**MINISTRY OF SCIENCE AND HIGHER EDUCATION OF THE RUSSIAN FEDERATION**

**FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION OF HIGHER EDUCATION**

**"NOVOSIBIRSK NATIONAL RESEARCH UNIVERSITY**

**STATE UNIVERSITY"**

**(NOVOSIBIRSK STATE UNIVERSITY, NSU)**

**09.03.01 - Informatics and Computer Engineering**

**Focus (profile): Software Engineering and Computer Science**

**TERM PAPER**

**Martynov Bodgan Alekseevich**

**Belov Danil Valerievich**

**"Conway’s Game of Life"**

Table of contents

[Introduction 3](#_Toc165143762)

[Problem Statement 4](#_Toc165143763)

[Hardware 5](#_Toc165143764)

[Software 6](#_Toc165143765)

[User Guide 7](#_Toc165143766)

[Conclusion 8](#_Toc165143767)

# Introduction

We have chosen and made a “Conway’s game of life” as project-game using Logisim and CdM-8.

The Game of Life, often referred to simply as Life, stands as a testament to the genius of British mathematician John Horton Conway, who introduced it in 1970. This remarkable cellular automaton operates independently, devoid of player interaction, where its progression hinges solely on the initial state provided, requiring no external input thereafter. Engaging with the Game of Life involves crafting an initial configuration and then observing its organic evolution.

Rules:

The universe of the Game of Life is an infinite, two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, live or dead. Every cell interacts with its eight neighbors, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

1. Any live cell with fewer than two live neighbors dies, as if by underpopulation.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

# Problem Statement

We have made 2 changes from the basic technical task :

1. The editor of the rules of “the game of life”. User can change rules …
2. Editing the field via the processor.

First of all, we have made work plan and divided the project into subtasks :

1. Divide project into Logisim part and Assembler (CdM8) part.
2. Realise Logisim part
3. Realise Assebmler part
4. Connect both parts
5. Check for bags and fix it
6. Optimise it

As mentioned earlier, our project has two parts : Logisim and Assembler (CdM8) .

Logisim :

The Logisim part of project will be responsible for main functions of game :

* Display
* System of generation cells
* Changing rules

Assembler (CdM8) :

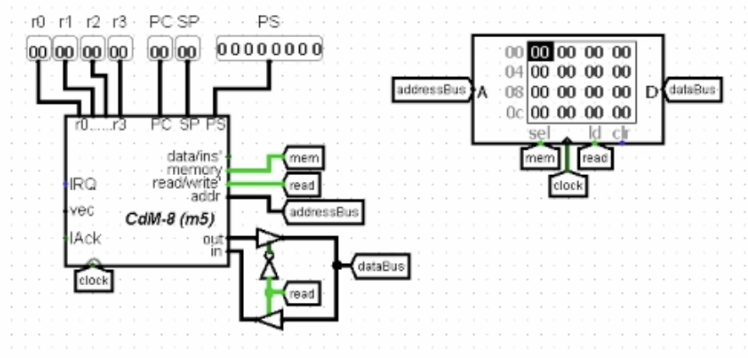
The CdM8 part of the project will be responsible for some functions

* Management - whether it is minimized, its current state and the ability to pause it.
* The control of four cursor keys - arrows down up left and right, pause and status buttons - invert, set death and set life.

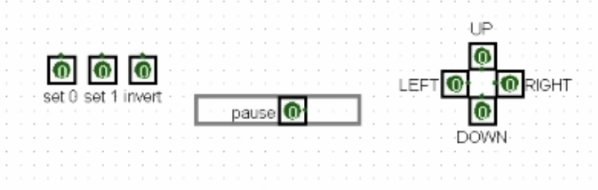
# Hardware

Main.circuit

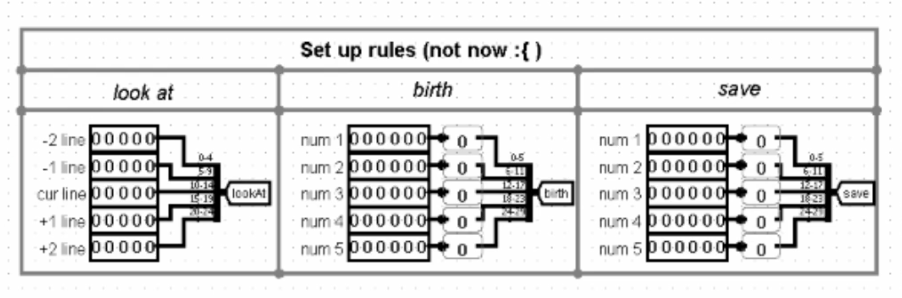
Let's start with the main file. First of all, we have made a processor with memory for further work, as in Coco IDE.



This is game management :

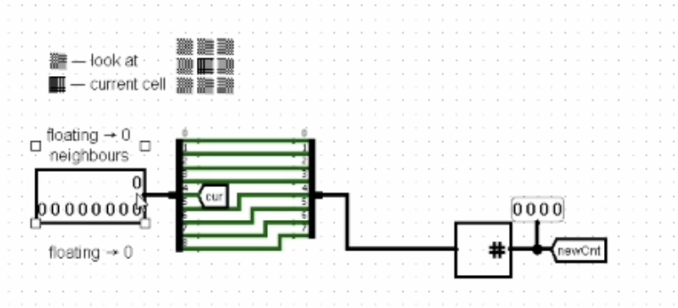


This scheme sets the rules of the game :

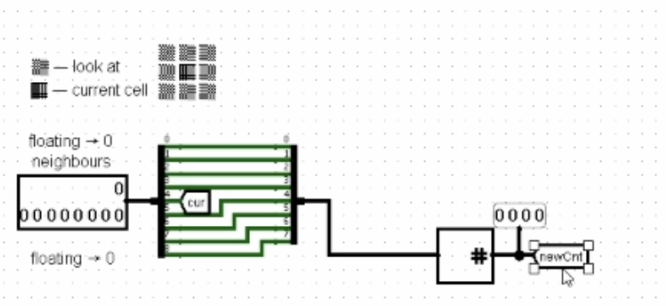


Alive.circuit

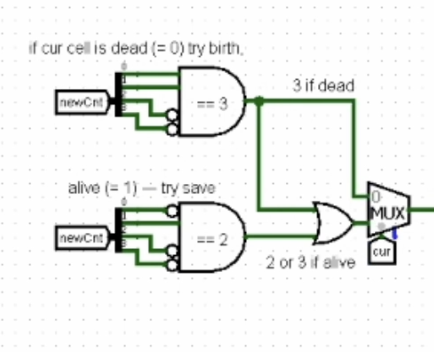
The next important file is AliveSTD. This file is responsible for whether the cell will live in the next iteration. To begin with, we serve the neighbors

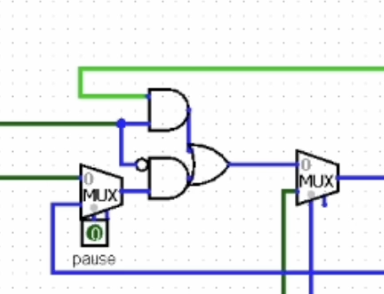
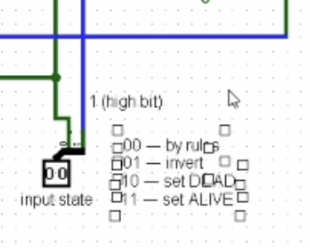


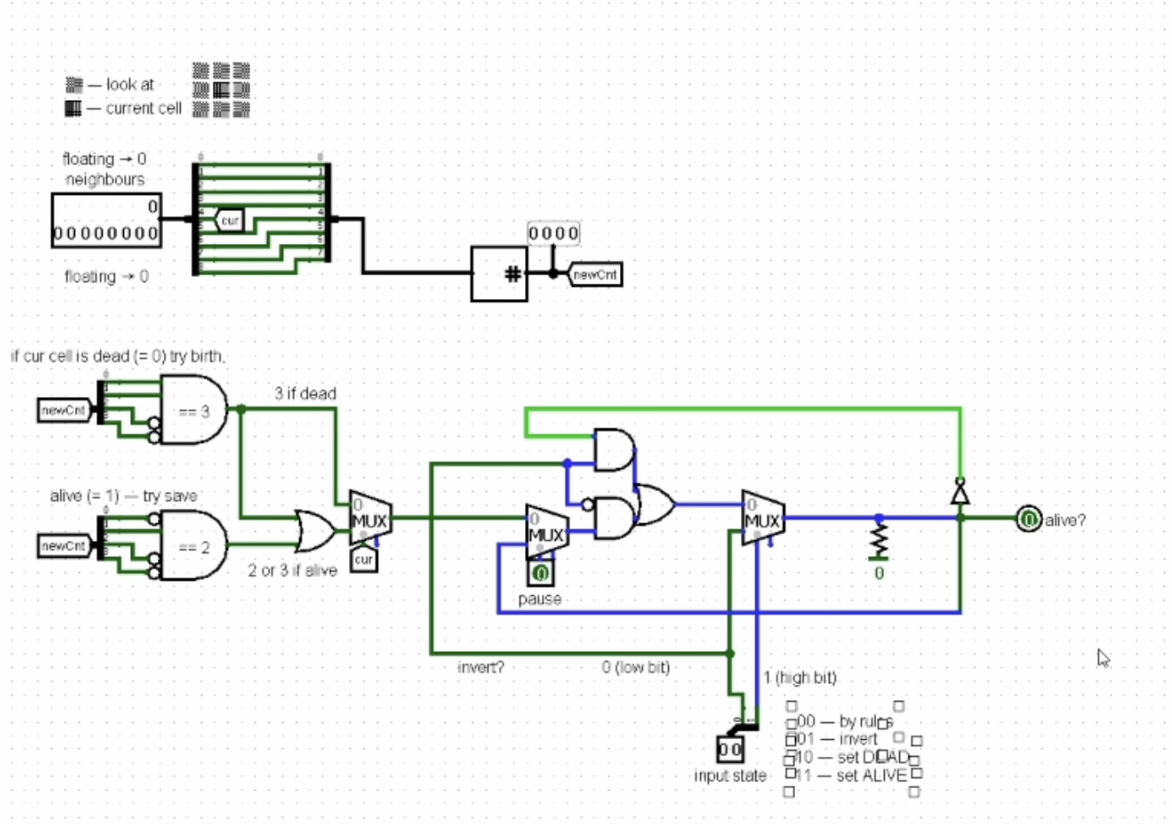
We count how many neighbors we have got :



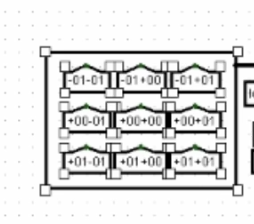
We check whether the cell will live according to the standard rules or not.



Next, the diagram below is responsible for choosing whether we want the next iteration or to change the cell manually.

And as a result, it outputs a value based on the result of all previous schemes

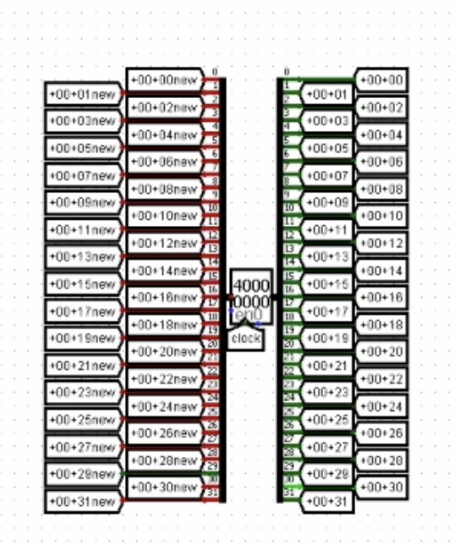
Line.circuit

After we get the value from the AliveSTD file, we pass the neighbors.

And we issue a new value



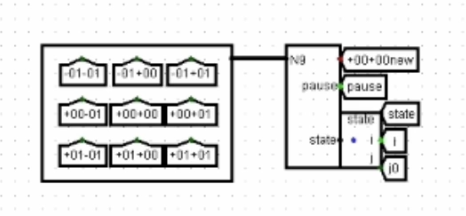
In this scheme, one register is stored, which is responsible for all cells in line. In the first versions of our project, there were 32 registers in this scheme, later we optimized it into one register to make it work faster.

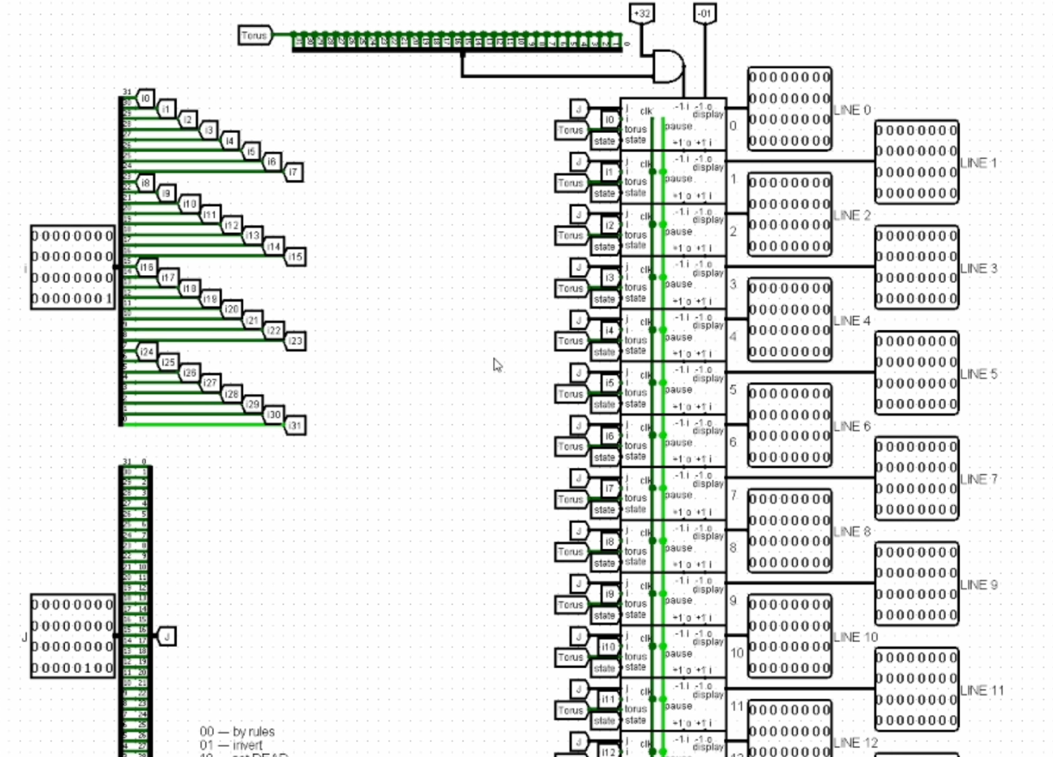


Matrix.circuit

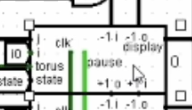
The next important file of our project is called MatrixSTD.

These 32 lines turn into a matrix.

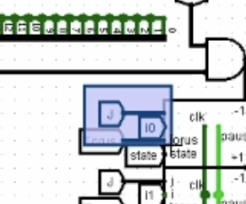




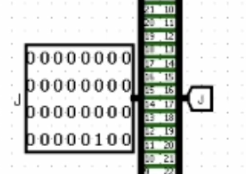
Now let's talk in more detail about the matrix. This is how one line in the matrix looks like



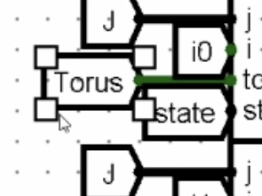
Here are two controls



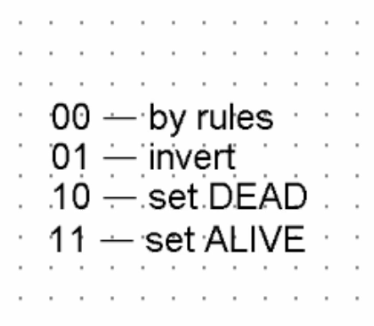
‘I’ tells you where the line cursor is now . ‘J’ passes a 32 bit number on which column it is located



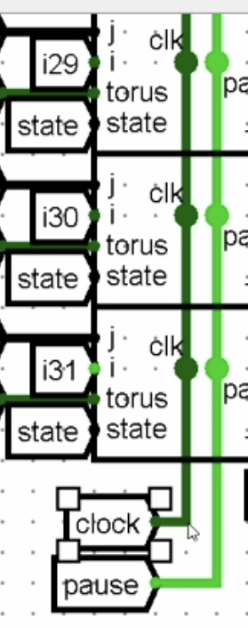
Torus is responsible for whether the field is collapsed or not, for example, as in the snake game. If the cell has gone to the left of the field, it will return to the right side of the field. To put it more simply , torus is the cyclicity of the field .



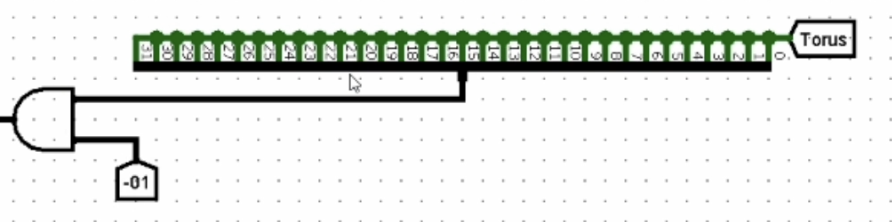
State are the states that we want from the cursor : old state , set to zero , set to one



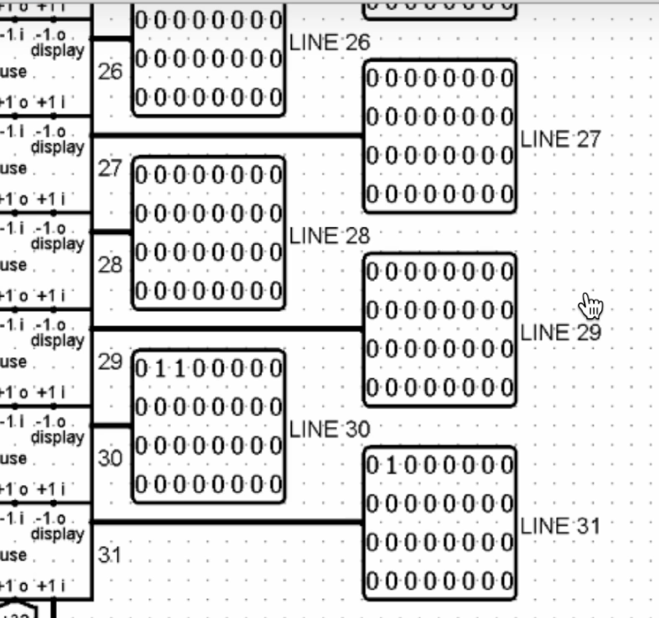
The Clock line is responsible for whether there is a pause or not. That is , whether there will be an iteration now or not .



And if the cell has turned to the end of the field - torus , then we reset all the cells , make them dead.



And we display everything using these diagrams.

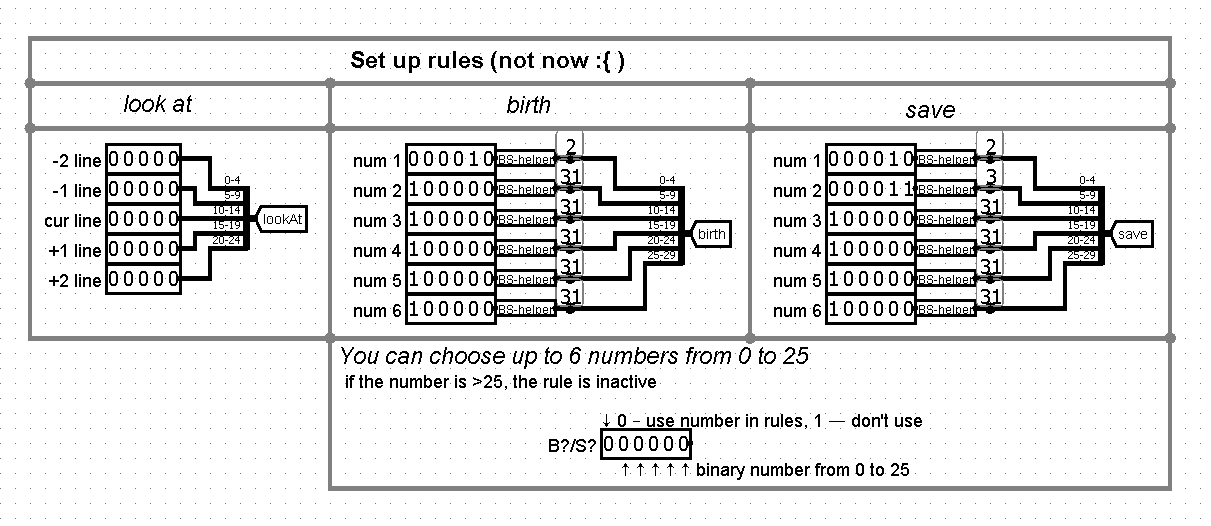
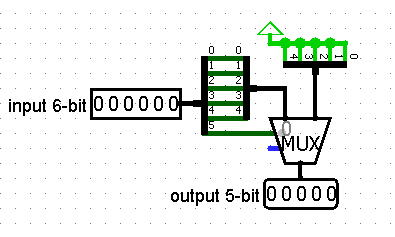


Let's pay a little attention to small schemes that perform the functions of assistants.

BS-Helper

It is used only in the main scheme

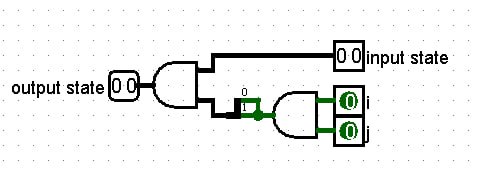
It is needed for convenient editing of rules

So that not all the numbers are shouted in units, but only the oldest

State-Helper

If the cursor (i, j) is on the selected cell, transfer the new state to the alive (or AliveSTD, without the rule editor) scheme.

Otherwise, pass the state (00) — old, that is, do not change the cell (since the cursor is not on it)



# Software

# User Guide

# Conclusion