Lecture 8: Logic Gates and Self-Simulation in the Game of Life

Modelling Complex Systems (Spring 2026)

Jordi-Lluís Figueras

Matematiska Institutionen, Uppsala Universitet

April 21, 2026

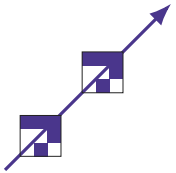
Prelecture discussion

Before Lecture 8, run `gameOfLifeInteractive.py` again, but now watch it as a signal-processing device rather than as a pattern toy.

- Launch a glider and ask whether it could represent a binary symbol.
- Look for collisions where a glider is destroyed, delayed, or redirected.
- Ask which stable objects could store a bit for a long time.

Questions to reflect on:

- If a glider means “1,” what should “0” mean?
- Can collisions implement the Boolean operations AND, OR, and NOT?
- What else is needed beyond logic gates if we want a full computer?



glider as moving signal

Lecture goals

In this lecture we will:

- reinterpret the Game of Life as a computational medium;
- encode binary information by timed glider streams;
- explain how collisions implement NOT, AND, and OR;
- connect collision-based logic to the idea of a universal computer;
- explain why a computer built inside Life can simulate Life itself.

Welcome to the Matrix

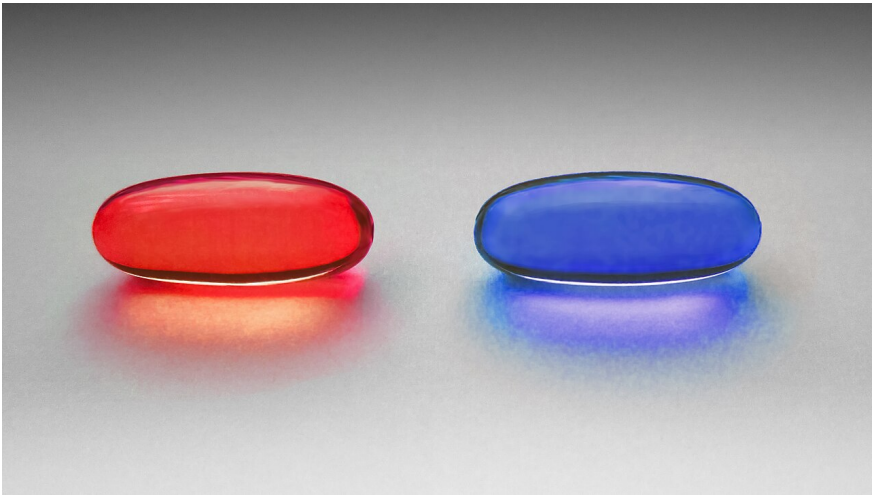


Image source: W.carter, *Red and blue pill*, Wikimedia Commons, CC BY-SA 4.0.
`https://commons.wikimedia.org/wiki/File:Red_and_blue_pill.jpg`

Rick and Morty

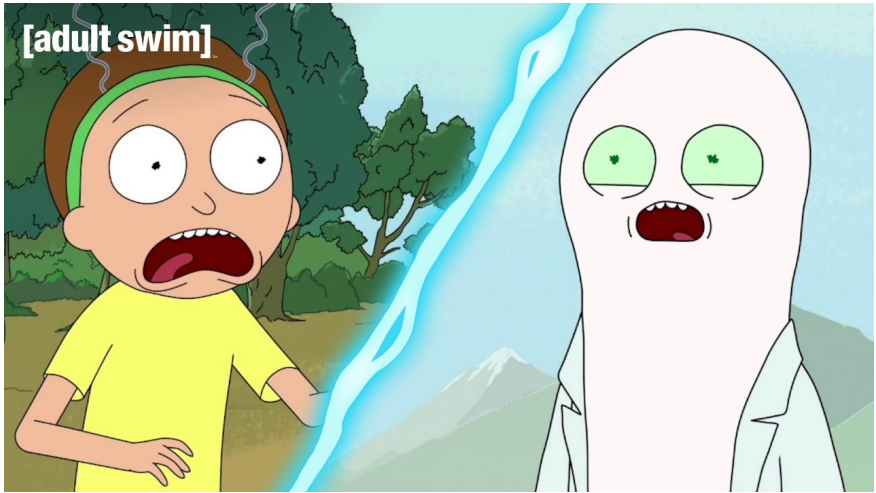


Image source: official *adult swim* YouTube thumbnail for *The Teenyverse Inside Rick's Miniverse | Rick and Morty | adult swim*, from the episode *The Ricks Must Be Crazy*. Open in browser

Our computers

How does a program work?

Question for the class:

How does a program actually work?

When you run a piece of code, what is really happening inside the machine?

Main components of a code

At a conceptual level, most programs are built from a few recurring ingredients:

- **input:** data enters the program;
- **variables and data structures:** information is stored and organized;
- **operations:** values are transformed by arithmetic and logical instructions;
- **control flow:** conditionals and loops decide what happens next;
- **output:** the program returns or displays a result.

At the machine level, all these pieces are still realized by operations on bits.

What is a computer made of?

At the physical level, an ordinary computer is not made of "Python," "apps," or "files." It is made of electrical devices that process and store binary signals.

- voltages encode the two logical values 0 and 1;
- transistors are combined into circuits;
- those circuits implement elementary logical operations;
- memory elements store bits between successive operations.

So when we say that a computer "runs a program," we really mean that a huge physical network of tiny switches is evolving in a controlled way.

The basic logical gates

The classical building blocks are logic gates such as AND, OR, and NOT.

Gate	Inputs	Output rule	Meaning
AND	A, B	$A \wedge B$	true only if both are true
OR	A, B	$A \vee B$	true if at least one is true
NOT	A	$\neg A$	reverses the bit

At the hardware level, these are *actual electric circuits*. Their outputs are determined by the physical behavior of currents and voltages in transistor networks.

All code boils down to this

No matter how sophisticated the software looks,

- high-level code is compiled or interpreted into simpler instructions;
- those instructions become elementary machine operations;
- those machine operations are realized by logic gates acting on binary signals.

In this precise sense, all code boils down to:

logic gates

This is the point of comparison for the Game of Life: if Life can realize logic in a sufficiently rich way, then it can realize computation.

A very abstract computer: the Turing machine

To understand the essence of computation, mathematicians use an extremely stripped-down model called a Turing machine. It is not meant to resemble a laptop physically. It is meant to capture the minimum ingredients needed for computation. In its simplest form, a Turing machine has:

- an infinite tape divided into cells;
- in each cell, one symbol chosen from a finite alphabet (for example blank, 0, and 1);
- a head that reads and writes one cell at a time;
- a finite list of internal states;
- a rule telling it what to write, where to move, and which state to enter next.

The tape is infinite in length, but each cell has only finitely many possible contents.

Where are the input, output, and program?

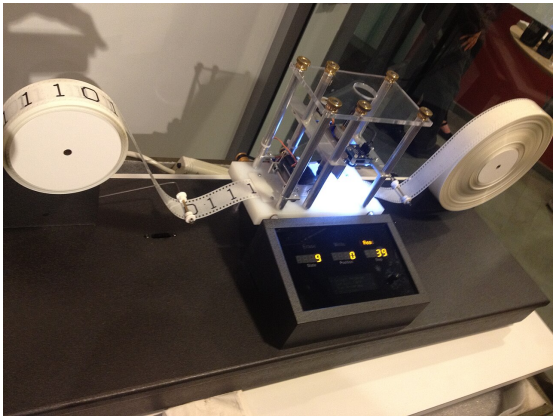
For a Turing machine, it is helpful to separate data from control:

- the input is the initial content of the tape;
- the output is the tape content when the machine halts;
- the head reads, writes, and moves left or right;
- the internal state remembers which stage of the computation we are in;
- the finite table of transition rules is the program.

So, in a compact slogan:

$$\text{tape} = \text{data} \qquad \text{head} + \text{state} = \text{control} \qquad \text{rules} = \text{program}$$

A physical Turing machine model



Physical model of a Turing machine.
Source: GabrielF, "Model of a Turing machine," Wikimedia Commons, CC BY-SA 3.0.

Example 1: write a single 1 and stop

Suppose the tape is initially blank and the machine starts in state q_0.

One possible rule is:

Current state	Symbol read	Action	Next state
q_0	blank	write 1, stay put	halt

So the machine turns

```
... _ _ [_] _ _ ...      into      ... _ _ [1] _ _ ...
```

This is trivial, but it already shows the basic pattern: read, write, move, change state.

Example 2: move right until the first blank

Now imagine the tape contains a block of 1's and we want the head to move to the first empty cell after that block.

Rules:

Current state	Symbol read	Action	Next state
q_0	1	move right	q_0
q_0	blank	halt	halt

For instance,

```
...  _ [1] 1 1 _ _ ...   ⟹   ...  _ 1 1 1 [_] _ ...
```

The content of the tape does not change; the machine is only searching for a location.

Example 3: append one more 1

If we combine the previous idea with one extra rule, the machine can extend a unary string.

Rules:

Current state	Symbol read	Action	Next state
q_0	1	move right	q_0
q_0	blank	write 1, halt	halt

So

```
... _ [1] 1 1 _ _ ...   ⟹   ... _ 1 1 1 [1] _ ...
```

Interpreting the string 111 as the unary number 3, this machine computes $3 \mapsto 4$.

Why is it the most basic type of computer?

A Turing machine is one of the most abstract models of a computer because almost everything non-essential has been removed.

- No screen.
- No processor in the modern engineering sense.
- No programming language syntax.
- Only symbols, local rules, and symbols written on a tape.

Yet this is enough to perform arbitrary algorithms. So the Turing machine is often taken as the minimal mathematical notion of a computer.

Why Turing machines matter here

If a system can simulate a Turing machine, then it can in principle carry out any algorithmic computation. This is why Turing machines matter in this lecture:

- they give a precise benchmark for computational universality;
- they let us replace the vague phrase “acts like a computer” by a mathematical statement;
- they connect logic gates to the formal notion of computation.

So later, when we say that the Game of Life can emulate computation, we really mean that it can emulate a machine of Turing type.

From patterns to signals

Why the Game of Life is more than a pattern generator

In the previous lecture, the Game of Life already showed three ingredients that matter for computation:

- **mobile patterns:** gliders can carry information across the grid;
- **stable patterns:** still lifes and eaters can absorb or hold information;
- **repeatable interactions:** carefully timed collisions have reliable outcomes.

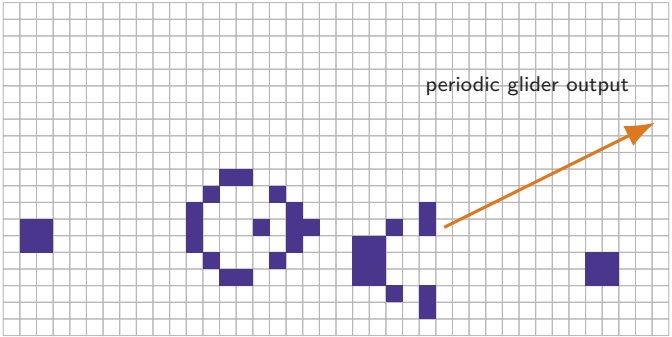
This is exactly the type of structure needed in digital computation:

Key idea

Bits must be representable, transmitted, transformed, and stored.

The remarkable fact is that Conway's local rule is rich enough to support all four tasks.

The Gosper glider gun



The Gosper glider gun is a finite Life pattern that repeatedly emits gliders. It is one of the key devices that turns Life from a pattern toy into a signal-generating medium.

(Do simulation)

Encoding a bit in Life

The simplest encoding is:

Logical value	Physical realization in a lane	Interpretation
1	a glider arrives at the scheduled time	signal present
0	no glider arrives at that time	signal absent

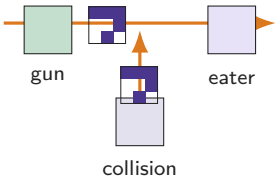
Important remarks:

- the meaning of "1" and "0" depends on a *chosen time window*;
- therefore computation in Life is both spatial and temporal;
- synchronization matters: a glider arriving too early or too late may encode the wrong symbol.

The basic components of a Life circuit

- **glider guns** generate periodic signal streams;
- **eaters** remove unwanted gliders without being destroyed;
- **reflectors and conduits** redirect signals;
- **collision sites** implement logic.

At the lecture-note level, we draw these circuits schematically, but every component corresponds to actual known Life constructions.



Computation is not attached to a single shape

In electronics, an AND gate has a standard circuit symbol. In Life, the situation is different:

- there is no unique canonical geometry for a gate;
- many different pattern collections can realize the same Boolean function;
- what matters is the input-output behavior, not a specific shape.

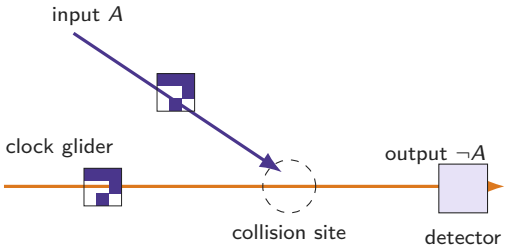
So when we draw a Life gate, we should read it as:

Operational viewpoint

"Here is a collision-based arrangement of real Life signals and stable objects whose effect is this Boolean operation."

Logical gates in Life

Actual NOT gate in Life: glider annihilation

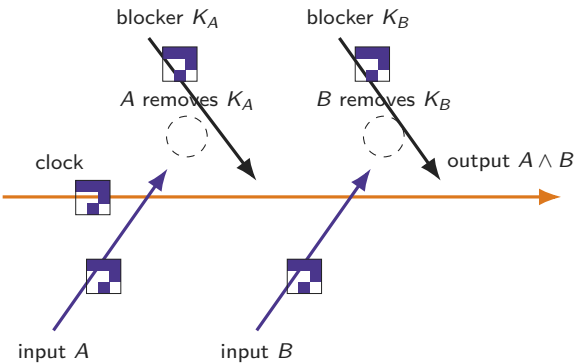


If the input glider is present, it meets the clock glider and both are destroyed. If the input glider is absent, the clock glider reaches the detector and represents output 1.

A	$\neg A$
0	1
1	0

This is an actual Life mechanism: negation is implemented by suppressing a periodic reference signal.

Actual AND gate in Life: remove both blockers or the output dies



The clock glider wants to travel to the output lane.

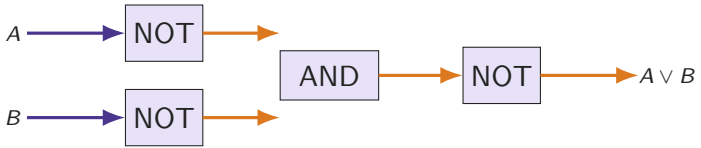
- If $A = 0$, blocker K_A survives and kills the clock.
- If $B = 0$, blocker K_B survives and kills the clock.
- Only if $A = 1$ and $B = 1$ are *both* blockers removed.

A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

Actual OR gate in Life: compose known Life modules

The easiest reliable way to obtain OR is often to build it from actual Life subcircuits we already understand:

$$A \vee B = \neg(\neg A \wedge \neg B).$$



Interpretation

This is still an *actual Life OR gate*: each box is a real glider-based Life subcircuit, and the whole composition has the OR truth table.

Why these gates are enough for combinational logic

Once we can build AND, OR, and NOT, we can realize any Boolean expression:

- XOR and NAND can be composed from them;
- adders, decoders, comparators, and multiplexers then follow;
- more elaborate control circuits can be assembled by combining simpler gates.

For example,

$$A \oplus B = (A \vee B) \wedge \neg(A \wedge B).$$

So at the purely logical level, the Game of Life can already reproduce the algebra used in ordinary digital circuits.

External video example

For a concrete visual demonstration of Game-of-Life logic circuits, including NOT, AND, and OR gates, see:

Open YouTube example

What should we remember from this lecture?

- In Life, a binary symbol can be encoded by the presence or absence of a glider in a timed lane.
- Actual Life circuits implement NOT by suppressing a reference glider, AND by requiring two blockers to be removed, and OR by composition of real Life subcircuits.
- These logic constructions can be organized into larger computational devices.

Suggested bibliography

For students who want to read more:

- M. Gardner, "Mathematical Games: The fantastic combinations of John Conway's new solitaire game 'life'," *Scientific American*, 223(4), 120–123, 1970.
- E. R. Berlekamp, J. H. Conway, R. K. Guy, *Winning Ways for Your Mathematical Plays*, Vol. 2, A K Peters, 2003.
- A. Ilachinski, *Cellular Automata: A Discrete Universe*, World Scientific, 2001.
- M. Mitchell, *Complexity: A Guided Tour*, Oxford University Press, 2009.
- P. Rendell, *Turing Machine Universality of the Game of Life*, Springer, 2016.

Acknowledgment

These slides were prepared with the assistance of OpenAI Codex / ChatGPT 5.4 as a drafting and editing tool.
All mathematical, pedagogical, and course-specific content remains under the responsibility of the lecturer.