

# Game of Life to Simulate Computers

Michael Cho

Seoul International School

*mycho2003@gmail.com*

## ABSTRACT

*It has previously been found and proven that the Game of Life can be used to simulate computers, and more generally, Turing Machines. In this paper, we investigate the construction of various aspects of a computer, like memory and logic, with particular attention on the practicality of each component's use. We demonstrate the feasibility of constructing a computer within the Game of Life by making a full adder.*

**Keywords:** Conway's Game of Life, Logic Gates, Turing Machine, Zero-player game, Computer Simulation

## 1. INTRODUCTION

Conway's Game of Life is a cellular automaton created by John Conway in 1970. It is a deterministic zero-player game, meaning that its change over time is purely decided by the original configuration of the game. The game itself consisted of a grid of squares that can either be populated or not, and initially configuring the game thus consists of toggling certain squares, or 'cells', from unpopulated to populated (or vice versa). The game is then run in "turns" governed by the following rules:

- Any populated cell with two or three populated neighbor cells (including vertical, horizontal, and diagonal directions) survives
- Any unpopulated cell with three live neighbors becomes populated.
- All other live cells die and become unpopulated.

The rules of this game lend themselves to a sort of simplistic model of life due to how unpopulated cells can become populated, as if by reproduction. The other rules, like how cells can

die from overpopulation or underpopulation in their neighbors, also are reminiscent of ecological interactions; as a result, the Game of Life is also researched in biology among other fields such as CS, mathematics, physics, and complexity science. The many interesting structures that have been discovered in the Game of Life also have made it the subject of many studies, often using it to make models of various phenomena.

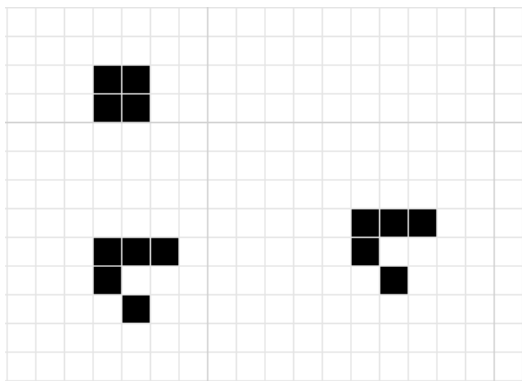
## 2. Game of Life for Computers

One of the many things that have been modeled in the game of life are computers, as well as computer-run programs such as games and calculators. The key to creating such programs in the game of life lies in the internal structure of computers.

### Turing Completeness

The concept of a Turing machine is, essentially, that of the most simple theoretical computer. It has a potentially infinite "tape" that serves as memory and stores input data, variables, and results, and is capable of moving a read/write head to read symbols from the tape and replace them with others. The specific actions of a Turing machine are determined by a Finite State Machine, which acts as its program and gives, for every possible combination of state and symbol, the new state, the new symbol to write, and the direction in which to move the head.

Then, the property of Turing Completeness can be applied to a system of data-manipulation rules, such as programming languages or cellular automata, if they can be used to simulate any Turing Machine. The Turing Completeness of such a construct can be shown through a proof that it can be used to simulate a Turing Complete system, or in other words, by showing that it can simulate a Turing Machine. It



was proven that the Game of Life was Turing Complete through its potentially infinite size and its capability for simple logic, and a Turing Machine was constructed in GoL by Rendell.

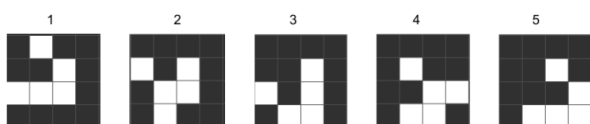
Due to the Turing Complete-nature of the Game of Life, we find that it is possible to construct logic and computers within the Game of Life.

### Memory

Firstly, to make a computer within any kind of system would require some method of storing memory. We find that, using a mechanism called “sliding block memory”, it is possible to store memory in a Game of Life computer in a fairly direct manner.

**Figure 1: Collision of 2 gliders and a block in sliding block memory.**

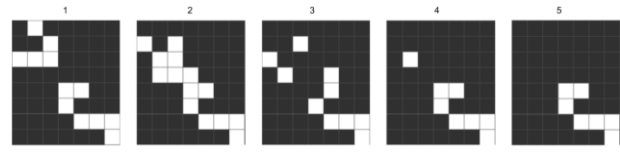
Through positioning a 2x2 block (which is a stable formation when not interacting with other cells; if there is only a 2x2 block, it does not change) and two gliders as shown above and having them collide, it is possible to translate the 2x2 block one unit diagonally downwards and to the right. It is also possible, through the collision of the block and



3 gliders, to move it one unit in the opposite direction. Thus, through considering the position of the 2x2 block to be the value of a stored variable, one may consider that the collision of 2 gliders

reduces the value of the variable by 1, while the collision of 3 increases the value by 1. Dean Hickerson's sliding block memory has been used in practice to construct a Universal Turing Machine, in Paul Chapman's URM.

### Modeling Logic Gates with the

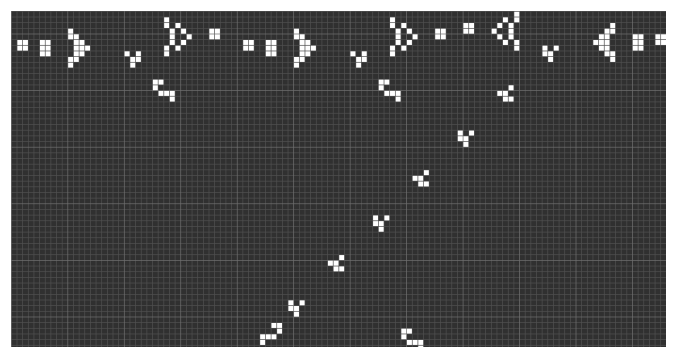


### Game of Life

The inner workings of a computer essentially boils down to two structures: variables and logic gates. A variable stores information, usually in the form of a 0 or 1. These variables can be toggled, or changed, and are how the computer converts information to display or a user interface. Logic gates (AND, NOT, OR... to name a few) are what processes these variables. The operations of a computer are created through these logic gates; for example, one can make addition using such logic gates and variables indicating quantities. Due to the deterministic nature of these logic gates, they can be modelled in the game of life (GoL) without much complication.

**Figure 2: Movement of a glider.**

One very common structure in the game of life is a glider. A glider is defined as any pattern that, over a certain period of moves, translates its position while maintaining its shape. Figure 2 above shows how one glider moves diagonally to the bottom-right, one unit every 4 generations. A



glider gun, then, is a pattern that endlessly shoots gliders. Furthermore, an eater is a pattern that, when colliding with a glider, returns to its original position as if “eating” the glider. As shown in Figure 3, an eater takes 4 generations to return to

its original configuration after collision with a glider (which disappears).

**Figure 3: An eater colliding with a glider.**

Using these structures, we may create logic gates in the game of life.

We may simulate variables using a glider gun that can be turned on or off; this toggling “switch” can be found in the form of an eater placed in front of the glider gun, so that if the variable is a 1, then the lack of an eater allows the glider gun to shoot gliders, and if the variable is a 0, then the eater blocks the path. This is important because gliders, when colliding at a 90 degree angle, can destroy each other without a trace.

The first logic gate to create is the NOT logic gate, which simply outputs a 1 if the variable is a 0 and outputs a 0 if the variable is a 1. This can be created by setting a constant, unblocked glider gun shooting gliders at a 90 degree angle from the variable glider gun. If the variable is 1, then the glider guns' gliders will collide, and if the variable is 0, then the constant gun will have an open pathway; thus, the output is represented by the presence, or lack thereof, of the constant gun's gliders.

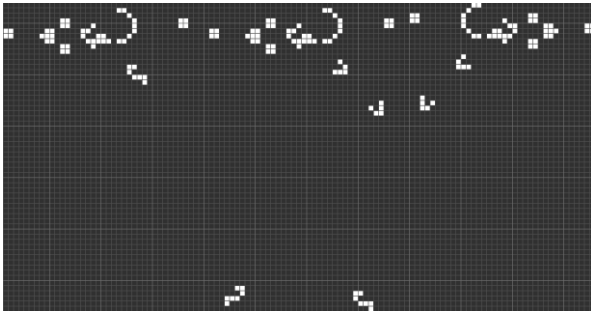
**Figure 4: AND gate with both variables set as 0. The bottom right eater does not collide with any gliders; thus, the output is 0.**

**Figure 5: AND gate with one variable set as 0, and the other set as 1.**

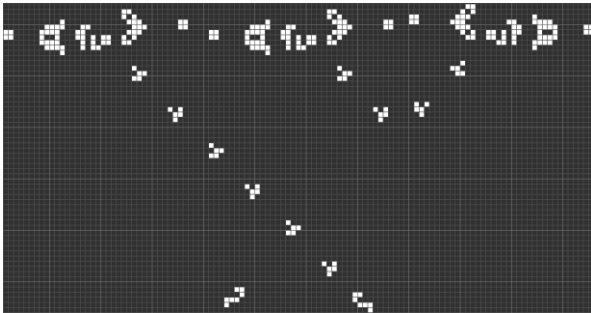
**Figure 6: AND gate with both variables set as 1; the constant glider gun is blocked, and the second variable gun causes the detector eater to collide with gliders. Thus, the output is 1, as expected.**

We may also create the AND logic gate, which notably involves two different variables. This is possible through setting a constant glider gun at a 90 degree angle with both variable guns, with run parallel. The output is found by the presence of the second variable gun's gliders, since it will only have an open pathway if the constant gun is “blocked” by the first variable gun's gliders. Shown in Figures 4, 5, and 6, the bottom right “detector” eater will only collide with gliders if both variable guns are on.

**Figure 7: OR gate with both variables set as 0; both constant guns cancel out, thus the bottom eater does not collide and the output is 0, as expected.**



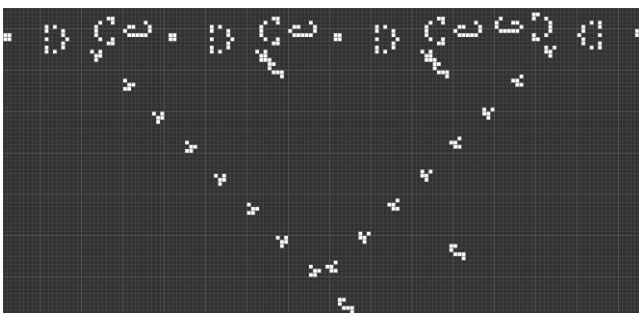
**Figure 8: OR gate with one variable set as 1; the variable gun "blocks" the constant gun, causing the detecting bottom eater to collide with the**



**other constant gun's gliders. Thus, the output is 1, as expected.**

**Figure 9: OR gate with both variables set as 1; same as Figure 8, the constant gun is cancelled and the bottom eater detects an output of 1.**

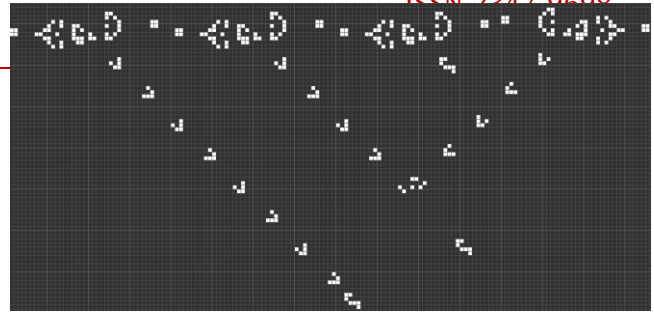
Finally, the OR logic gate can be modeled similarly to the AND gate, but adding another constant glider gun running parallel to the two variable guns, but on the other side of them from the perpendicular gun. The output is then determined by the parallel constant gun, since its path will only be blocked if the perpendicular constant gun is not blocked by the variable guns (in other words, only if both variables are 0). As



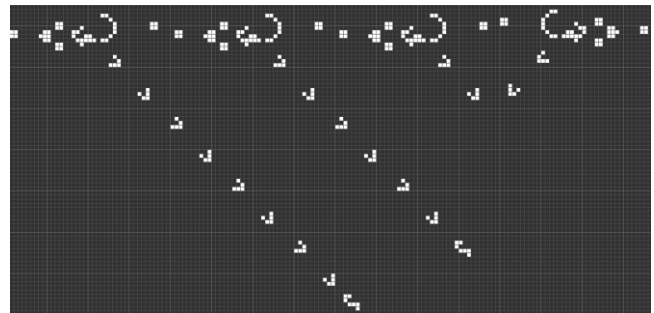
shown in Figures 7,8, and 9, as long as one of the variable guns are on, the constant gun on the right will be cancelled out, and the detecting eater will collide with gliders from the other constant gun.

### Overlapping Glider Streams

While using basic logic gates in order to create more complex circuits and computer structures, one



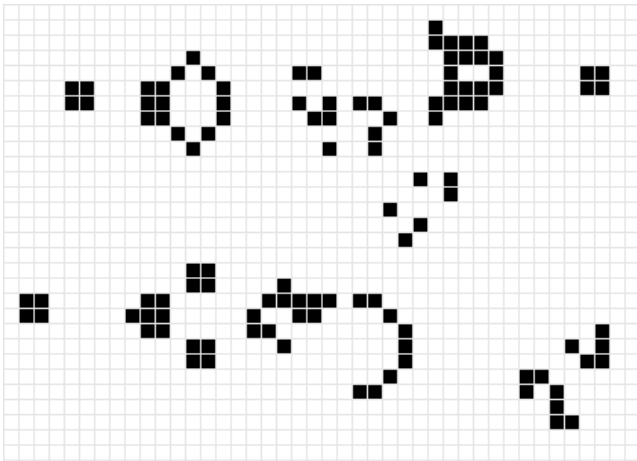
observed complication is the overlapping of gliders. Since many circuits (when observed two-dimensionally) have overlapped wires, we thus find that it is necessary to find a way for two glider streams to intersect without interacting with each other.



The most basic Gosper glider gun has a period of 30 ticks, or generations; however, because the speed of a glider is  $c/4$  (moves one diagonal unit every 4 generations), the space between adjacent gliders in a glider stream is too small to avoid collisions when two streams intersect. Thus, we find that it is much more practical to use a twogun, a glider gun of period 60.

**Figure 10: The twogun, a glider gun of period 60.**

Since the period of the glider gun is 60 ticks, by setting a timing difference of about 30 ticks between two intersecting glider streams, it becomes possible to have intersecting streams that do not interact (the timing difference does not necessarily have to be exactly 30 ticks; the period of 60 leaves enough room for some variation). Such differences in timing can be made by translating one glider gun 7 or 8 units in the opposite direction of the stream, which (since the period of the glider is 4) causes 28 or 32 ticks of timing difference. More exact manipulations can be carried out through having one glider gun be at a different point in its period. In no circuit will it be necessary for three wires to intersect simultaneously; thus, it becomes possible to express such circuits in the game of life through such manipulations in timing.



### 3. Making a Full Adder

To demonstrate the ability of the Game of Life to be used for a computer (or at the least, complex circuits), we decided to make a full adder.

**Figure 11: A circuit diagram of an adder, which was used for the construction of the following adder.**

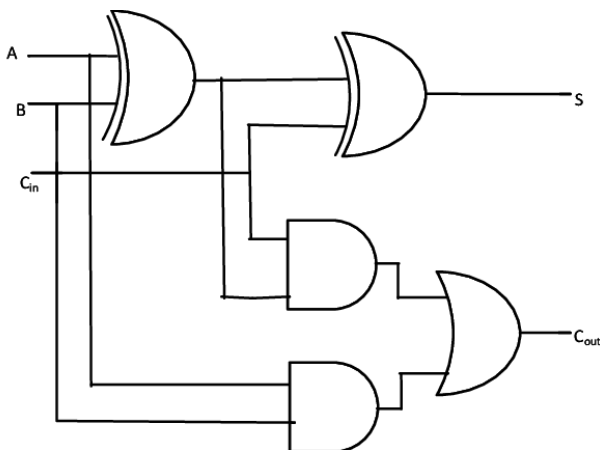
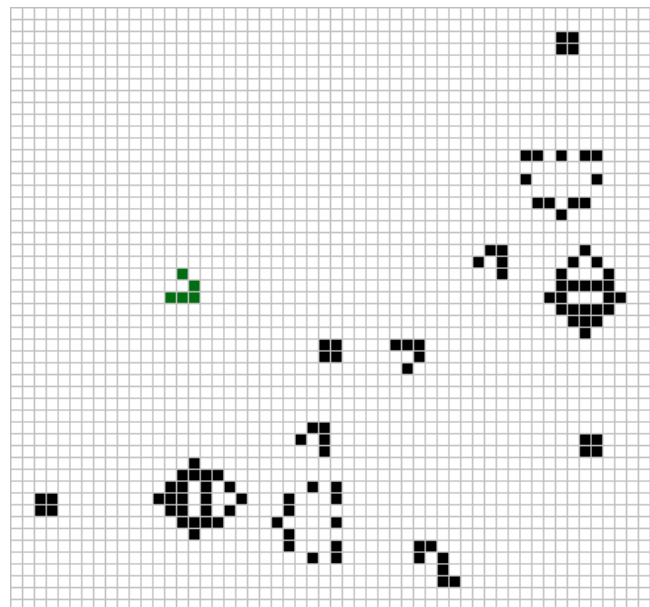
A full adder is a circuit which takes as input two numbers (which are to be added), as well as the input carry bit, and outputs a sum bit and a carry bit. For example, if the two numbers are 1 and 0, and the input carry bit is 1, then the output sum bit would be 1, while the output carry bit would be 0. It is possible to set multiple such full

adder circuits in sequence in order to carry out multi-bit addition. Figure 11 shows the circuit diagram for the adder, which is used to construct our adder.

### The XOR Gate

Though the AND and OR gates, which are used in the above circuit diagram, have already been shown to be fairly simple to make in section 2.3, it was necessary to create an XOR gate, which is more complex than the other logic gates shown above. At first, we attempted to create an XOR gate by combining AND, OR, and NOT gates, since the construction of all of these gates have already been found. However, at this point we noted another problem that was also relevant to the adder circuit: often, the same variables are used in more than one gate (meanwhile, the expression that represents XOR using the other gates uses each input twice). This could, in theory, be solved by simply having duplicate inputs for each value for every gate that uses it; however, not only would this take too much time to adjust variables, we believed it to be unfaithful to the idea and application of using the GoL as a computer.

**Figure 12: A glider duplicator that takes the green glider as an input and outputs two gliders.**



Thus, we found that using glider duplicators in the circuit was an effective solution to multiple gates needing a certain value as an input; by duplicating the glider stream, we were able to essentially create two glider streams representing the same variable, which could be the inputs to two different gates. The duplicator, shown in Figure 12, takes the green glider as input and outputs two gliders, one in the same direction and one to the upper right.

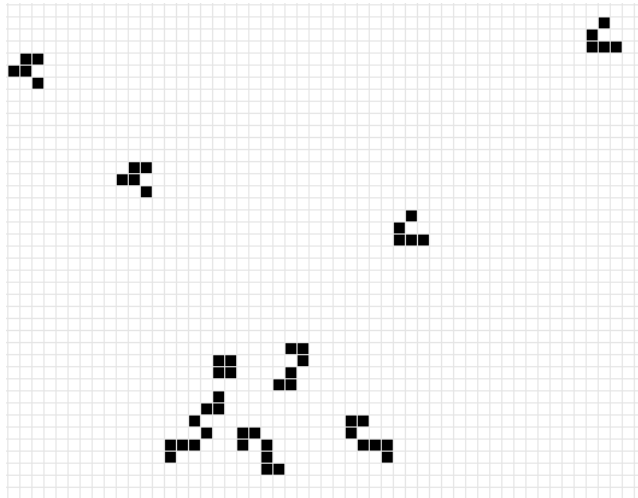
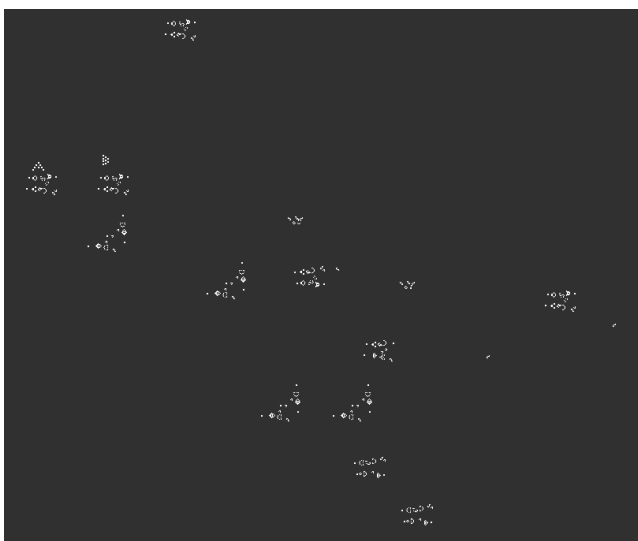


Figure 13: A 90 degree glider reflector.

Furthermore, the direction of the duplicated glider (and glider in general situations as well) could be manipulated using a glider reflector, removing any issues for the duplicated glider stream's direction being incompatible with a gate. Though the reflector in Figure 13 reflects gliders



coming from the upper right, simply reflecting the pattern horizontally or vertically based on the input stream direction allows for general use.

Going back to making the XOR gate, by using a combination of multiple logic gates as described above, we were able to create the following XOR gate:

Figure 14: An XOR gate made using 2 AND gates and one OR gate.

The XOR gate shown in Figure 14 is fully



functional, with input glider guns A and B indicated in the figure. However, when we attempted to create the adder circuit using this

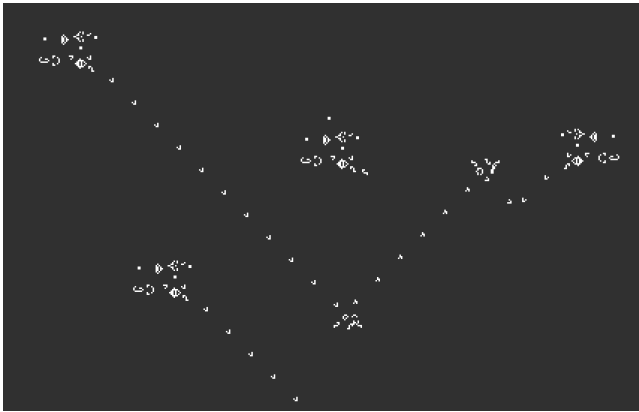


XOR gate, the gate's large size and logical complexity made it very impractical for use in complex circuits such as an adder, and especially



more logically intensive circuits.

Addressing this problem, we were able to come up with a far more elegant XOR gate much



smaller in size and less complex than the one shown above.



**Figure 15: The improved XOR gate, with input glider guns A and B.**

**Figure 16: The improved XOR gate, with A=1 and B=1.**

**Figure 17: The improved XOR gate, with A=0 and B=0.**

**Figure 18: The improved XOR gate, with A=1 and B=0.**

**Figure 19: The improved XOR gate, with A=0 and B=1.**

This XOR gate's main idea is to use glider reflectors to have the two input glider streams

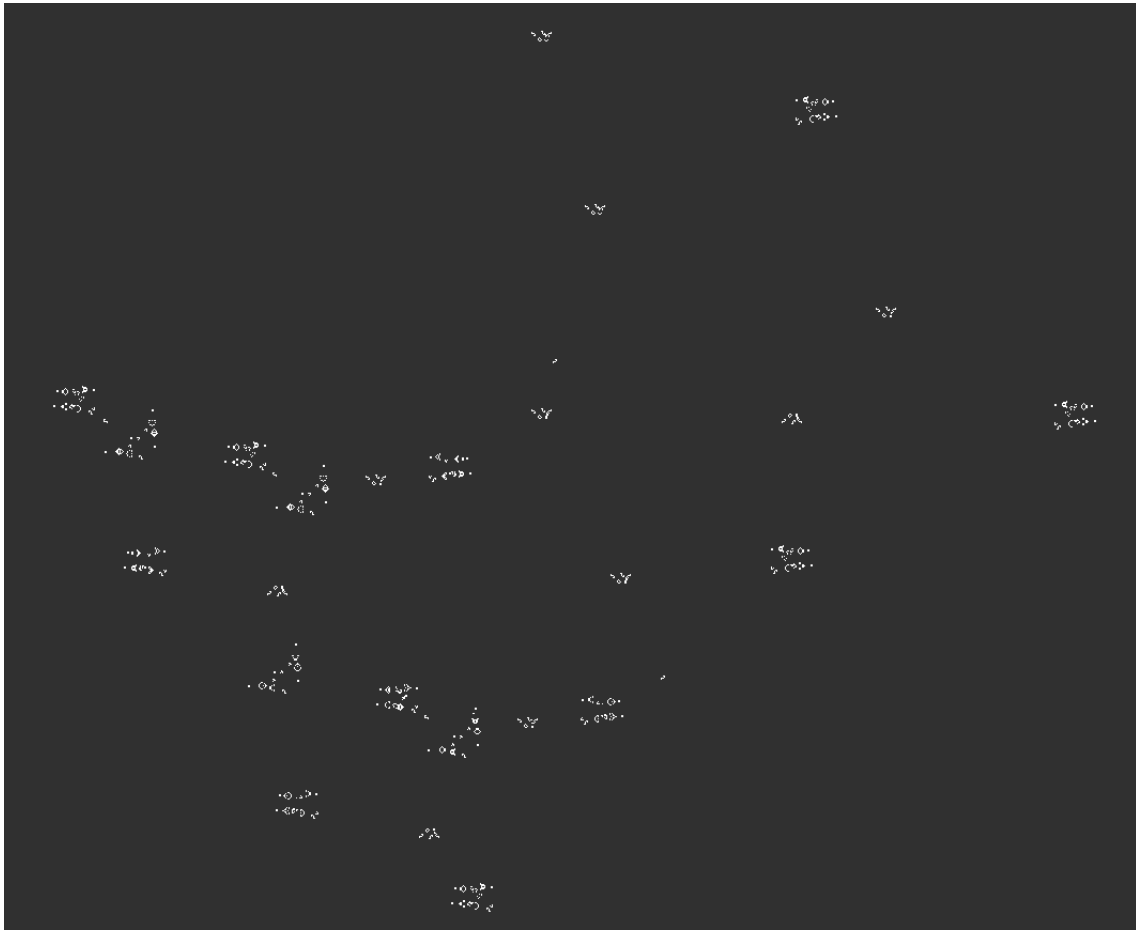
shown in Figure 15 collide with each other (if they are both "active", of course). In order to accomplish this, both A and B streams initially travel in the same direction. However, using a glider reflector, the A stream is made to collide with the B stream, as shown in Figure 16. Thus, if the streams are both active or both inactive, there will be no gliders after the collision point, like in Figures 16 and 17.

However, if one is inactive and one is active, we want the output stream to be traveling in the same direction. As a result, another reflector after the collision point so that if A is active and B is inactive, the resulting A stream travels in the same direction as the B stream.

Notice that a glider stream only emerges to the bottom-right (after the collision point) if the inputs should output 1 through an XOR gate, like in Figures 18 and 19; thus, the presence of a glider stream to the bottom-right is the output of the XOR gate.

However, note that the double-reflected A stream and the B stream are not exactly in the same location. Then, a constant glider gun is made to collide with both the double-reflected A stream or the B stream (whichever, if any, are not interrupted by each other's activity); this stream is NOT(A XOR B), since if the XOR gate should return true, the constant stream would collide with either A or B. Afterwards, it is a simple matter of adding another constant glider gun to collide with the first, which becomes NOT(NOT(A XOR B)), or simply (A XOR B) as a result.

Thus, we are able to create a far more compact, elegant XOR gate for use in the adder circuit.



**Figure 20: The completed adder circuit, using the improved XOR gate and other logical gates**

#### 4. Conclusion

Finding that Conway's Game of Life was Turing-Complete, we thus investigated the construction of a Turing Machine (and more specifically, a computer) in the Game of Life. Through research, we found not only a way to create memory but also logic using various glider mechanisms. Through a deeper examination of these glider mechanisms, and the creation of a compact XOR gate, we created a Full Adder circuit in the Game of Life. In future research, it would be interesting to investigate the construction of other components of a computer/Turing Machine, like loops, recursion, or other operations besides addition, in the Game of Life.

#### 5. REFERENCES

1. Dean Hickerson (March 1, 1990). Description of sliding block memory. Paul Callahan's Game of Life Miscellany. Retrieved on April 4, 2020, from <https://conwaylife.com/ref/lifepage/patterns/sbm/sbm.html>.
2. LifeWiki contributors. Sliding Block Memory. Retrieved April 4, 2020, from [https://www.conwaylife.com/wiki/Sliding\\_block\\_memory#cite\\_note-1](https://www.conwaylife.com/wiki/Sliding_block_memory#cite_note-1)
3. Paul Rendell (August 30, 2001). A Turing Machine in Conway's Game of Life. Retrieved April 4, 2020, from <https://www.ics.uci.edu/~welling/teaching/271fall09/Turing-Machine-Life.pdf>.
4. LifeWiki contributors. Glider Duplicator. Retrieved April 11, 2020, from [https://conwaylife.com/wiki/Glider\\_duplicator](https://conwaylife.com/wiki/Glider_duplicator)
5. LifeWiki contributors. P4 Bumper. Retrieved April 11, 2020, from [https://conwaylife.com/wiki/P4\\_bumper](https://conwaylife.com/wiki/P4_bumper)
6. Jean-Philippe Rennard. Implementation of Logical Functions in the Game of Life. Retrieved March 28, 2020, from <https://www.renard.org/alife/CollisionBasedRennard.pdf>
7. Shyam Akashe (August, 2013). Block Diagram of Basic Full Adder Circuit. Retrieved April 18, 2020, from [https://www.researchgate.net/figure/Block-Diagram-of-basic-full-adder-circuit\\_fig1\\_260632359](https://www.researchgate.net/figure/Block-Diagram-of-basic-full-adder-circuit_fig1_260632359)