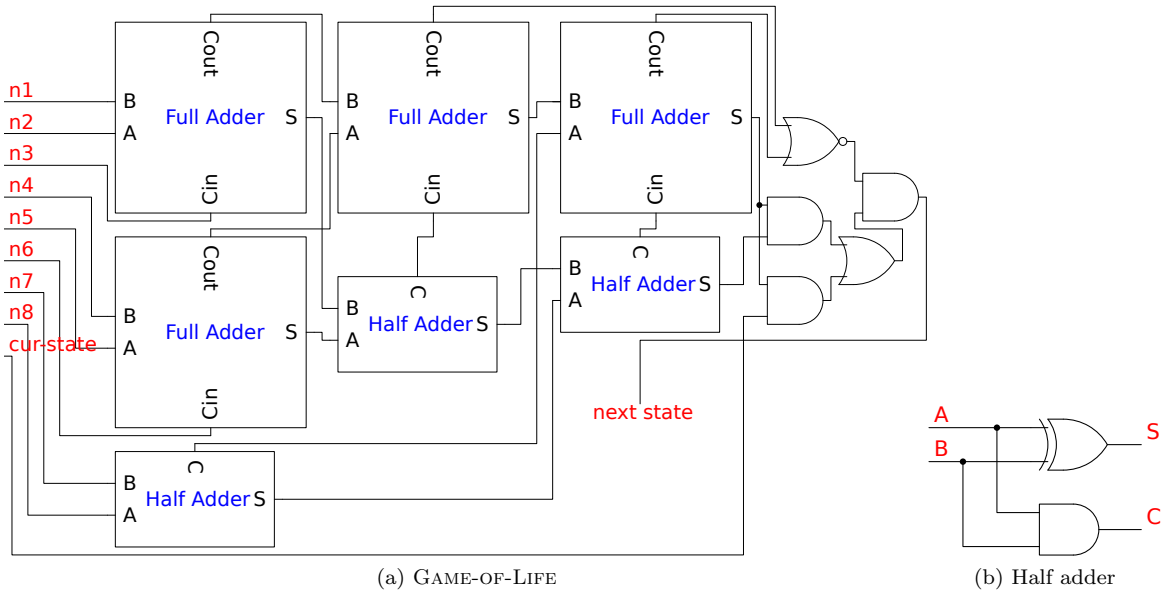
Implementing Finish-GOL requires only 5 gates, and is visible on the upper-right side of Figure 2a. Figure 2a is a complete implementation of Game-of-Life.

2.2 Other Design

With the core logic implemented, the only thing left to do is adding the machinery that will save and display the state of the cell, allow the user to manually set the state of the cell, and allow an external clock to tell the cell when to update.

Since all we are storing is the current state of the cell, a one-bit quantity, a single D-type flip-flop is all that we need. We want the user to be able to manually override the function and set the state to either on or off, so we will use a D-type flip-flop with $\overline{\text{set}}$ and $\overline{\text{reset}}$ inputs.

The $\overline{\text{set}}$ and $\overline{\text{reset}}$ inputs will be driven by a set of two switches: one SPDT switch (STATE) that controls what state the cell will be set to, and one normally-low push-button switch (PROGRAM) that will asynchronously set the cell's state to the state of STATE. PROGRAM is NANDed with STATE to get $\overline{\text{set}}$, and PROGRAM is NANDed with $\overline{\text{STATE}}$ to get $\overline{\text{reset}}$.



(a) Game-of-Life (b) Half adder

Figure 2

Once this is done, we complete the feedback loop by replacing the external *cur-state* input with the non-inverting output of the D-flop, and connecting the end output of the Game-of-Life circuit in Figure 2a to the D input of the D-flop.

Our circuit is complete (Figure 3), and in addition to the 8 neighbor state inputs, it now requires an external clock signal. At this point, we can replace our detailed schematic with a black box. By connecting these black boxes in a grid and adding an external clock, we can create an arbitrarily large simulation of Conway's Game of Life. An example 3x3 grid is reproduced in (Figure 4).

2.3 Design limitations

The major weakness of this design is the clocking setup. As it is, race conditions between the clock pulse and states updating will be common. The way to prevent this is to let the clock pulse propagate from one corner of the grid down by adding circuitry to each cell that, upon receving a clock pulse and updating its state, re-broadcasts the clock pulse to the cell below. The limitation of this model is that, to avoid cells receiving clock pulses from more than one source, we must give special circuitry to the top row of cells that resends the clock pulse to both the cell below *and* the cell to the right.

Another weakness is the difficulty of connecting many modules together, as is apparent from the messiness of Figure 4. A cleaner design might use a 3x3-cell grid as the fundamental unit. This would simplify the connection process by quadratically reducing the number of connections to be made.

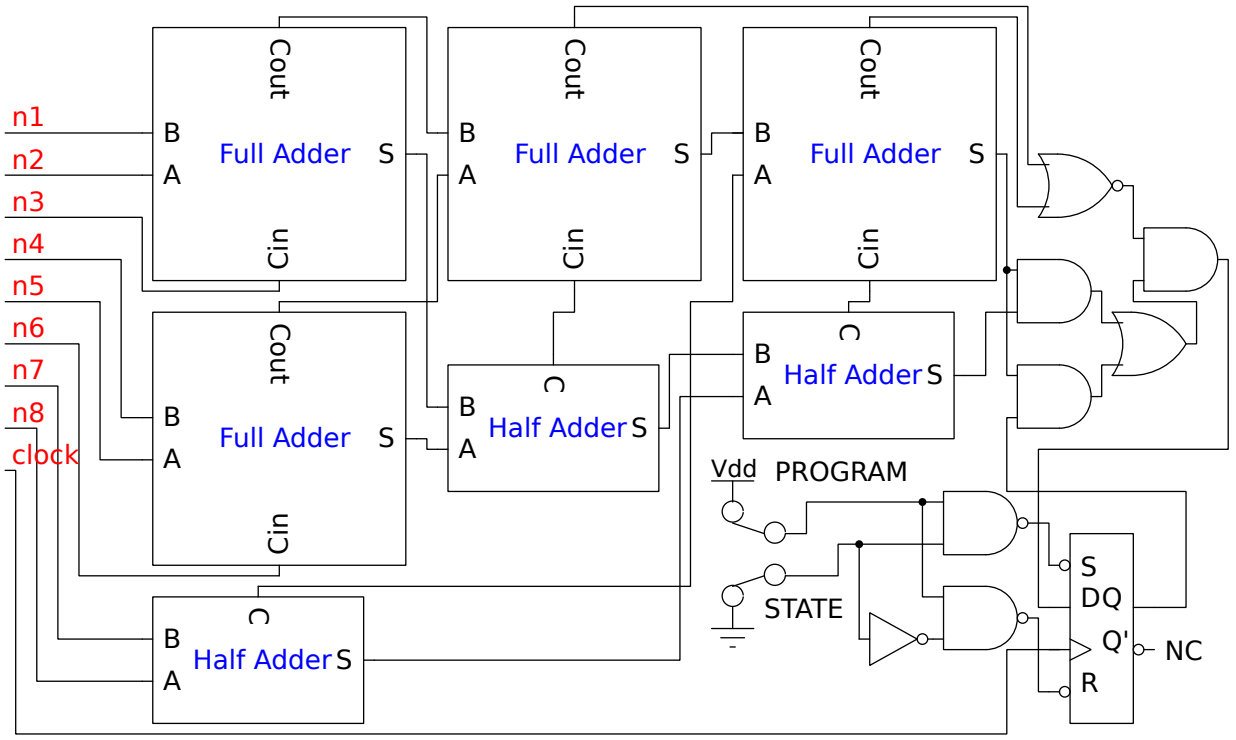


Figure 3: Complete single cell; PROGRAM switch high, STATE switch low

Finally, as the number of connected modules becomes greater, the effects of propagation time would become apparent. Since each cell depends only on the cells directly adjacent to it, however, propagation delay shouldn't pose a problem.

3 Conclusion

While effective for small grids, this solution does not scale well enough to extend to large simulations, due to fundamental issues with both the chosen atomic unit and the clocking scheme. A more nuanced, propagating clocking scheme, while more complex to implement, would remove the design flaw.

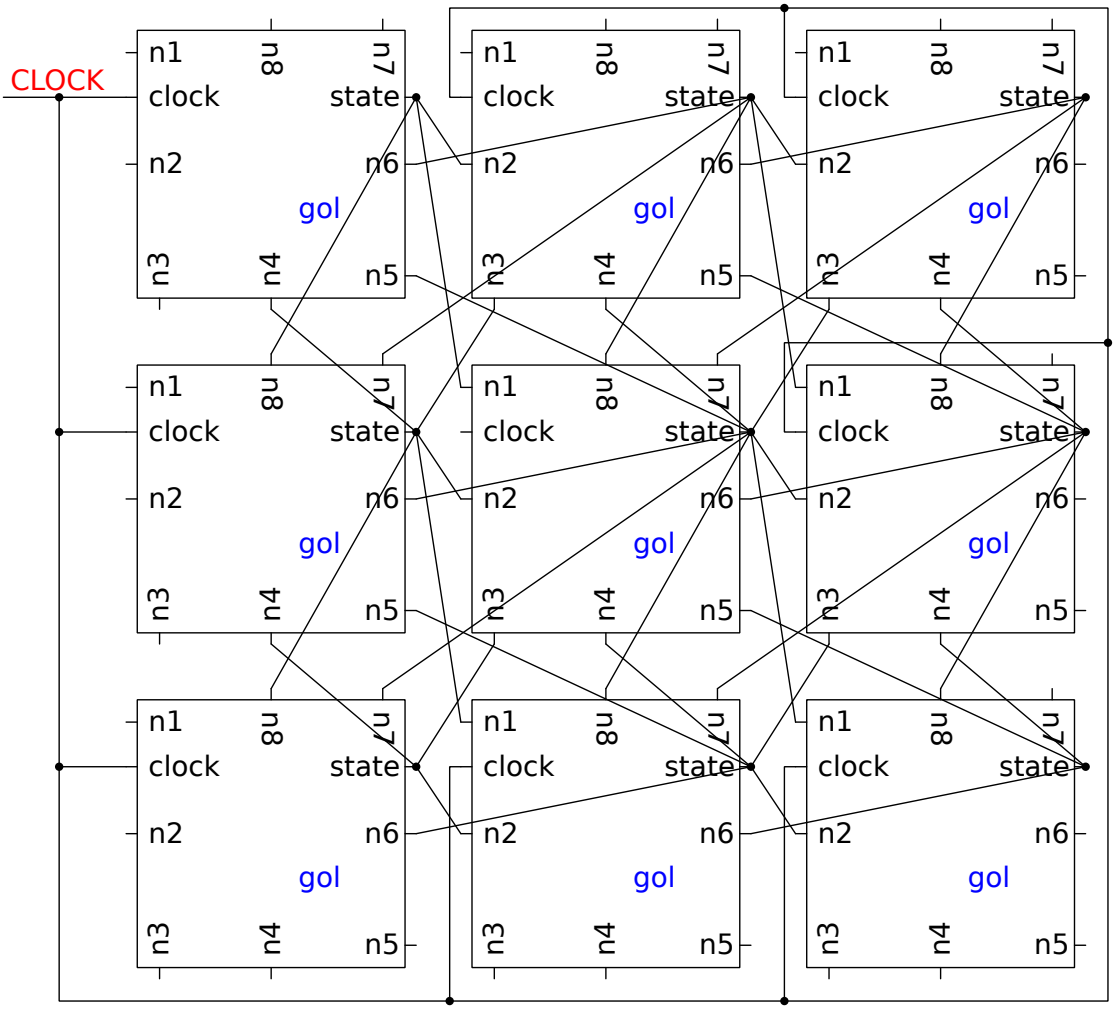


Figure 4: 3x3 array of single cells, clocked externally