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**

**Authors**

**Diza Michael - 22214**

**Yakhimovich Daniil - 22214**

Job topic:

**"The game of life “**

**Contents**

[**Introduction 2**](#_ht1eepapee3q)

[Rules 2](#_pkn81h45pc8b)

[Why 2](#_8h3j1afg62sz)

[**Main part 4**](#_3zswnz44stpu)

[User interface 4](#_aks47pgadnjs)

[User features 5](#_m631l6kt3mav)

[Software 7](#_o6b769wuwoz1)

[Main input 7](#_z9npnif3plhi)

[Load glider 9](#_i4t6vlht7tk0)

[Load pulsar 10](#_nlmak8dazu86)

[Connection 11](#_qro5d0pya0me)

[Hardware 13](#_qks8lxu9ps62)

[“Input State” 13](#_lhaxg5pn03zn)

[“Input” 14](#_vaj6wgbqst61)

[“Memory” 15](#_ou3q2svijxwy)

[“Rows producer” 17](#_a1gmpmg8jrku)

[“Pointer delay” 19](#_hggmdpddyh2c)

[“Converter 16 to 18” 20](#_xfcvzogkvsn9)

[“One row logic” 21](#_eeb5v9e8e25n)

[“One cell logic” 22](#_dor2ec14mpld)

[“Coordinates display” 23](#_tb34znpz1zx2)

[“Speed changer” 23](#_w28hrgysuczy)

[“Run button” 24](#_td6e9q9txz0c)

[**Conclusion 26**](#_gz1pmlku5e4o)

# 

# Introduction

The Game of Life, also known simply as “Life”, is a cellular automaton invented by British mathematician John Horton Conway in 1970. This is a zero-player game, which means that its evolution is determined by its initial state and does not require further input. User interacts with the Game of Life by setting up the first generation of cells and watching how it develops.

## Rules

The play area of the Game of Life consists of a grid of square cells, each in one of two states: alive or dead. Every cell interacts with its eight neighbors, which are horizontally, vertically, or diagonally bordering. At each tick of time, the next rules are being implemented:

1. If a live cell has two or three live neighbors, it survives.
2. If a dead cell has three live neighbors, it becomes a live cell.
3. Other live cells die in the next generation. All of the other dead cells stay dead.

Each new generation is entirely influenced by the previous one. The rules continue to be applied repeatedly to create new generations.

## Why

Our team has chosen to develop an implementation of the "Game of Life'' cellular automate due to its unpredictability and the complexity of the shapes that can be created from its simple rules. This decision was also motivated by the variety of programming languages in which this game can be implemented. In particular, we have selected Logisim and Cdm8 for our project. These tools offer a suitable combination of simulation capabilities and performance, allowing us to implement and test the "Game of Life'' algorithm efficiently. We expect that this project will provide valuable insights into the behavior of cellular automate and serve as a useful example for those interested in exploring this topic further.

# Main part

Our implementation of the game allows users to change cell states at any generation and observe the game process on a screen. The project consists of two interconnected and interacting parts:

* Hardware
* Software

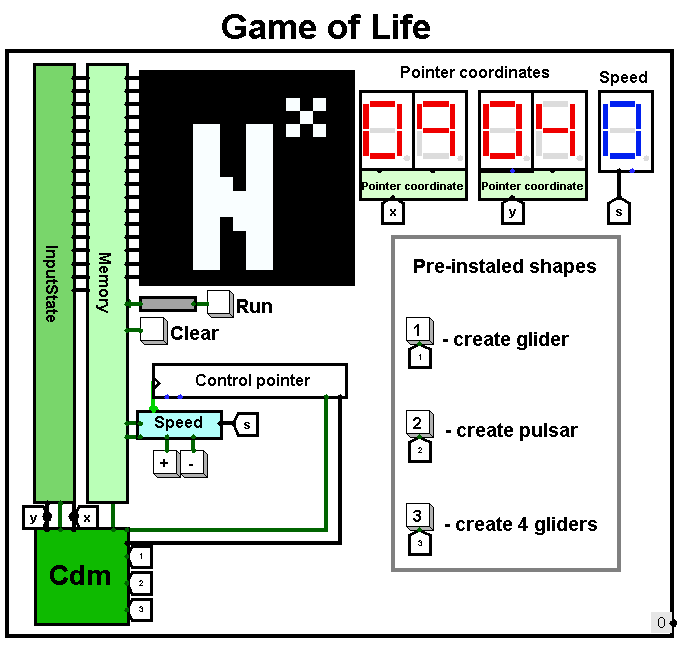
The hardware of our project consists of circuits developed at Logisim. The hardware is responsible for game logic processing and information displaying on the 16x16 LED screen.

The software part of our project is implemented on the "CdM-8-mark5" processor. We used modified Assembler code to implement the software, which is responsible for scanning user input. In particular, the software handles keyboard input, which allows users to change the state of the cells at any generation. And the software is also implemented to load pre-prepared patterns.

Our project uses the Harvard Architecture, but we don't need to store a lot of data while the program is running, so we have removed RAM. All the necessary data during code execution, we store in registers. These are sufficient for our project, since the software component should only handle keyboard input.

## User interface

The user interface (Fig. 1) contains a 16x16 LED screen, pins to change the frequency of the next colony state display, a pin to start and stop the game, a button to zero the field, a keyboard, buttons to enter pre-prepared patterns, 2 HEX Digit displays to show current pointer coordinates and 1 HEX Digit display to show speed of game.

(Fig. 1)

## User features

1) Users can change coordinates of the pointer, thereby moving it across the playing field by entering ‘w’, ‘a’, ‘s’, ‘d’ characters on the keyboard, and place or remove alive cells on the playing area, using Enter.

2) Start or stop the game, by pressing the “Run” button. Pressing this button switches game modes: In “output mode” user input is disabled, and game logic is processed. In “input mode”, a pointer appears on the screen, allowing the user to change the state of cells.

3) Create pre-installed generations in the game field, using “1”, “2”, “3” buttons. It executes instructions to the pointer: change cells' state in certain coordinates (it means that the remaining cells won't be changed).

4) Change game speed by using “+” and “-” buttons, connected to the “Speed” circuit. Our system offers four distinct speeds.

5) Clear screen by pressing the “Clear” button. Works at any moment, makes all cells dead.

## Software

Our project consists of two types of code: a parser for users commands and a simple input pattern.

### Main input

| asect 0xf3 Changes:   asect 0xf2 play:  asect 0xf1 setBit:  asect 0x00 start:  ldi r3, 0  ldi r0, Changes readkbd:  do  ld r0, r1  tst r1  until pl    ldi r2, 0x0a   if   cmp r1, r2  is eq   ldi r0, setBit  st r0, r0  ldi r0, Changes  br readkbd  fi    ldi r2, 0x64  if   cmp r1, r2  is eq  ldi r2, 16  add r2, r3  st r0, r3  br readkbd  fi    ldi r2, 0x61  if   cmp r1, r2  is eq  ldi r2, -16  add r2, r3  st r0, r3  br readkbd  fi | ldi r2, 0x77  if   cmp r1, r2  is eq  move r3, r2  shla r2  shla r2  shla r2  shla r2  if  tst r2  is eq  ldi r2, 16  add r2, r3  fi  dec r3  st r0, r3  br readkbd  fi    ldi r2, 0x73  if   cmp r1, r2  is eq  move r3, r2  shla r2  shla r2  shla r2  shla r2  ldi r1, 0xf0  if  cmp r2, r1   is eq  ldi r2, -16  add r2, r3  fi  inc r3  st r0, r3  br readkbd  fi    br readkbd    end |
| --- | --- |

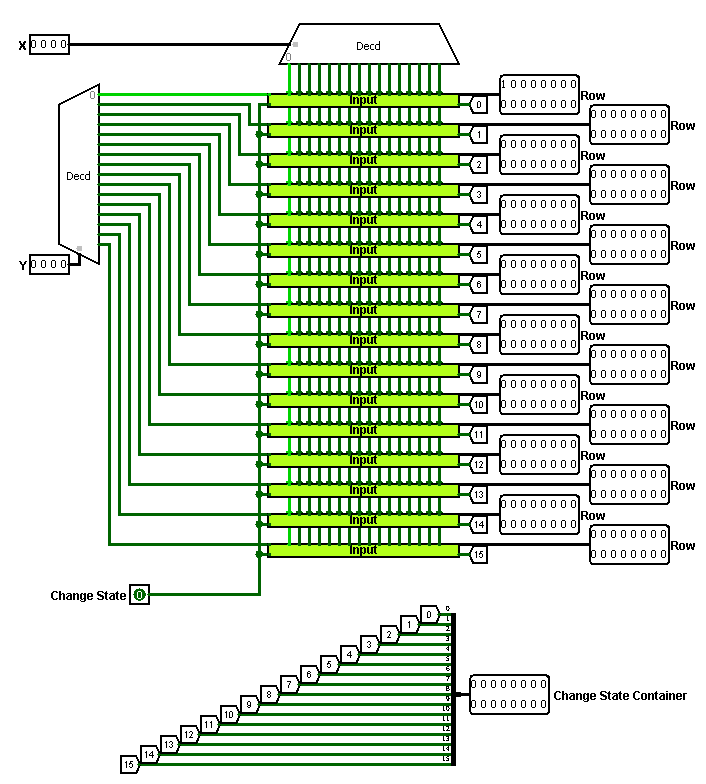
At the very beginning the addresses of the external devices and the beginning of our program are declared. Then the data is written to the registers. The main block of code begins. The program waits for data entering, the input indicator is the high bit of the number, if it is zero then the user has entered some key. After this value is compared to the codes of valid commands. If the entered command is wrong then the program does nothing. If the code is valid, the program will either move the pointer to another position or set a flag indicating that the state of the current cell has changed. The program saves only the current pointer coordinates, which are originally set to (0, 0), representing the upper left corner of the matrix.

Extra inputs

| | asect 0xf7 end:  asect 0x00 restart:  ldi r1, end  st r1, r3  br start  asect 0xf3 Changes:   asect 0xf1 setBit:  asect 0x20 start:  ldi r0, Changes  ldi r2, 16  ldi r3, 0 | | --- |   The general part for each additional input contains the declarations of the necessary external devices, writing constants and addresses to registers, and the "restart" code. The “restart” starts working when the loading of the pattern is complete. The “end” flag is raised here. From this address the execution of the main code continues.  Loading patterns has a naive implementation. Consistent change of pointer coordinates and cell states. Load glider  | inc r3  inc r3   inc r3  add r2, r3  add r2, r3  st r0, r3  ldi r0, setBit  st r0, r0  ldi r0, Changes    inc r3  st r0, r3  ldi r0, setBit  st r0, r0  ldi r0, Changes    inc r3  st r0, r3  ldi r0, setBit  st r0, r0  ldi r0, Changes    add r2, r3  st r0, r3  ldi r0, setBit  st r0, r0  ldi r0, Changes    dec r3  add r2, r3  st r0, r3  ldi r0, setBit  st r0, r0  ldi r0, Changes | | --- |  Load pulsar  | add r2, r3  add r2, r3  add r2, r3  add r2, r3  add r2, r3  add r2, r3  inc r3  inc r3  inc r3  ldi r1, 4  while   tst r1  stays pl  st r0, r3  ldi r0, setBit  st r0, r0  ldi r0, Changes  add r2, r3  dec r1  wend  add r2, r3  inc r3  inc r3  neg r2  ldi r1, 8  while   tst r1  stays pl  st r0, r3  ldi r0, setBit  st r0, r0  ldi r0, Changes  add r2, r3  dec r1  wend  add r2, r3  inc r3  inc r3  neg r2  ldi r1, 12  while   tst r1  stays pl  st r0, r3  ldi r0, setBit  st r0, r0  ldi r0, Changes  add r2, r3  dec r1  wend | inc r3  inc r3  neg r2  add r2, r3  add r2, r3  add r2, r3  ldi r1, 8  while   tst r1  stays pl  st r0, r3  ldi r0, setBit  st r0, r0  ldi r0, Changes  add r2, r3  dec r1  wend  inc r3  inc r3  neg r2  add r2, r3  add r2, r3  add r2, r3  ldi r1, 4   while   tst r1  stays pl  st r0, r3  ldi r0, setBit  st r0, r0  ldi r0, Changes  add r2, r3  dec r1  wend | | --- | --- |  Connection (Fig 2)  CPU connection.  The CPU is connected to 4 ROM blocks. In each of them is written some code. The upper three blocks are for automatic loading of the patterns. The lowest one is the main memory block, it contains code for analyzing user's commands entered from the keyboard. This is the main memory block, it works by default. The other memory blocks turn on, interrupt the main code, execute and return control to the main one.  The logic of the memory block selection (see fig. 3) is implemented with a demultiplexer. A number is written to the register - the number of the memory block. The demultiplexer selects one of the 4 blocks of code, which is executed. So after one of the patterns is loaded the 'end' flag is raised which clears the register. Its normal state is 0 when the main code is run. We also get “Run” to prevent input during the “output mod”.  (Fig. 3)  Raising the flag is implemented with the bit selector, then the output has only two values: '0' and '1'. To raise the flag you need to write something to a specific address, in the example (Fig. 4) the address: 0xf7.  (Fig. 4)  External I/O device (fig 5). When data is input, the values are written to the register and the high bit of the number is set to 0. This number is sent to the processor. “IOdat” - data bus, “IOaddr” - address bus, “IOsel” - selection of external devices.  (Fig. 5) |
| --- | --- | --- | --- | --- |

## Hardware

### “Input State”

(Fig. 6)

Circuit input:

* “X” “Y” – 4 bit coordinates (from “Cdm”)
* “Change state” – 1 bit instruction from “Cdm”

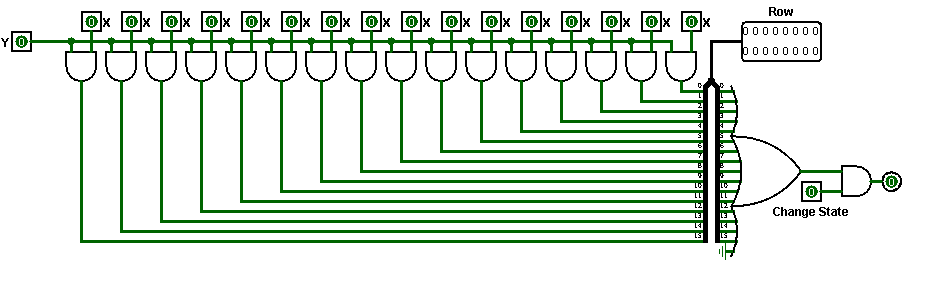
Circuit output:

* “Row” x16 – 16 bit rows (“Input” result)
* “Change State Container” – 16 bit number - contain change instruction for all rows (“Input” result)

Behavior:

The pointer coordinates are sent to the "Input State" circuit from the "Cdm". This circuit then converts the 4-bit coordinates into 16 rows of 16 bits each, representing the game field with only a pointer. Initially, every bit of these rows is set to 0 except for the coordinates of the pointer. A decoder is used to convert all possible positions of the 4-bit input into their corresponding 16-bit positions in the game field. Each of these positions has all bits set to 0 except for one, which corresponds to the location of the pointer on the field. In essence, the decoder creates one row and column of the game field with only one bit set to 1, representing the position of the pointer. At the bottom, the change instructions are concatenated into a single 16-bit number.

### “Input”

(Fig. 7)

Circuit input:

* “X” “Y” – 1 bit parsed coordinates (from “Input State”)
* “Change state” – 1 bit instruction (from “Input State”)

Circuit output:

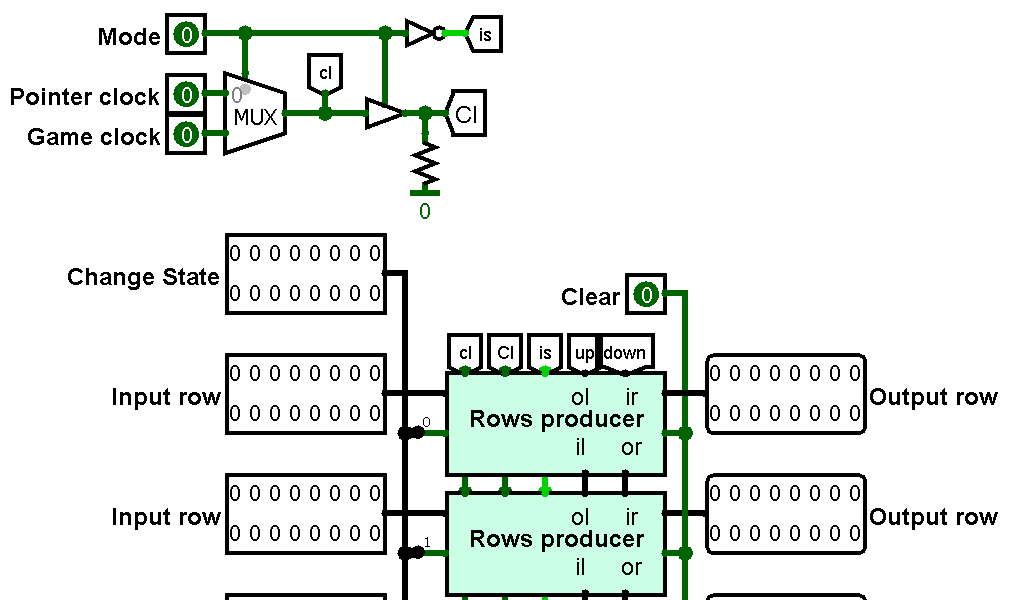
* “Row” – 16 bit row with/without pointer

Behavior:

Next we have 2 coordinates, so we use an “Input” circuit for every bit from Y to concatenate with “X“, if “Y“ bit toggled there will be a toggle bit in the field. Also we need to track is there cell that was changed by user (Enter was on keyboard), so if in this circuit was received flag and pointer in this row (we use “OR“ for all input “X“ coordinates to find toggled).

We have two sets of coordinates, and we use an "Input" circuit for each bit in “Y“ to concatenate it with “X“. If a bit in “Y“ is toggled, the corresponding bit in the field will also be toggled. Additionally, we need to keep track of whether a user has changed a cell (i.e., pressed the "Enter" key on their keyboard). To accomplish this, we check for a received flag and pointer in the row, using an "OR" operation across all input “X“ coordinates to find any toggled cells (Feature : we used 17-input “OR” and “Ground” just to place “OR” closer to the “Splitter” and avoid piling up wires).

### “Memory”

(Fig. 8)

Circuit input:

* “Mode” – 1, when game mode is “output”, 0, when game mode is “input”(from “Main”)
* “Pointer clock” – 1 bit (from “Main”)
* “Game clock” – 1 bit (from “Main”)
* “Input row” x16 – 16 bit row (from “Memory”)
* “Change state” – 16 bit container with instructions for each row (from “Memory”)
* “Clear” – 1 bit instruction(from “Main”)

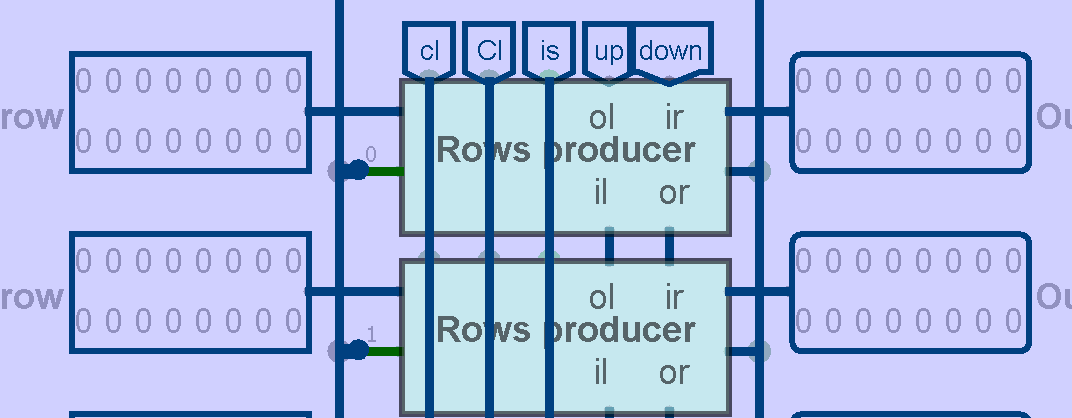
Circuit output:

* “Row” – 16 bit row with/without pointer

Behavior:

For the input, we have the entire game field divided into 16 rows, along with a container of instructions, various clocks, and the current game mode. Depending on the “Mode”, we select the appropriate frequency by using a “Multiplexer” to send different clocks to the “Cl“ and “cl” gates. When the mode is set to 'input' (state = 0), we use the “Pointer clock“, and when the mode is “output“ (state = 1), we switch to the “Game clock“. To the “is“ gate, we send the inverted “Mode“ signal (a feature made during development).

Next, below that, we analyze the rows of the field using the “Rows producer“ circuit. It receives the “Input row“ (which will be processed), “is“, “Cl“ and “cl“ gates, a 1-bit “Change state“, and the bordering rows.

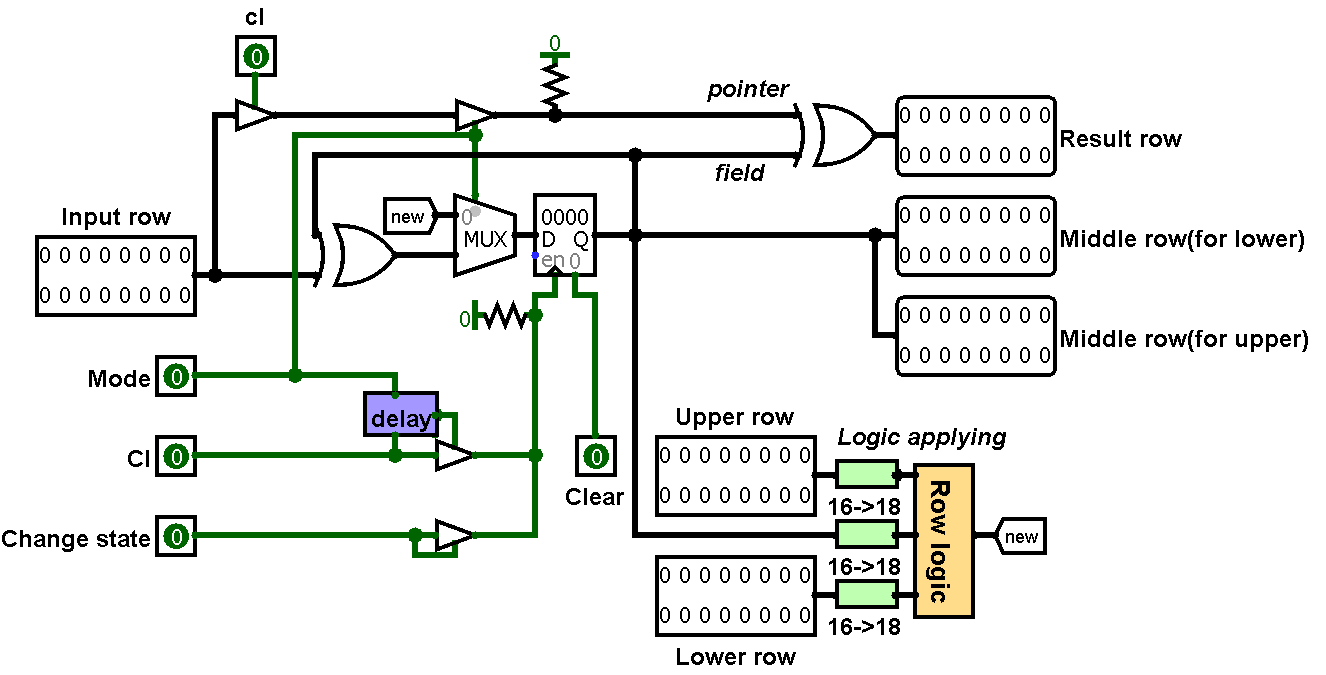
(Fig. 9)

On Figure 9, you can observe that gates “is”, “Cl”, and “cl” for each row. These wires are located under the “Rows producer” circuit to prevent gate reuse and ensure a “Memory” design.

The “Rows producer” circuit labels such as “ol” (output left), “or” (output right), “il” (input left), and “ir” (input right) are made to indicate the connection of these circuits. Near to “ol“ and “or“ are outputs that contain processed rows for the upper and lower 'Rows producer' circuits. Similarly, “il” and “ir” inputs serve to receive results from other “Rows producer” circuits, specifically from “or” and “ol” outputs.

The situation for the first and last rows is slightly more complex. We transmit output and input to the “down” and “up” gates to make the field looped from the top and bottom corners, with the upper being connected to the last row and the lower to the first row.

### “Rows producer”

(Fig. 10)

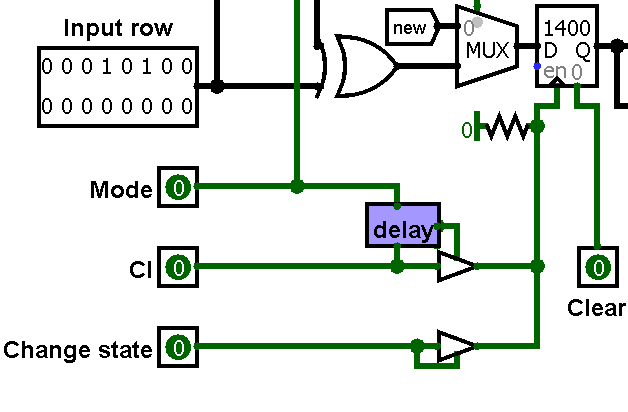
Circuit input (all from “Memory”):

* “Input row”– row with pointer
* “Change state” – instruction for pointer
* “cl” – pointer clock
* “Cl” – game clock
* “Mode”– not “Run”
* “Upper row” – row along “Input row”, in “Memory” appearance “ir”
* “Lower row” – row below “Input row”, in “Memory” appearance “il”
* “Clear” – instruction of field cleaning

Circuit output:

* “Output row” – result
* “Middle row(for upper)” “Middle row(for lower)” – results which essential for another rows calculations

Behavior:

(Fig. 11)

Firstly you can see 3 pins at Figure 11 (“Mode”, “Cl”, “Change state”) which are responsible for work with the “Register”. “Pointer delay” circuit lets the circuit ignore several ticks when the game just started (it’s essential to avoid pointer influence at game logic), so after 4 ticks, it transmits 1 to the “Controlled buffer” and sends “Cl” ticks further. So if “Mode” is toggled (“input” mode), data in “Register” will be changed only if “Change state” instruction is switched. Otherwise, data will be constantly updated after delay (with “Cl”). “Clear” reset data when toggled.

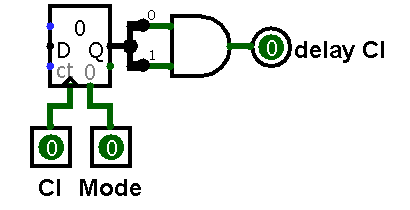
So we have saved rows in “Register”. From here data is transmitted in 3 directions:

* output (left wire near “Register” in Figure 10). Divided into two outputs from upper and lower row (design feature)
* to logic circuits (bottom wire). Then the current row and bordering input rows transform into 18 bit rows (It’s essential to loop the left and right corners of the field). And transmit to the “One row logic” circuit, which applies a game rules to all cells from the row. From here it is sent to the “new” gate.
* to “XOR” gates (upper wire). “XOR” with “Input row” required to change the state of the cell in coordinates of the pointer. Then “Multiplexer” is used to choose rows in “output mode”(“new” gate) and “input mode”(with pointer changes). Finally it sends back to memory (“Register”)

At the end we use “XOR” on the result and “Input row” to display the pointer.

There are also 2 “Controlled buffer”. Left to make pointer blinking (with “cl” pin frequency) and right to hide pointer when “Mode” toggled (game started).

### “Pointer delay”

(Fig. 12)

Circuit input:

* “Cl” – 1 bit (from “Rows producer”)
* “Mode” – 1 bit (from “Rows producer”)

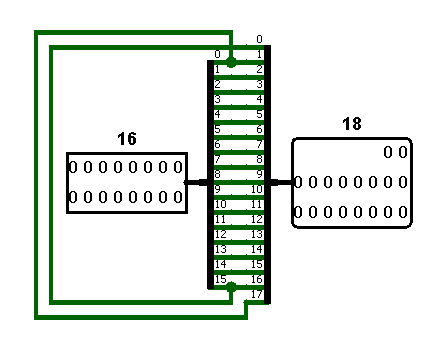
Circuit output:

* “delay Cl” – 1 bit

Behavior:

If “Mode” has state 0 (“input mode”), “Counter” increases value, when it becomes equal to 3(11 in binary, so “AND” used), it has 1 in output. If “Mode” toggled, counters have 0 in any way.

### “Converter 16 to 18”

(Fig. 13)

Circuit input:

* “16” - 16 bit row (from “Rows producer”)

Circuit output:

* “18” - 18 but row

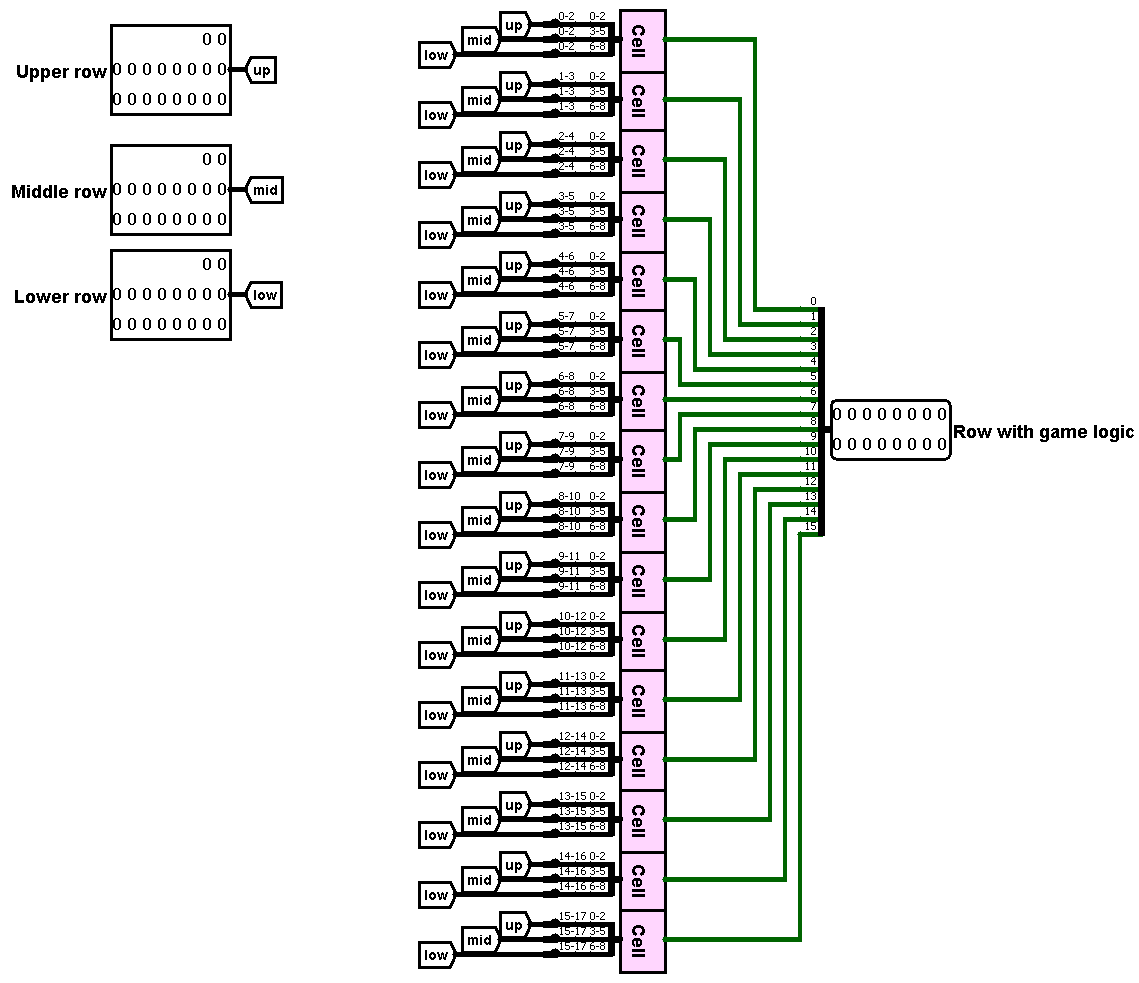
Behavior:

This circuit create 18-bit row and set :

* 0 bit state to 17th bit
* 15th bit state to 0 bit

Thus 100001010101000 turns into 01000010101010001.

### “One row logic”

(Fig. 14)

Circuit input:

* “Lower row” – 18 bit row (from “Rows producer”)
* “Middle row” - 18 bit row (from “Rows producer”)
* “Upper row” - 18 bit row (from “Rows producer”)

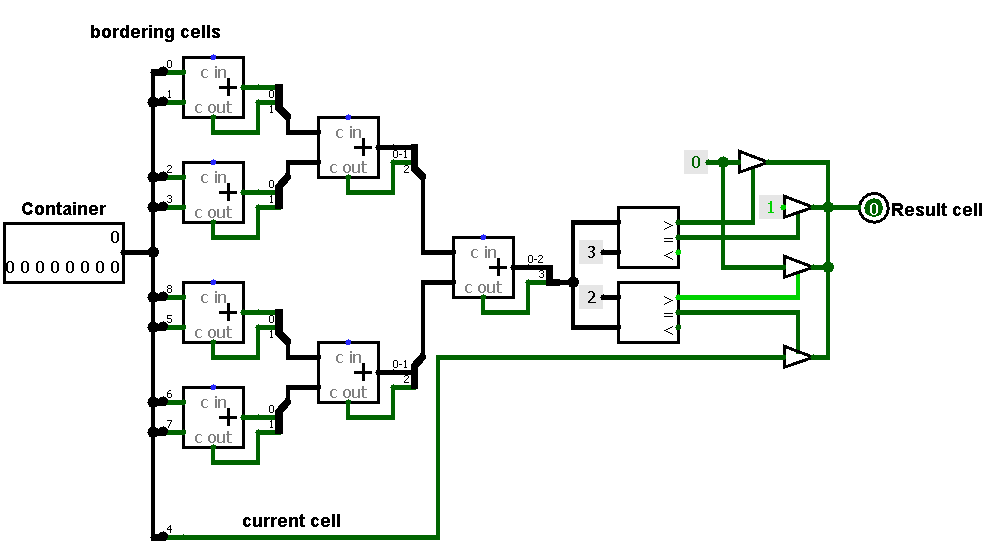
Circuit output:

* “Row with game logic” – result (shows in display)

Behavior:

First of all we send bordering rows in gates “up”, “low”, “mid”. Then we select groups of 3 cells(1 main cell and 2 horizontal neighbors) for each cell, and unite each group from the “mid” gate with bordering groups from “up” and “low” gates(vertical neighbors) to 9-bit containers. These containers transmit to “One cell logic”. At the output of this we have a state of 1 cell (in the middle).

### “One cell logic”

(Fig. 15)

Circuit input:

* Container – 9 bit, represents “square” of bordering cells (from “One row logic”)

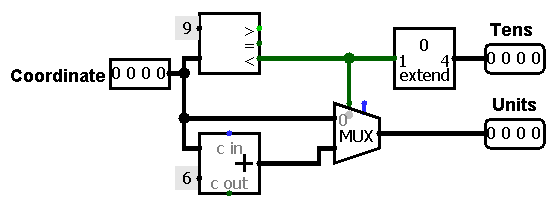
Circuit output:

* Result state (state of cell in the middle)

Behavior:

Using 7 “Adder” circuits we count the number of bits in the container with state 1 (except 4th bit, it is the central cell). According to the number, we change the cell state.

### “Coordinates display”

(Fig. 16)

Circuit input:

* “Coordinate” - 4-bit number (from “Main”)

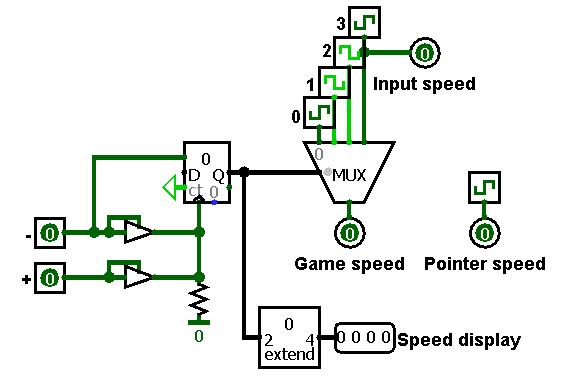
Circuit output:

* “Tens” - 4-bit number
* “Units” - 4-bit number

Behavior:

This circuit converts 4-bit numbers (HEX) to 2 4-bit numbers (Decimal). We converse it by adding 6 (“Adder” circuit). If “Coordinate” is greater than 9, “Tens” will be 1(0001 in binary, so “Bit Extender” was used).

### “Speed changer”

(Fig.17)

Circuit input:

* “+” - 1-bit pin (from “Main”)
* “-” - 1-bit pin (from “Main”)

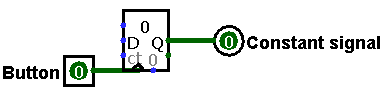
Circuit output:

* “Input speed” - keyboard input frequency
* “Game speed” - processing of game logic frequency
* “Pointer speed” - pointer blinking frequency
* “Speed display” - 4-bit number for HEX display

Behavior:

If “+” toggled, counter increases speed number, if “-” - decreases. Then the speed number is transmitted to “Multiplexer”, then to “Game speed”. “Pointer speed” and “Input speed” are constant. Also speed number sent to “Bit extender” (essential for display),then to “Speed display”.

### “Run button”

(Fig. 18)

Circuit input:

* Button - 1-bit signal(from “Main’)

Circuit output:

* Constant signal - 1-bit value

Behavior:

Transform short-time signal from button to constant value, using “Counter” (1 bit and Wrap around value, thus every time, button pressed, value changed)

# Conclusion

In conclusion, we have successfully implemented a version of the Game of Life cellular automate using Logisim and Cdm8. This implementation allows users to interact with the game by changing cell states and observing the game process on screen. Overall, our implementation of the Game of Life has been a success and has allowed us to deepen our understanding of cellular automate and its behavior.