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

Demidov Dmitry

Letov Alexey

Dubrovin Artem

Job topic: "CONWAY GAME OF LIFE"

Novosibirsk, 2024

Table of contents

Introduction

The Conway’s Game of Life, often referred to simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970. The game is a zero-player game, meaning that its evolution is determined by its initial state, requiring no further input from human players. Conway's Game of Life is simulated on a two-dimensional square grid. Each cell in this grid can be in one of two states: alive or dead (or "populated" or "unpopulated"). The state of each cell changes from generation to generation based on the states of its eight neighbors according to a set of simple rules.

Basic Rules:

1. Birth: An empty or dead cell becomes alive (populated) if it has exactly three alive neighbors.
2. Survival: An alive cell remains alive if it has two or three alive neighbors.

3. Overpopulation: An alive cell dies (becomes empty or dead) if it has more than three alive neighbors.
4. Underpopulation: Conversely, an alive cell dies if it has fewer than two live neighbors.

But we also implemented three more cellular automata (the rules by which cells are born and die).

Maze rules:

1. Birth: A cell becomes alive (a wall in the maze) if it has 3 neighbors.
2. Survival: A cell remains alive if it has 1, 2, 3, 4, or 5 neighbors.

H-trees rules:

1. Birth: A cell becomes alive if it has 1 neighbor.
2. Survival: A cell remains alive if it has from zero to eight neighbors.

Persian Carpet rules:

1. Birth: A cell becomes alive if it has from two to four neighbors.
2. Death: Cells don't survive the next generation.

1. Problem statement

We were very interested in this project theme among all other ideas, so we wanted to take it. But we were quite disappointed with the standard implementation of the game in life, so we wanted to improve it, for example, to add several kinds of cellular automata. Besides, one of the reasons for improving our project was its popularity, so we wanted to add some interesting features to our project

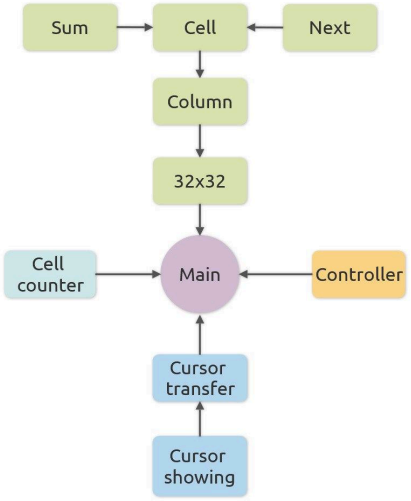
2. Analogues

These are features that make our project different from the standard game-to-life implementation:

1. 32x32 display, instead of the standard 16x16 display.
2. Cursor that is shown on the display and with which you can change the state for the cells of the initial pattern
3. 4 different cellular automata (certain rules by which cells are born and die)
4. Live and dead cell counter implemented through hex digit display

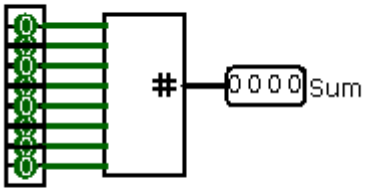
3. Hardware

Overall we have 10 schemes in our project, you can see their relationship in this scheme:



Let's start with the basic, how our cell is organized in general and how it works

Sum of neighbors



Inputs:

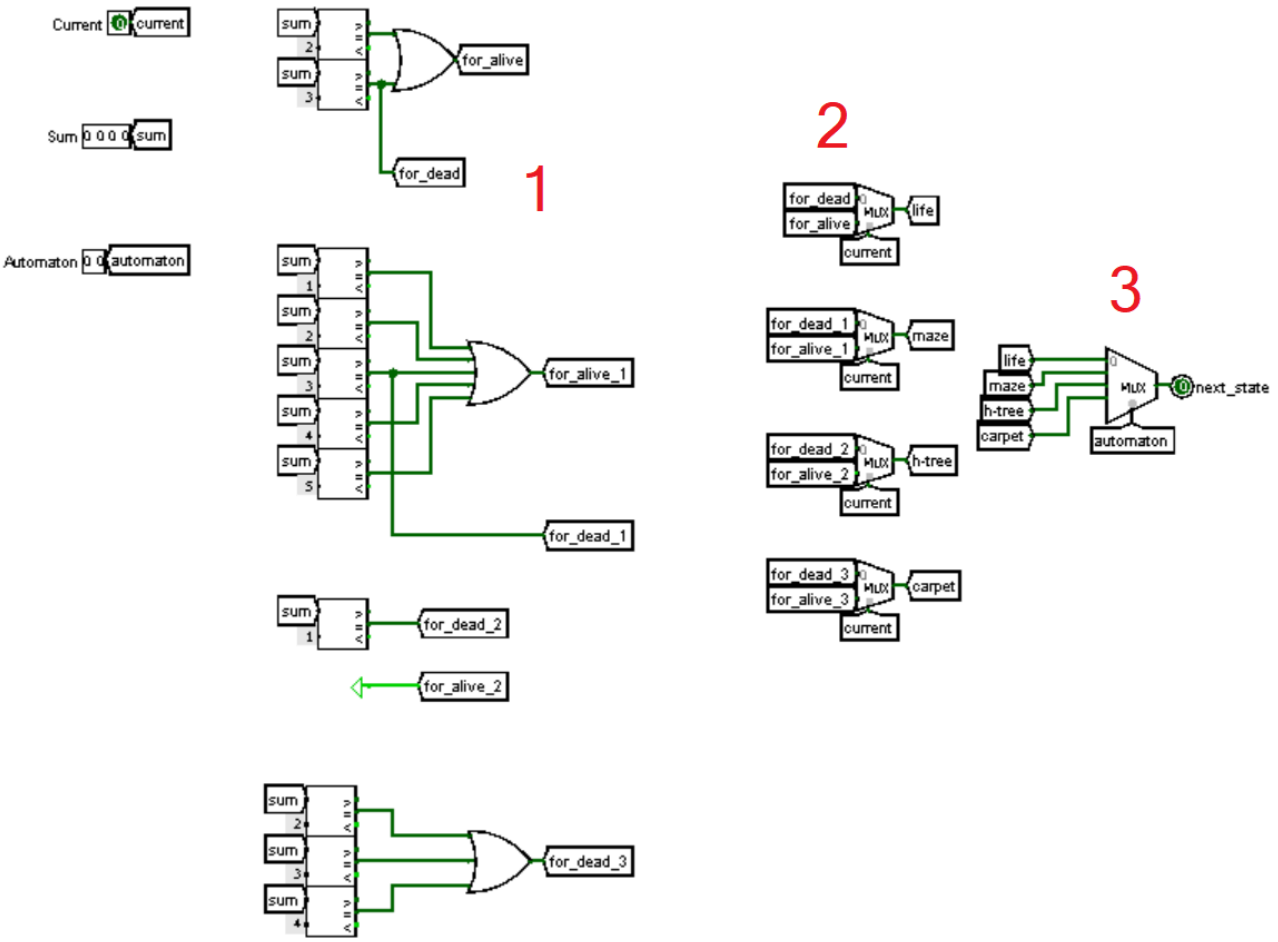
- 8 1-bit inputs

Outputs:

- 1 4-bit output

Here we have 8 pins that are being summarized by bit adder and the result is input for the next phase selector.

Next phase selector



Inputs:

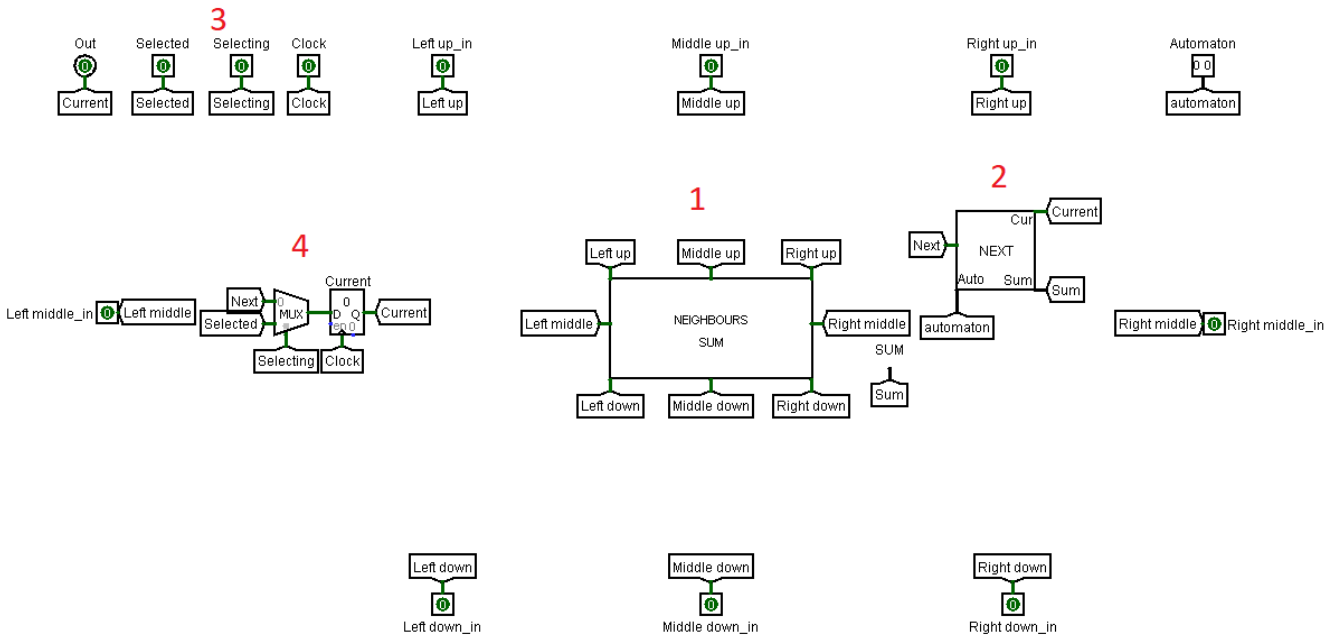
- 1-bit flag for current cell's state
- 4-bit input for number of neighbors
- 2-bit flag for automaton

Outputs:

- 1-bit output for next cell's state

1. For each possible automaton this scheme is counting next state for cell based on number of neighbors (both for dead and alive cases)
2. This scheme is choosing next state based on which state the cell has
3. This scheme is choosing next state based on automaton number

Cell optimized



Inputs:

- 8 1-bit inputs(neighbors state)
- 3 1-bit inputs(flags)

Outputs:

- 1 1-bit output

1. In general, we have 8 pins that are responsible for displaying the state of neighboring cells (whether they are alive or not). This part of the circuit counts their bit sum and it is used by the part of the circuit number 2.

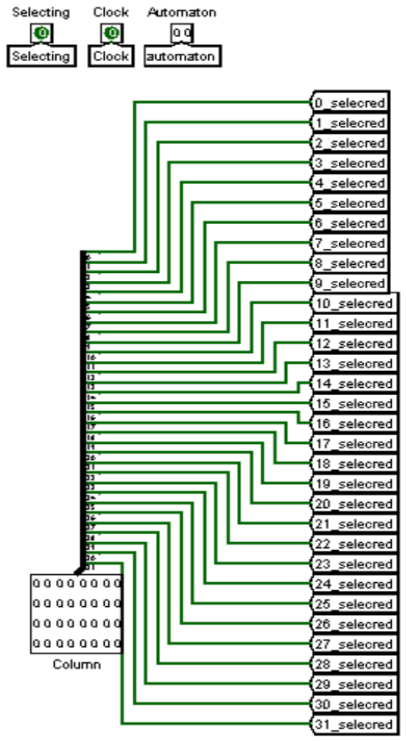
2. This part of the circuit responsible for calculating the next state of the cell depending on the sum and current state of the cell, the result is used in the 4’th part of the circuit.

3. Here we have 3 input flags and 1 output. Selected is a value which the user had chosen for the initial cell state before the game simulation. Selecting flag marks when the user is in the

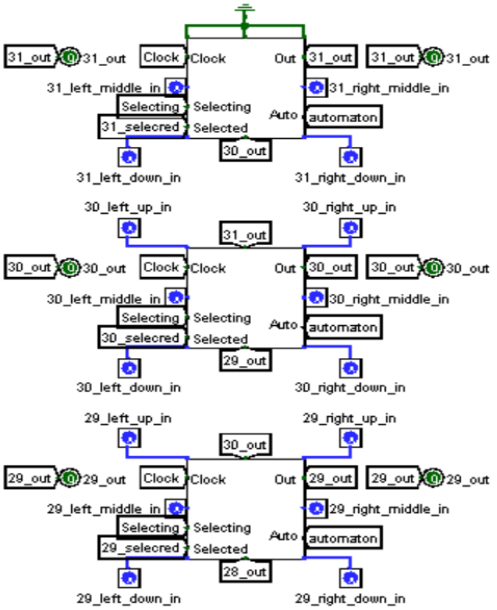
process of selecting cells for the initial pattern. Clock is always working, as it is responsible for updating the cell state. Current shows the current state of the cell.

4. This mechanism determines the current state of the cell. If the Selecting flag is set, current showing the user's choice whether to include this cell in the initial pattern or not. If the Selecting flag is not set, the current state of the cell will be stored in the register using next as a value. Eventually current(the current state of the cell) will be the output of this scheme.

Column of 32 cells



1



2

Inputs:

- Initial flags to provide them into cells
- 32-bit input
- 6 * 32 (6 for each cell) inputs to connect cells with other rows

Outputs:

- 32 * 2 (2 for each cell) outputs to provide cells' states to other rows

1. This contact is needed for changing the cell's state, it provides cell information about is that cell selected or not.

2. There we can see 32 cells connected to each other, and having 6 inputs each to receive information about neighbors and calculate next cell's state, and also 2 outputs for each to provide information about cell's state to other rows.

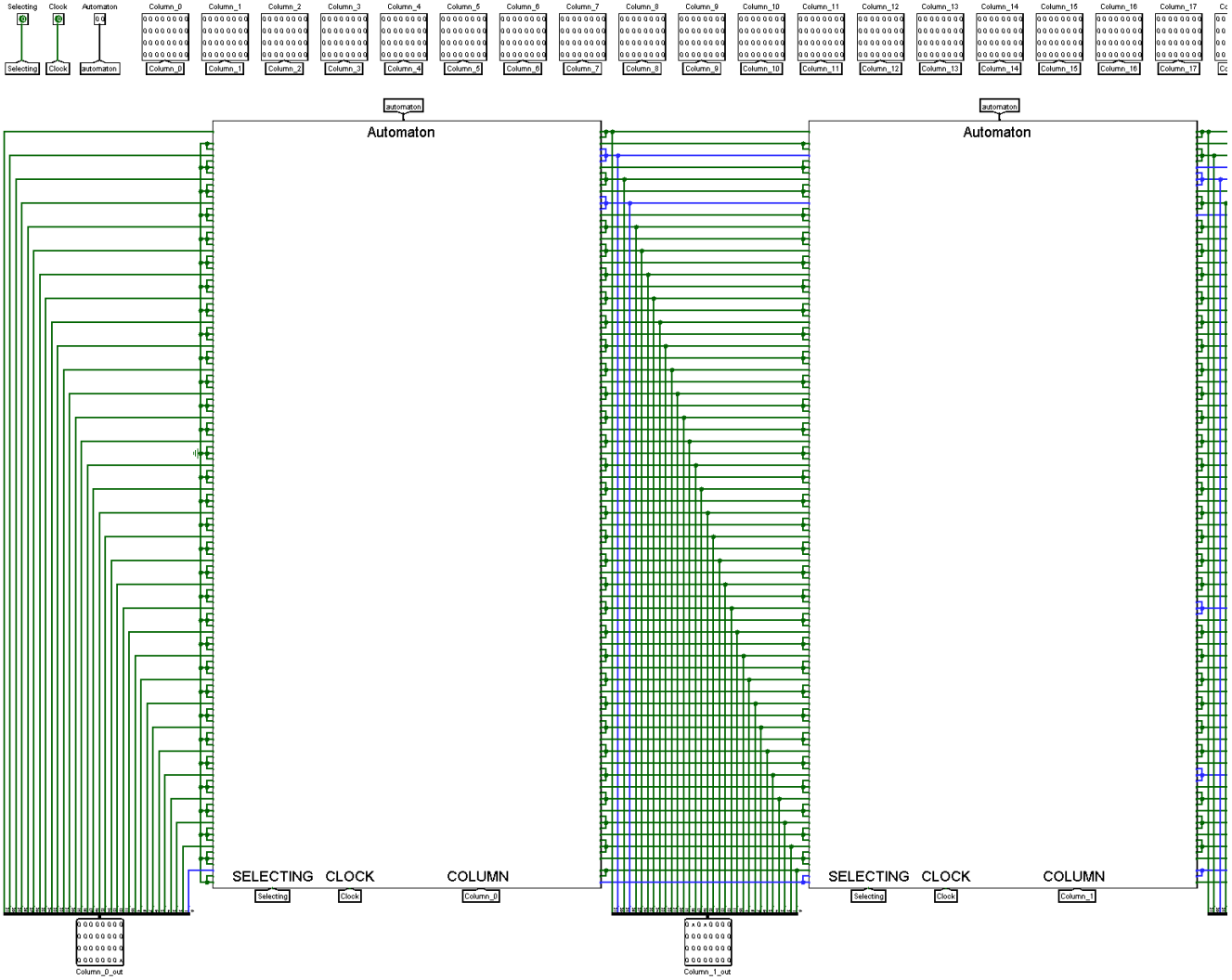
32x32 cells

Inputs:

- 32 32-bit inputs(for initial pattern of all cells that are in the game)

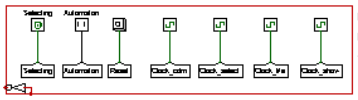
Outputs:

- 32 32-bit outputs(for displaying current state of all cells that are in the game)



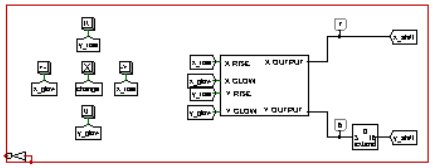
1. We have 32 32-bit inputs that are initial pattern of all cells and 2 flags, Selecting and Clock.
2. There are 32 columns in this part of the circuit. They connect neighboring cells to each other, for a total of 6 neighbors, as two more neighbors are already connected in the circuit of the column itself(top and bottom). Also, the first column has the left side grounded and the 32’nd column has the right side grounded, because in our implementation we consider the cells outside the field to be dead. And there are also 32 32-bit outputs of cell state in the columns.

main



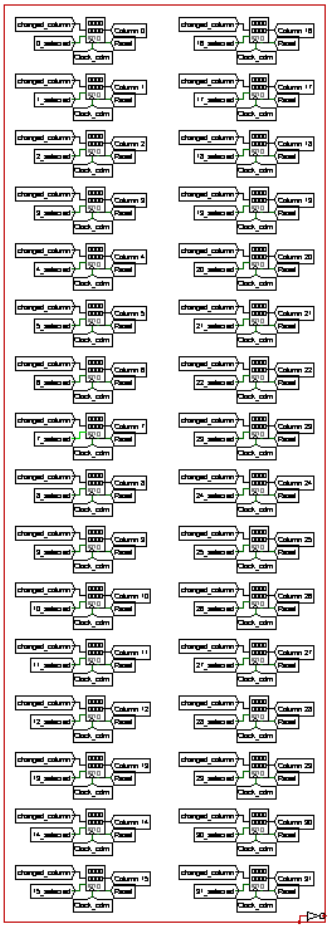
1

INITIAL FLAGS



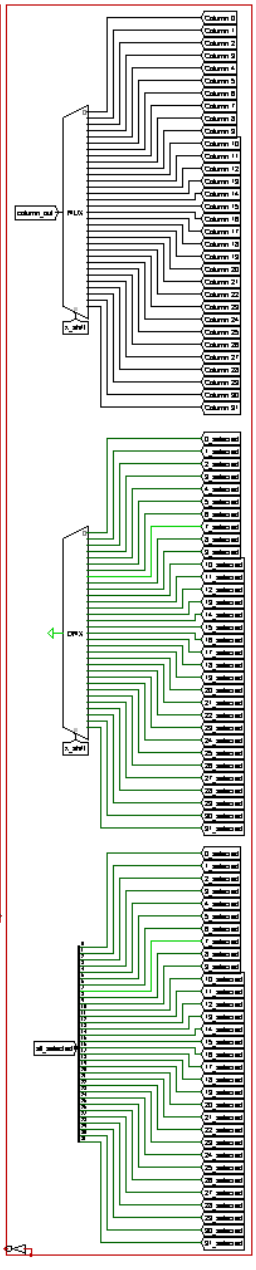
2

JOYSTICK INPUT

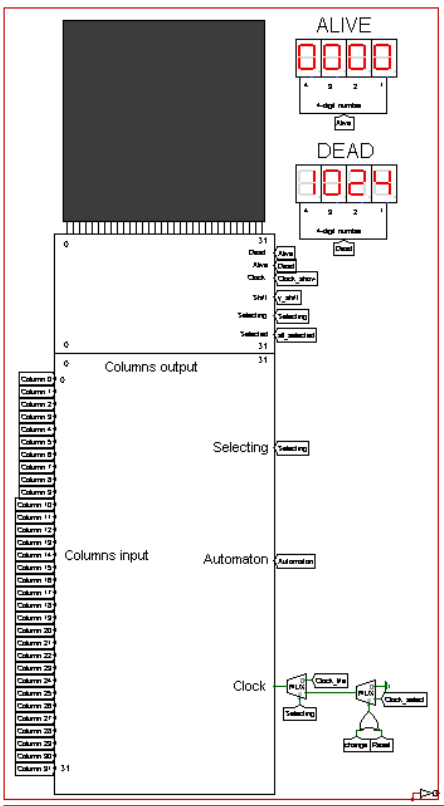


INITIAL PATTERN

5



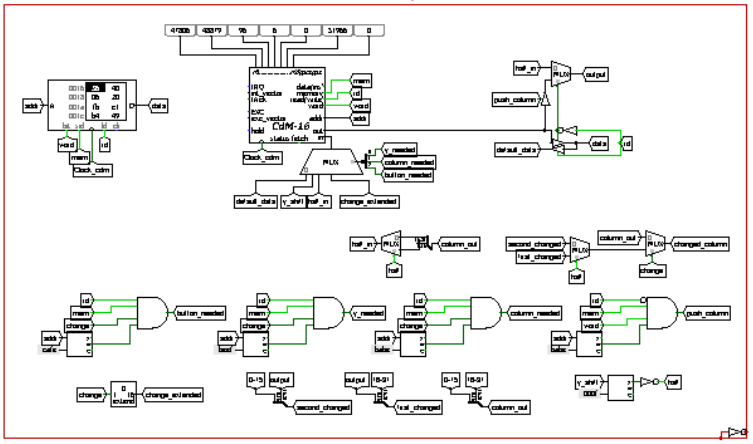
3



7

4

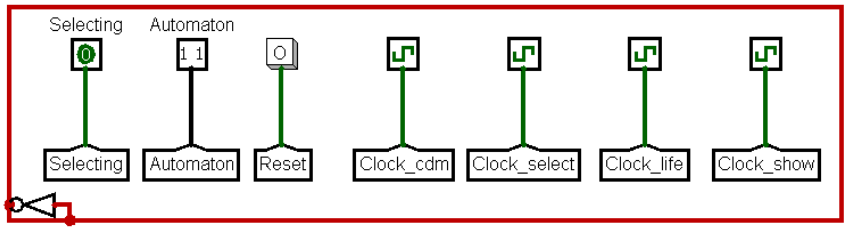
FIELD



CDM-16

6

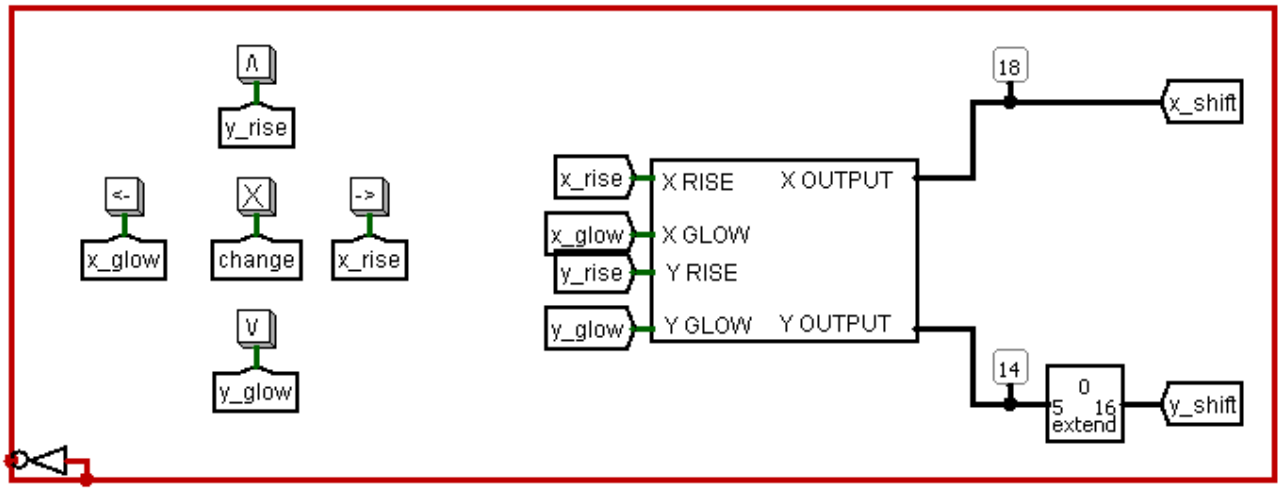
1.



00 - Conway's game of life
01 - Maze
10 - H-trees
11 - Persian carpet

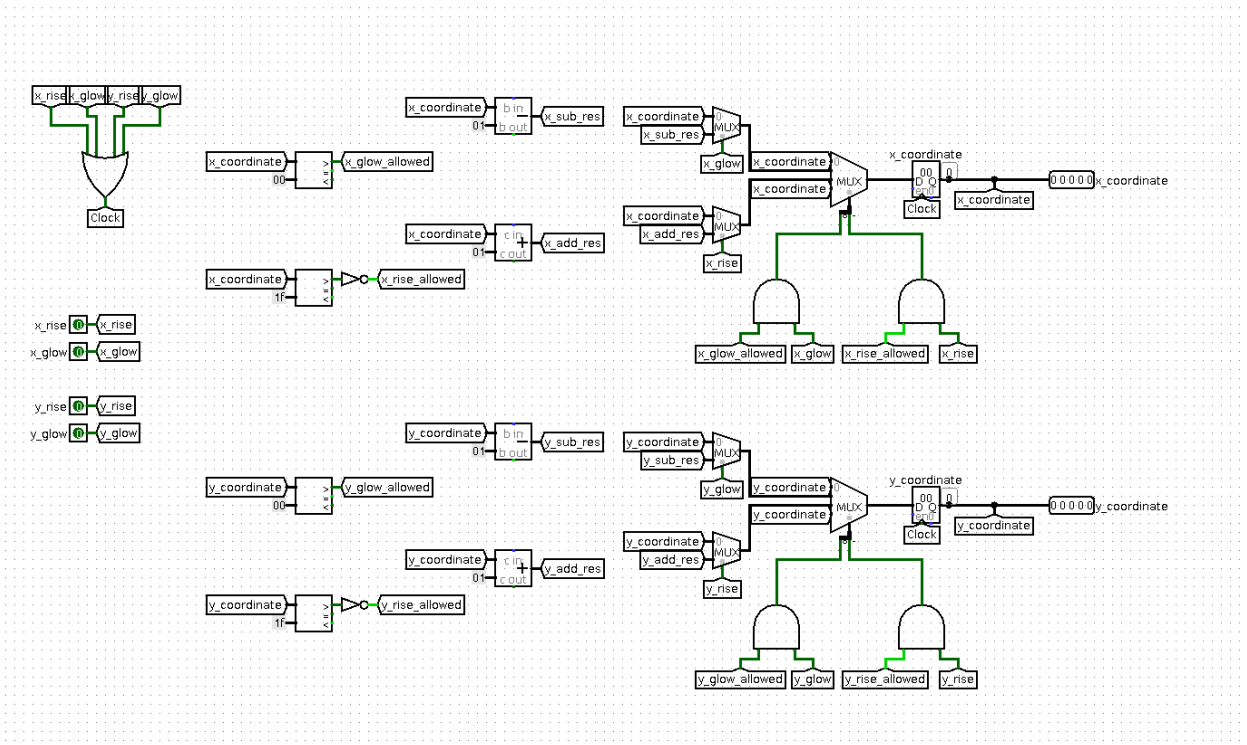
Overall we have seven flags. Selecting flag is used for marking if user during initial pattern placement phase. Automaton pin defines the cellular automaton to be used for generation. Reset button clears initial pattern data. Also we have four clocks with different duration and because of that, our program is more optimized. Clock_cdm is used by cdm-16 and to update initial pattern registers. Clock_select is used by 32x32 columns while choosing the initial pattern. Clock_life is used while there is an evolution process going . Clock_show is used to update the cursor position.

2.



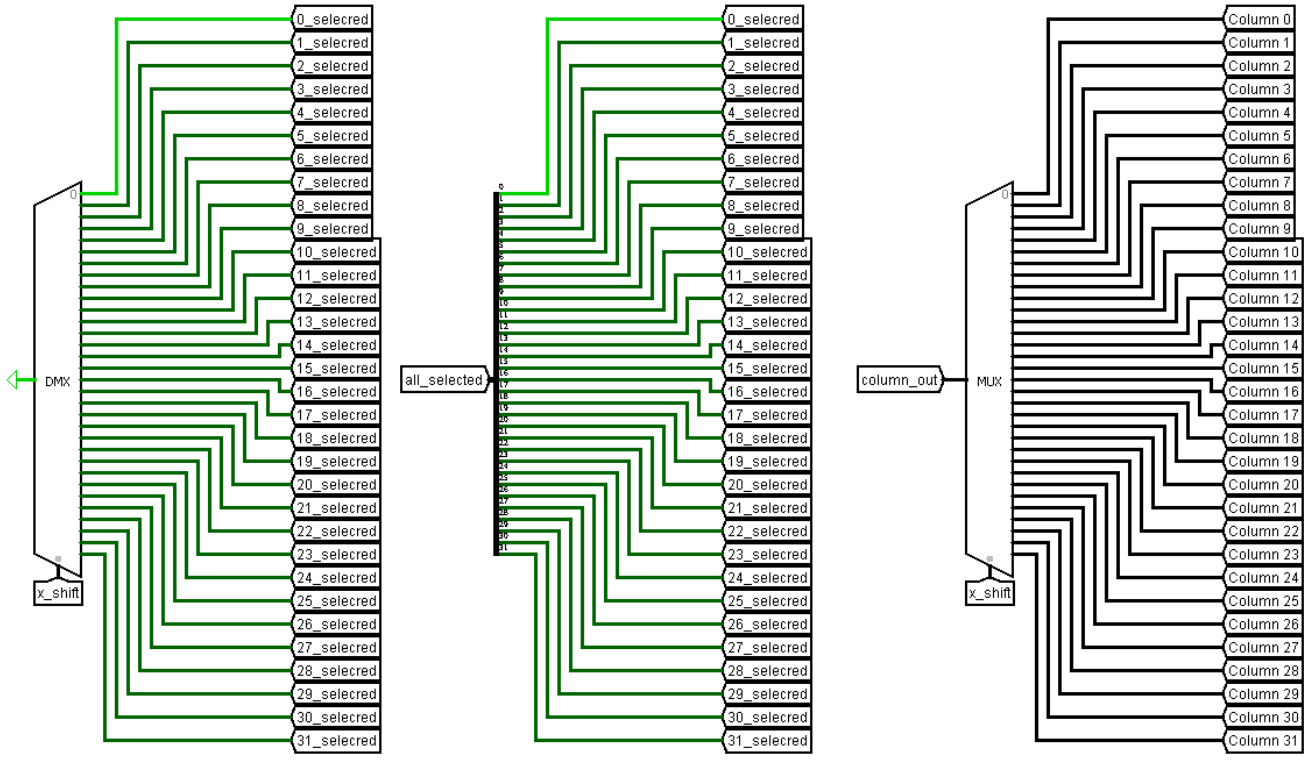
JOYSTICK INPUT

Here we have 5 buttons, four of which change the cursor position and one is responsible for changing selected cell value. There is X and Y coordinates that are being changed through buttons. The X coordinate is used by the logisim environment. While Y is used by the CDM-16 processor, so we use a bit extender to increase the bit width to 16 to work correctly with the processor.



This is what our mechanism for storing and changing coordinates looks like. At the beginning for each coordinate there are two comparators, which determine whether they are within the allowed limits. After that, depending on the signal of the buttons to the coordinates, the values in the registers are replaced by the changed coordinates.

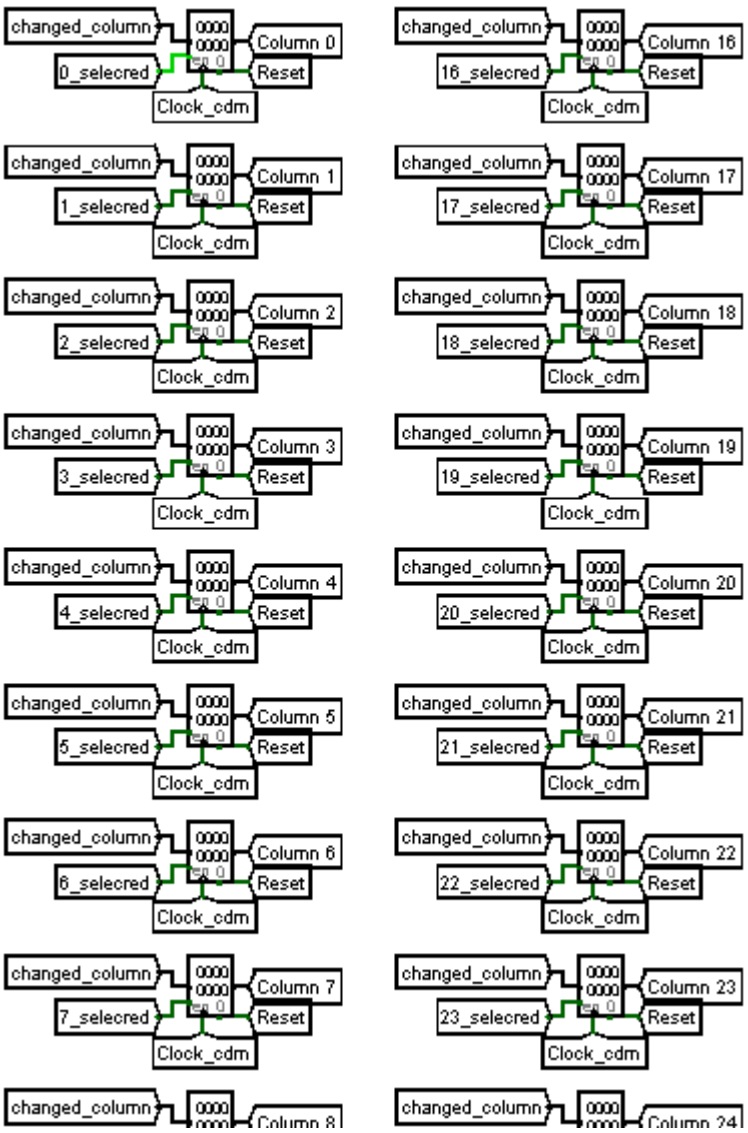
3.



There are 3 parts to this scheme. The first one looks at the current cursor position at x coordinate and puts a flag on the corresponding x column. Then the second part receives this

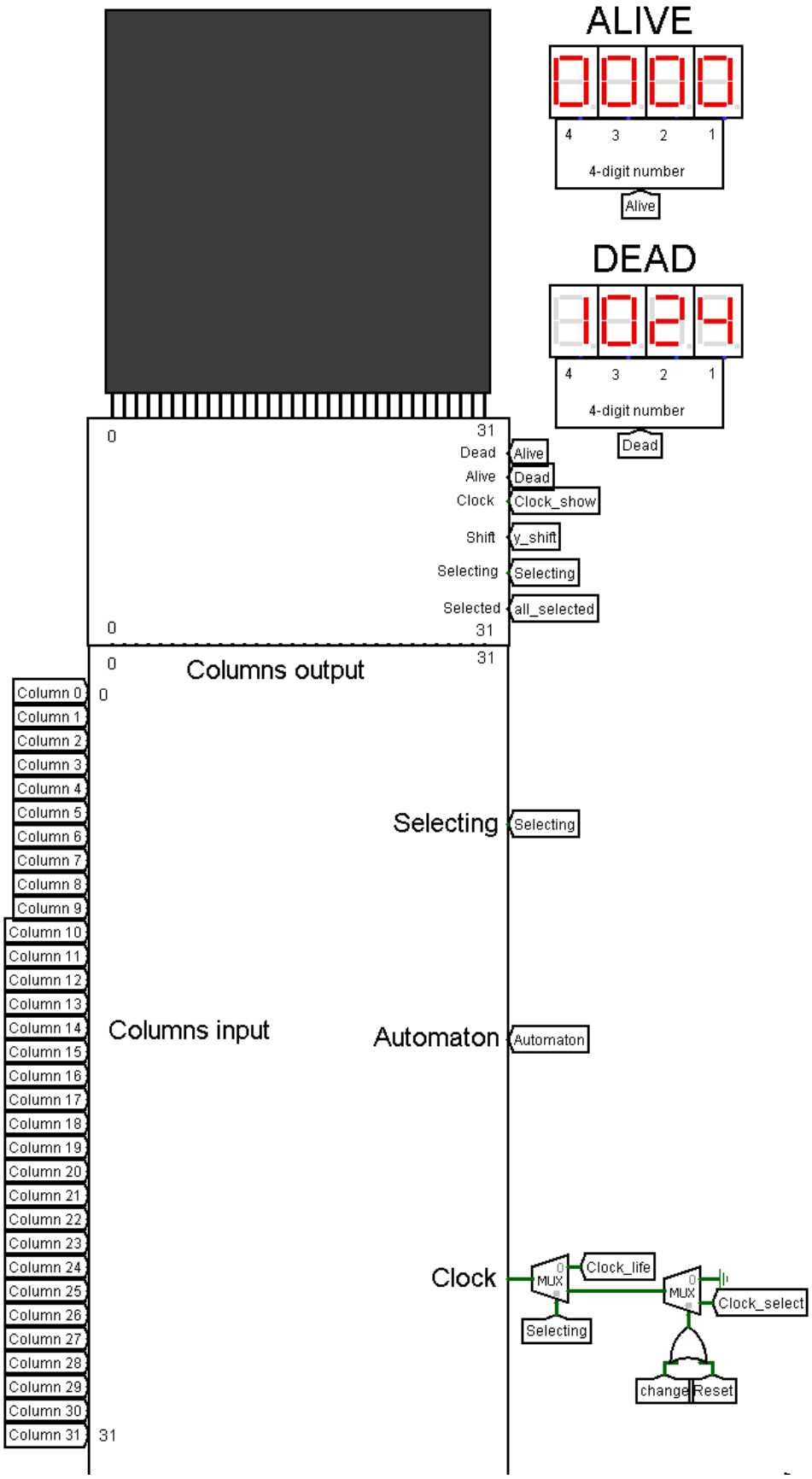
flag and sends it to the all_selected tunnel, which is one of the parts of the cursor display system. Part 3 connects the data from the column where the cursor is now and sends it to the column_out tunnel. The tunnel itself is used to change the data during the initial pattern placement.

4.



This part of the circuit is responsible for storing the initial location of the pattern. Each register has 5 tunnels connected to it. changed_column replaces the contents of the register with a new column if the user has changed it in some way. n_selecred is a flag that is needed so that the column is updated only when the cursor is in it. Column n connects to 32x32 cells, passing the initial location of the pattern.

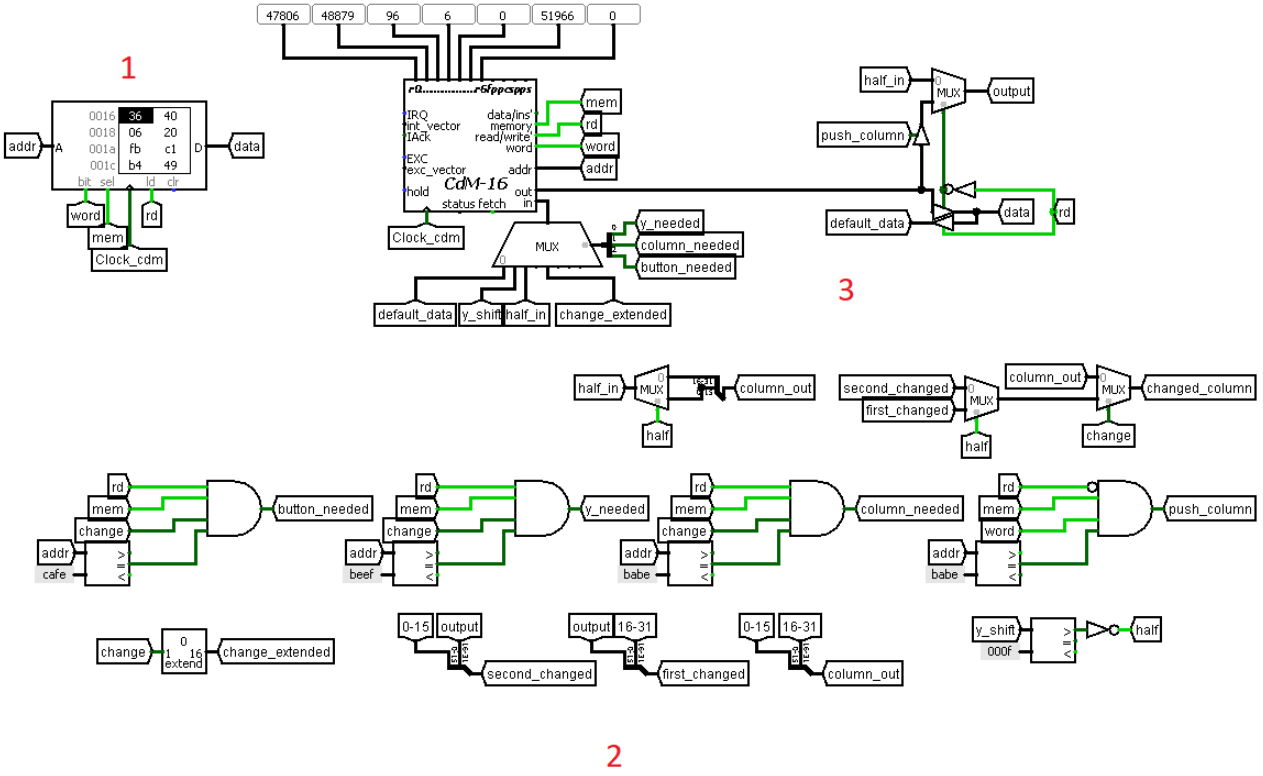
5.



Here we have a 32x32 led display, through 32 32-bit columns inputs goes data with initial placement pattern. From 32 32-bit columns input goes data for correct cursor displaying.

y_shift and all_selected can be thought of as coordinates of the cursor. It also provides a mechanism for selecting and connecting the currently needed clock.

6.



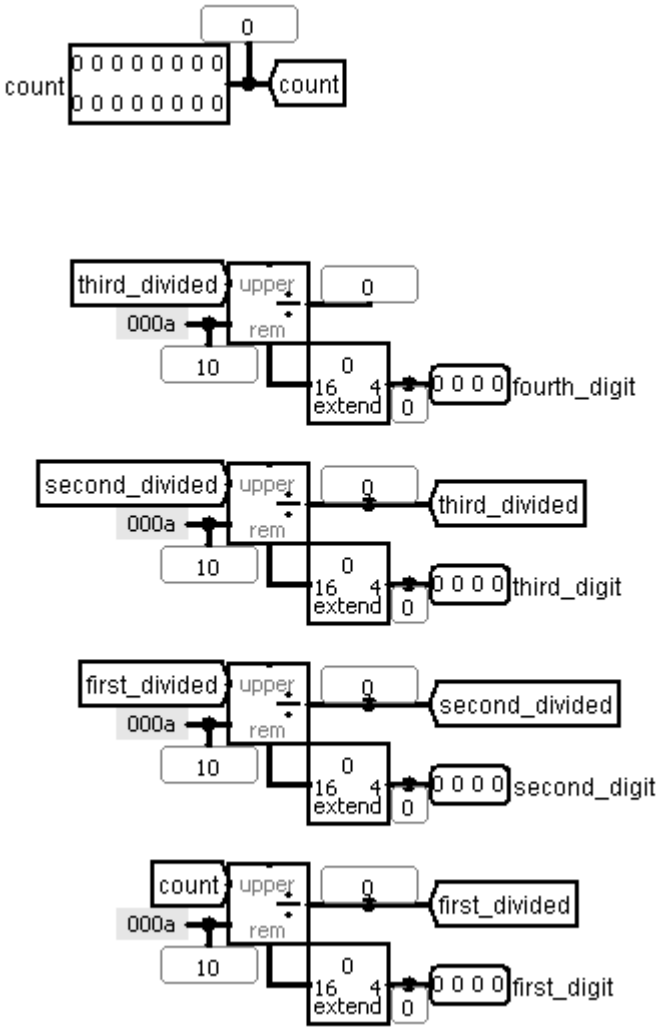
6.1 Overall we made connection of cdm-16 with RAM as in default Von Neumann configuration.

6.2 At the top here we have 4 flags, which are given to the processor for correct operation of its calculations. The necessary condition for their application is that the address in RAM matches a certain address, which is written in the Software part. button_needed indicates that the processor needs to apply an input change, i.e. put the living cell in the initial placement of the pattern. y_needed indicates that the processor needs to input the y-coordinate. column_needed indicates that the processor needs to input data from the correct half of the column. push_column is needed to show that the processor is ready to upload the processed data to the output.

A little lower down we have a mechanism for determining the needed half of the column. Since the cdm-16 is a 16 bit processor, it cannot process 32 bits from the whole column, so we divide the original column_out into two parts 0-15 and 16-31 and select the needed one using a comparator. Also here we connect the changed part of the column to the unchanged part of the column, for further use in the processor section. Also here we have the change extension, since it is originally 1 bit, but we need 16 bit.

6.3 First here we define the half_in data, it contains information about the half of the column we need. A multiplexer is connected to the processor input, which sends data to it depending on the flag defined in 6.2. After that the processor output is sent to the output tunnel, if the push_column flag is set. Determining whether to send data to output now or not is done with the rd flag, which indicates whether the program is reading memory now or not. As a result, value of the half determines which part of the column has changed and this data goes to the changed_column tunnel, from where it is sent to the registers.

7.



Inputs:

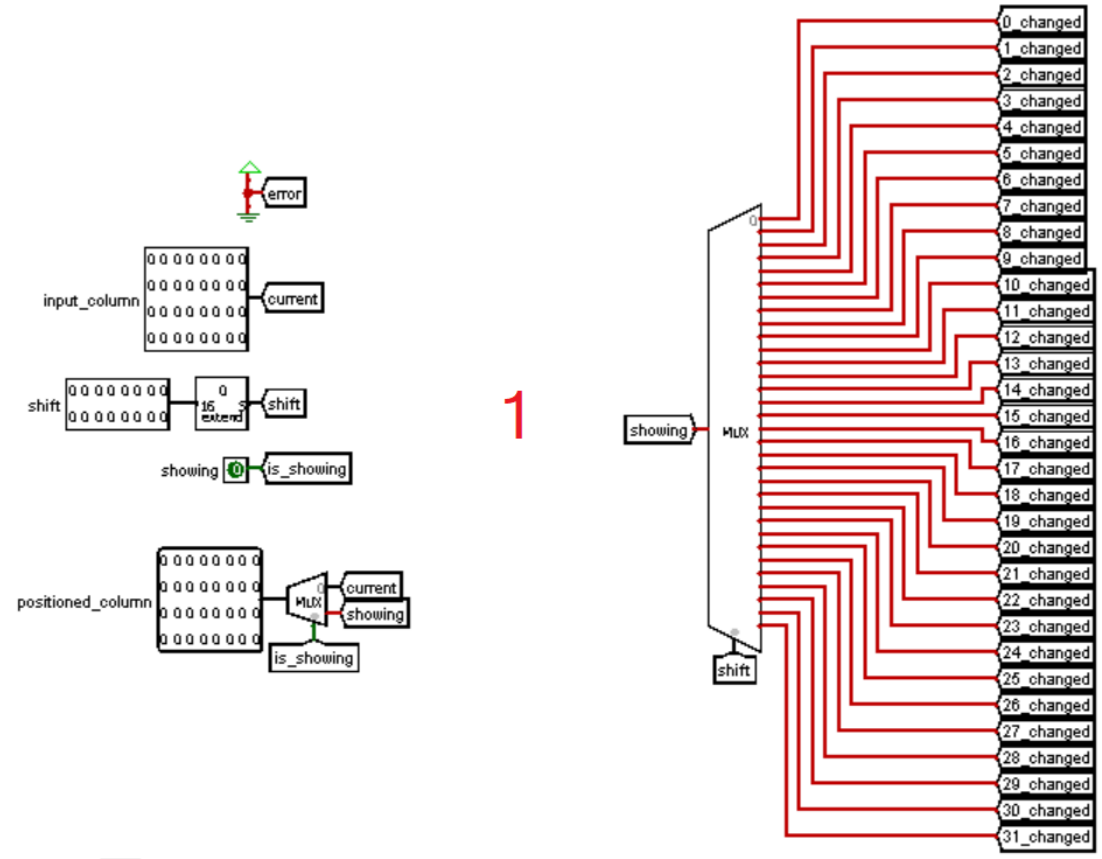
- 32-bit input for column

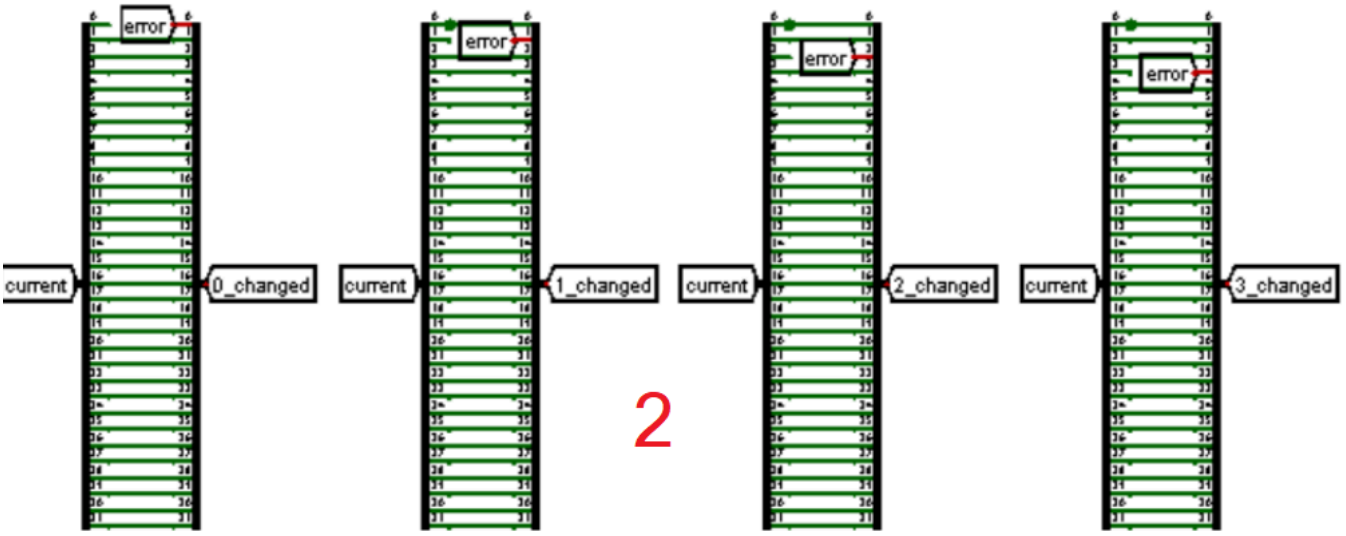
Outputs:

- 4 4-bit outputs for each digit

16-bit input for a number is divided by 10, the remainder of the division is the first digit, the divided result is divided by 10 again, the remainder of the division is the second digit. Such an algorithm goes for 4 times overall to split a number into 4 digits.

Position showing





Inputs:

- 32-bit input for column
- Flag, which means is column selected or not

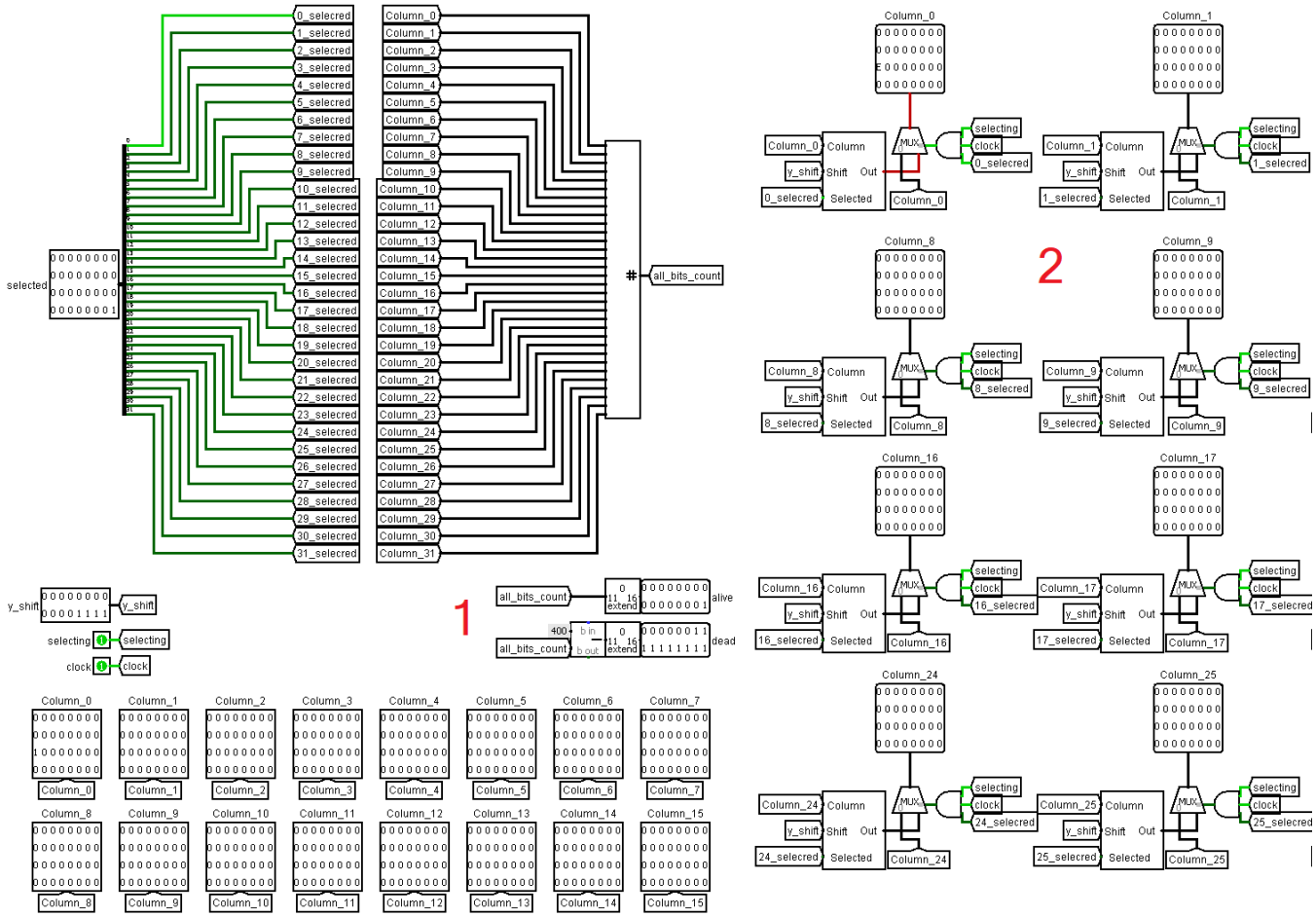
Outputs:

- 32-bit output for column with highlighting

1. Based on the shift, this scheme is choosing, which bit is now selected and should be provided.

2.Based on the number of selected cell, this scheme is changing selected cell's state to the error.

Position transfering



Inputs:

- 32-bit input for the selected column flag
- 1-bit input for the selecting flag
- 1-bit input for clock
- 16-bit input for the Oy shift
- 32 32-bit inputs for columns dat

Outputs:

- 32 32-bit outputs for columns data, one of which can show a cursor position
- 2 16-bit outputs for dead and alive cells count

1. All columns bits are being summarized and the result is alive cells count, and “1024 - result” is dead cells count.

2. Based on clock, selecting and selected flags the output will be an original column data or Oy shift bit replaced with error to see a cursor current position.

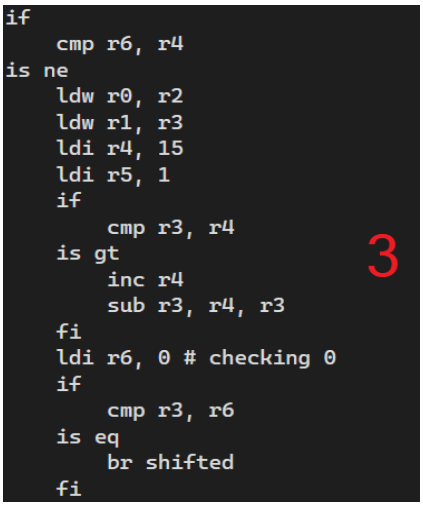
4. Software

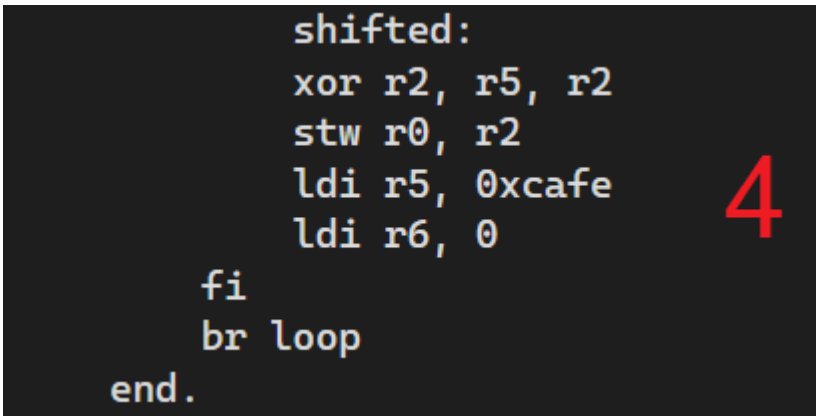
```
# r0 - column address
# r1 - y shift address
# r2 - column
# r3 - y shift
# r4 - firstly previous state, secondly 16 comparison
# r5 - firstly button address, secondly shifted res
# r6 - firstly button condition, secondly shift count
```

1

```
main:
    ldi r0, 0xbabe # column
    ldi r1, 0xbeef # y shift
    ldi r5, 0xcafe # button adress
    push r4
    loop:
    ldw r5, r6
    pop r4
    tst r6
    push r6
    bz loop
```

2





1. Registers usage in program.
2. Loading column number, y-shift and current change button state to the certain addresses. Then pushing the first button state to the stack. After that, in the loop loading the current button state and taking the previous button state from the stack. If the button's state is 0, it is not being pressed, and we should do nothing, so we go to the beginning of the loop.
3. If button state is not 0, we don't go straight to the beginning of the loop, because it means that button is being pressed. Firstly, we are comparing previous and current button states, if they are equal we should do nothing, because button is being holded, so we do not go to other code and after end of if-fi construction we go straight to beginning of the loop. Secondly, we are comparing y shift with 16, cause our registers have only 16 bits, so we are splitting our column into 2 parts, and if y shift is bigger than 15, we should decrease it's coordinates by 16 to have appropriate bit number for register. Then we are checking y shift 16 times to understand what the Oy shift equal is. Then, we go to the shifted subroutine.
4. There, when we know the Oy shift, we can finally XOR our column with bit word, which has 1 at the y shift position and 0 on the other places, to change a certain cell's state. After that we go straight to the beginning of the loop.

5. Interaction between Hardware and Software

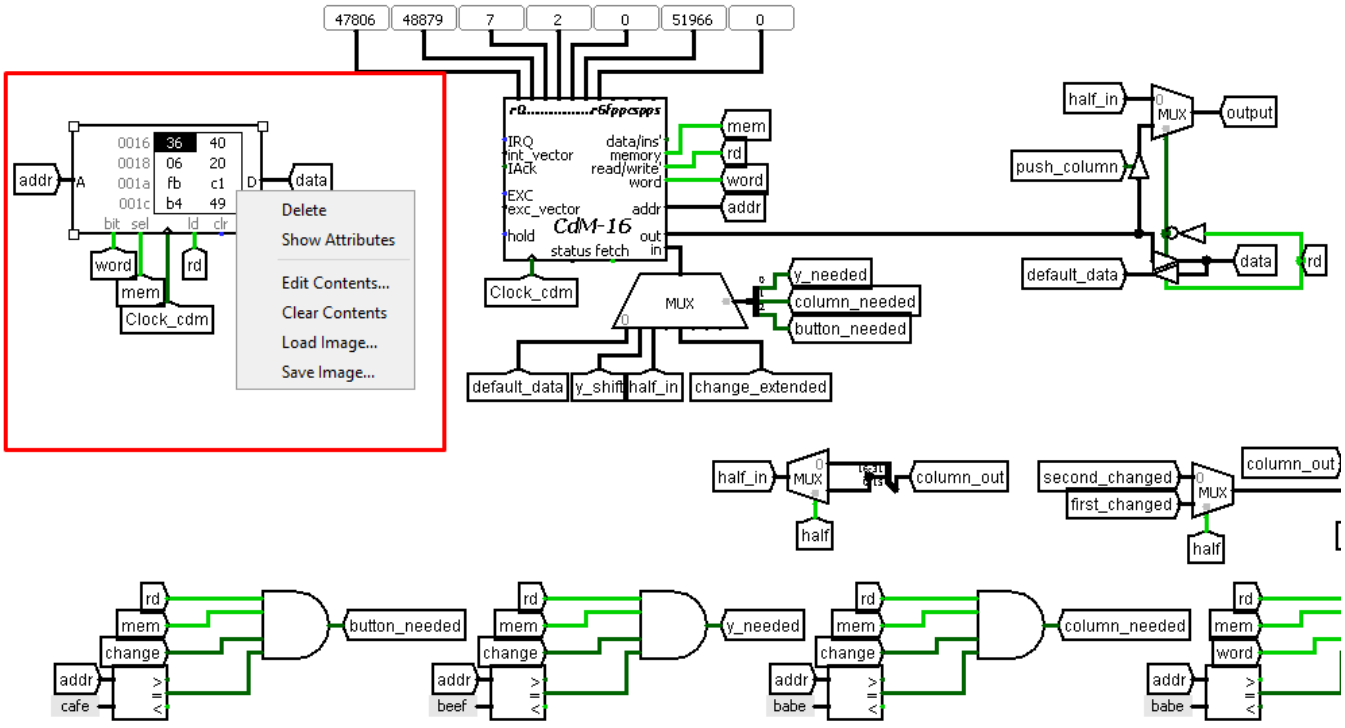
Generally, Software is used to change the initial pattern for a game. In an infinite loop, CDM-16 compares the change button current and previous states. If the button is pressed now and was not pressed previously, CDM-16 loads half of a column data from Logisim. Then, it loads the Oy shift from Logisim, XOR'es half of a column with "1 << Oy shift" and stores the result to

output. Afterwards, changed half of the column data replaces the old half, and the new data for the initial pattern gets updated in a register.

6. User Manual

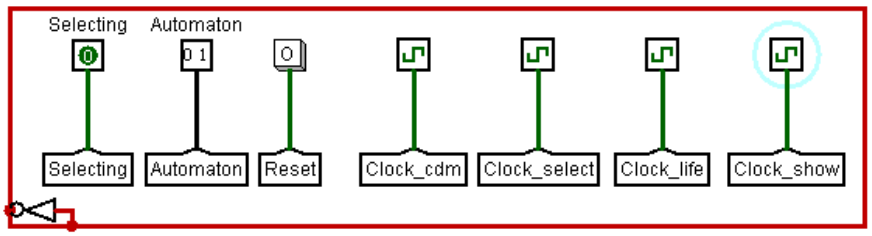
These are the steps to run our game:

1. Load image file joystick_optimized.img into RAM.



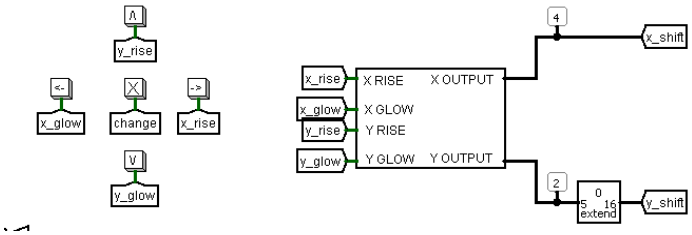
2. Enable ticks(Ctrl+K)

3. Choose what type of automaton you want to use using the clue on the right.



00 - Conway's game of life
01 - Maze
10 - H-trees
11 - Persian carpet

4. Set the Selecting flag to begin marking the initial pattern with a cursor.
5. Control the cursor using the buttons, moving it around the field by x and y coordinates. Mark a cell as alive using the "change" button (hold it for 0.5 seconds).



6. When you have decided to finish the initial pattern placement, unset the Selecting flag and the program will start the gameplay itself. If you want to pause or clear the field, select the Selecting flag and continue placement. If you press the "Reset" button, the field will be cleared.

Conclusion

First of all, we implemented the Conway's game of Life, which will go in our portfolio. We also modified the project by adding 3 more cellular automatons, increasing the game field by 4 times, alive and dead cells counter and cursor position showing. Secondly, we gained extremely valuable experience of working in a team on a large project, which will definitely help us in our future professional career. And finally, of course, technical skills. We greatly improved our understanding of working with Git, Logisim and Assembly language. Overall, it was a very useful experience for us.