**Documentation**

**“Game of Life” implementation**

Content:  
Introduction………………………………………………………………………....pg 1

Rules…………………………………………………………………………….…..pg 1

Why……………………………………………………………………………...….pg 1

Main part……………………………………………………………...……….……pg 2

Software…………………………………………………………………………….pg 2-3

Hardware……………………………………………………………………………pg 4-7

Conclusion…………………………………………………………….…………….pg 7

**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. A person interacts with the Game of Life by creating first generation of cells and watching how it develops.

**Rules**

Game of Life’s play space is a 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 horizontally, vertically, or diagonally bordering. At each tick of time, the next rules are being implemented:

1) If live cell have two or three live neighbors, it survives.

2) If dead cell have three live neighbors, it becomes a live cell.

3) Other live cells die in the next generation. All of other dead cells stay dead.

Each generation is a pure function of 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**

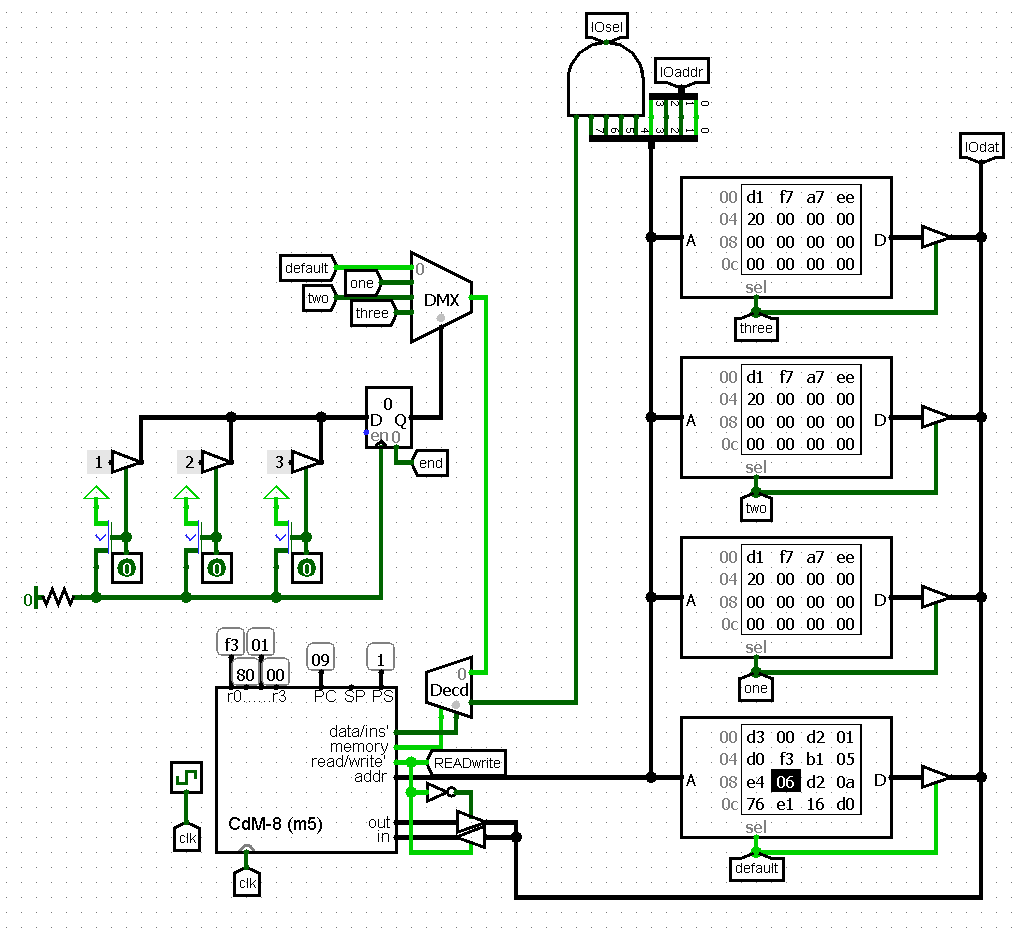
Our implementation of the game allows users to change cell states at any generation and observe the game process on a screen. The project consists of two interconnected and interacting parts:

* The hardware part of our project consists of circuits designed in Logisim. The hardware is responsible for processing game logic and displaying information on the 16\*16 LED screen.
* The software component of our project is implemented on a “CdM-8-mark5” processor. We used modified Assembler language code to implement the software, which is responsible for scanning user input. Specifically, the software handles input from a keyboard, which allows users to change cell states at any generation.

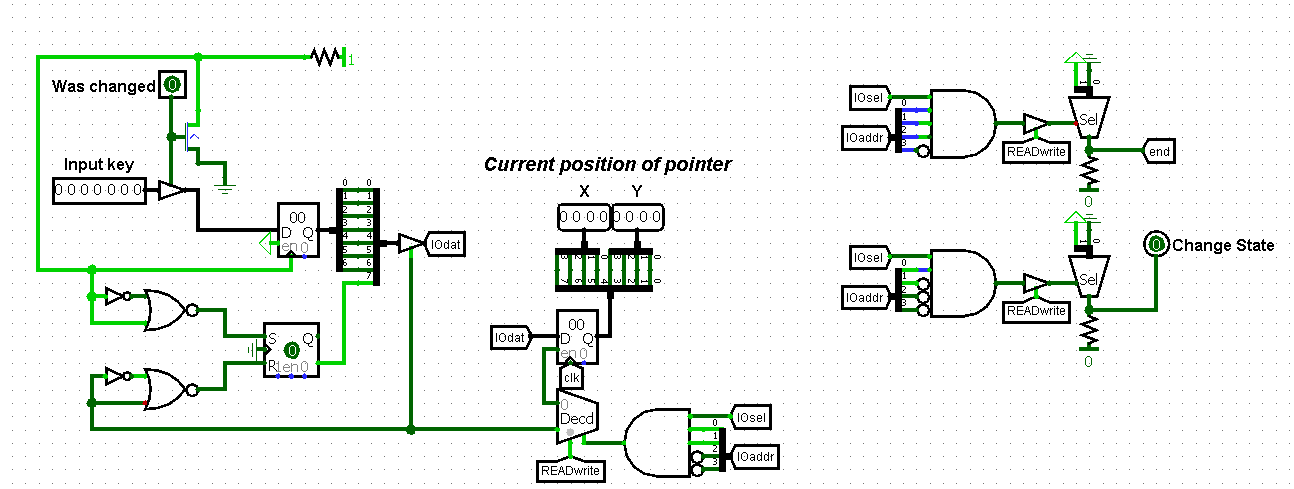
In our project, we used the von Neumann architecture, which is a computer architecture based on the principle of stored-program computers. The architecture uses a joint memory for both code and data, which is sufficient for our project because the software component only needs to handle data input from the keyboard.

User can:

* Change coordinates of pointer and move it by entering ‘w’, ‘a’, ‘s’, ‘d’ letters to keyboard. Place/remove alive cells on game field, using Enter in keyboard.
* Run/stop game, switching **“Run”** pin. While this pin toggled, user can’t input information, game logic processing in this moment.
* Create pre-installed generations in game field, using **“1”**, **“2”**, **“3”** buttons. It executes instruction to pointer: change cells state in certain coordinates. If game started, shapes will appear when game stopped.
* Change game speed, using pins connected to **“Speed”** circuit. Each toggled pin increases the speed of game.
* Clear screen, pressing **“Clear”** button. Works at any moment, makes all cells dead.

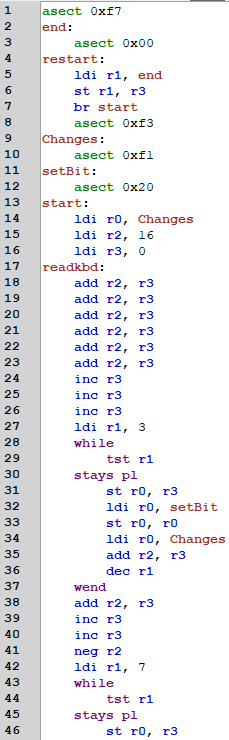
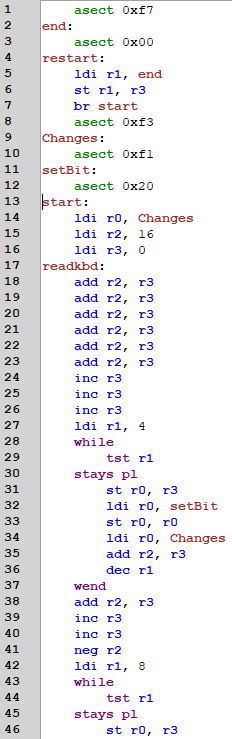
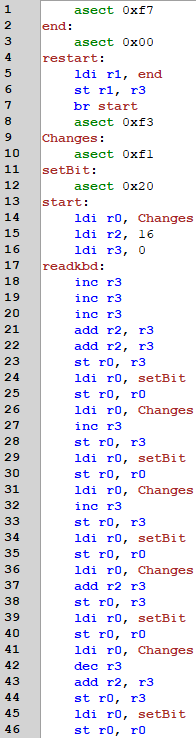
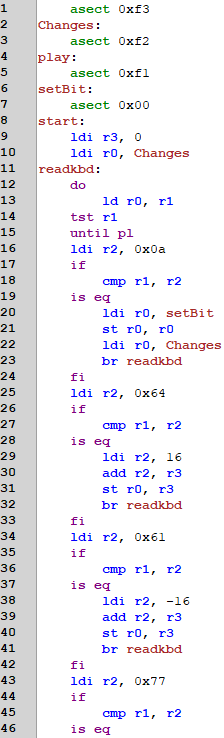
**Software**

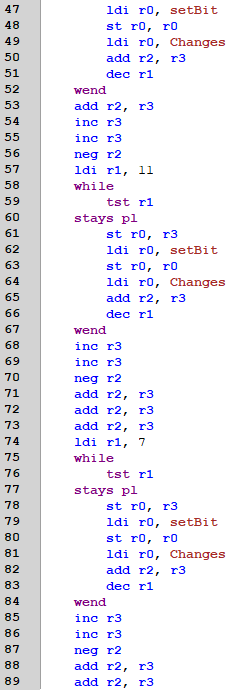
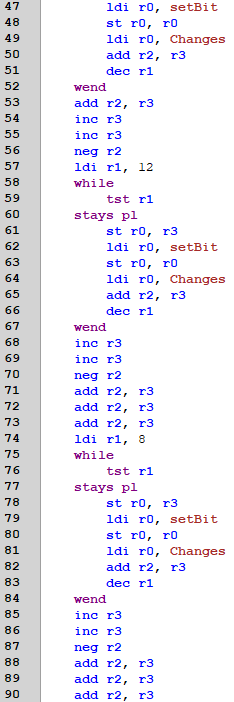
The program takes an ASCII character code as input. There are only five valid code options: 'a', 's', 'w', 'd', and a space(enter on keyboard). If the program receives any other options, it will not perform any action. Based on the code received, the program will either move the pointer to a different position or set a flag indicating a change in the state of the current cell. The program only stores the current coordinates of the pointer, which are initially set to (0, 0) representing the upper-left corner of the matrix.

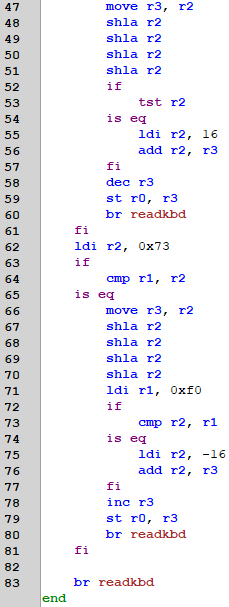


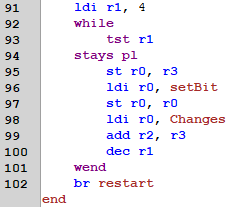
**Assembler code:**

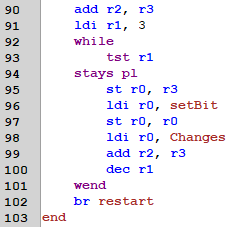
**inputByCdm.asm pattern.asm pattern2.asm pattern3.asm**

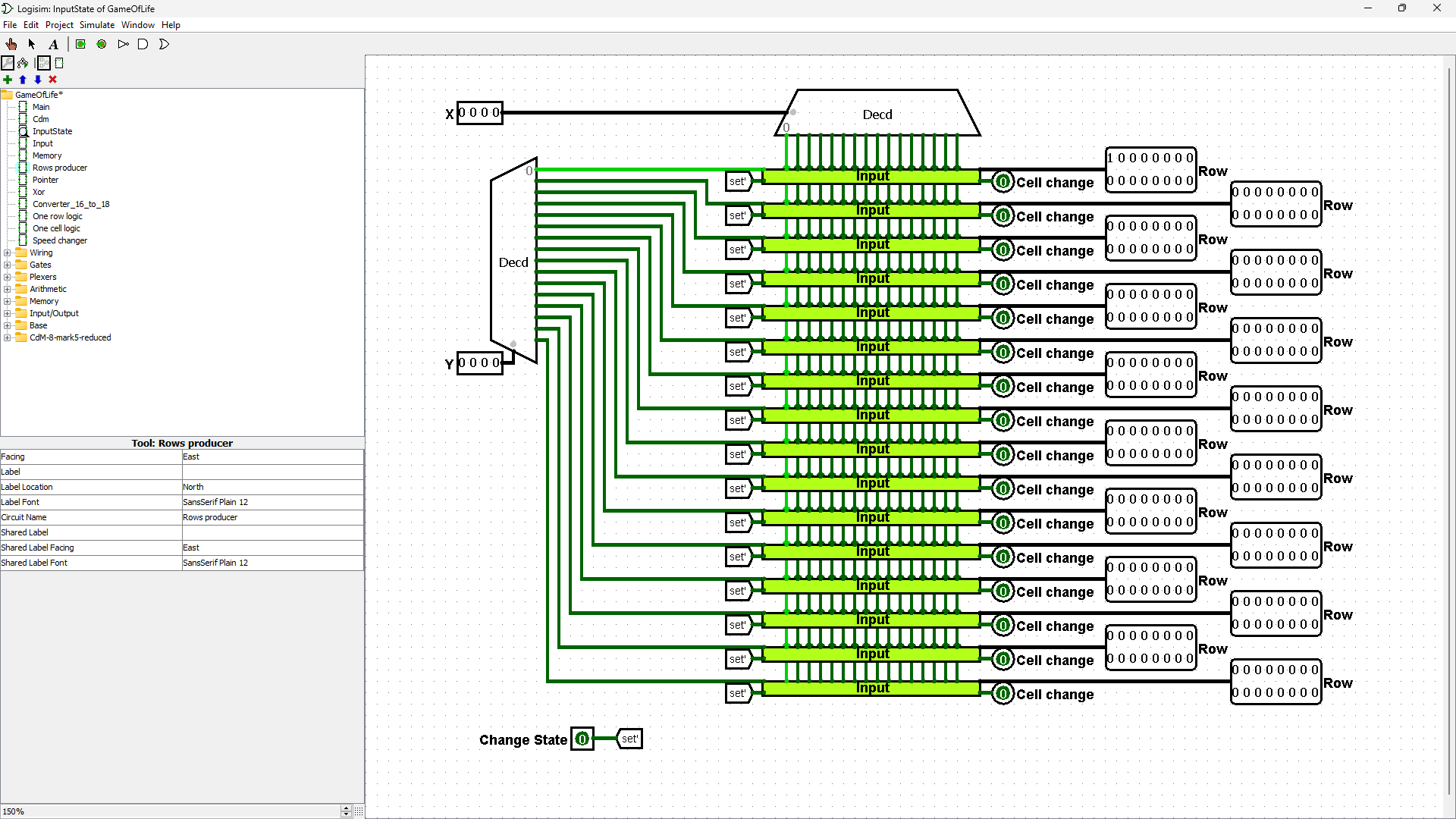










**Hardware**

**“InputState”**

Circuit input:

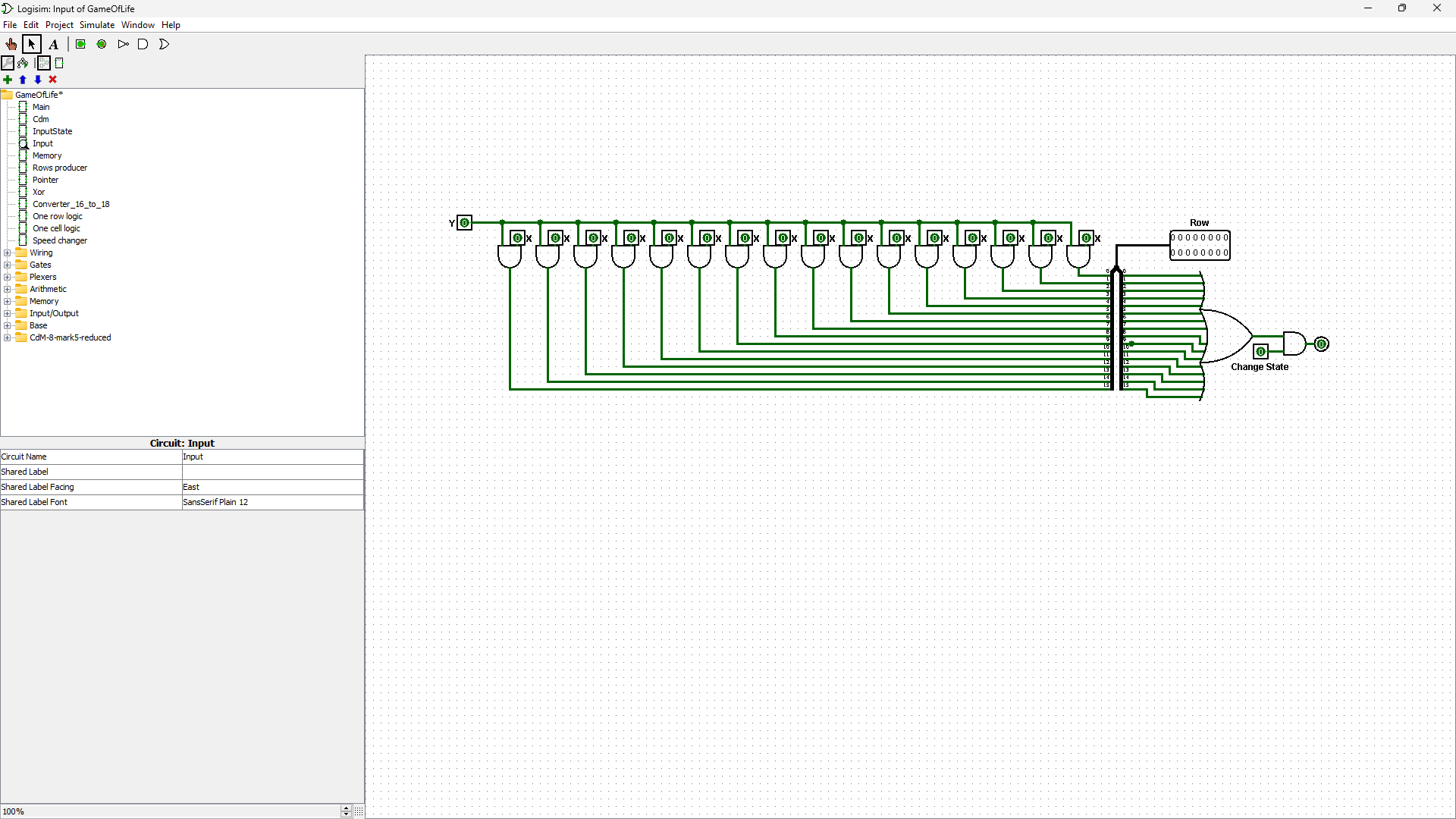
* **“X” “Y”** – 4 bit coordinates (from **“Cdm”**)
* **“Change state”** – 1 bit instruction from **“Cdm”**

Circuit output:

* **“Row”** x16 – 16 bit rows (**“Input”** result)
* **“Cell change”** x16 – 1 bit instruction, (**“Input”** result)

Behavior:

The pointer coordinates are sent to the **"InputState"** 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 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.



**“Input”**

Circuit input:

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

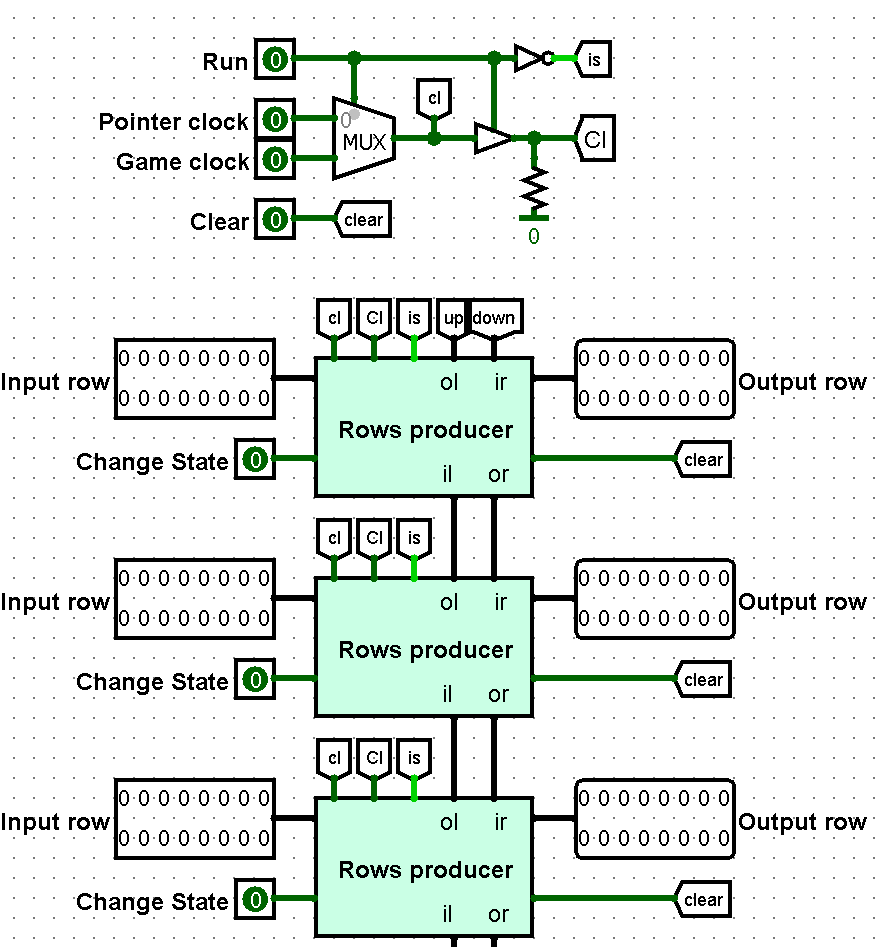
Circuit output:

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

Behavior:

Next we have 2 coordinates, so we use **“Input”** circuit for every bit from Y to concatenate with X, if Y bit toggled there will be toggled bit in 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.



**“Memory”**

Circuit input:

* **“Run”** – 1 bit, toggled to start game (from **“Main”**)
* **“Pointer clock”** – 1 bit (from **“Main”**)
* **“Game clock”** – 1 bit (from **“Main”**)
* **“Input row”** x16 – 16 bit row (from **“Memory”**)
* **“Change state”** x16 – 1 bit instruction (from **“Memory”**)
* **“Clear”** – 1 bit instruction(from **“Main”**)

Circuit output:

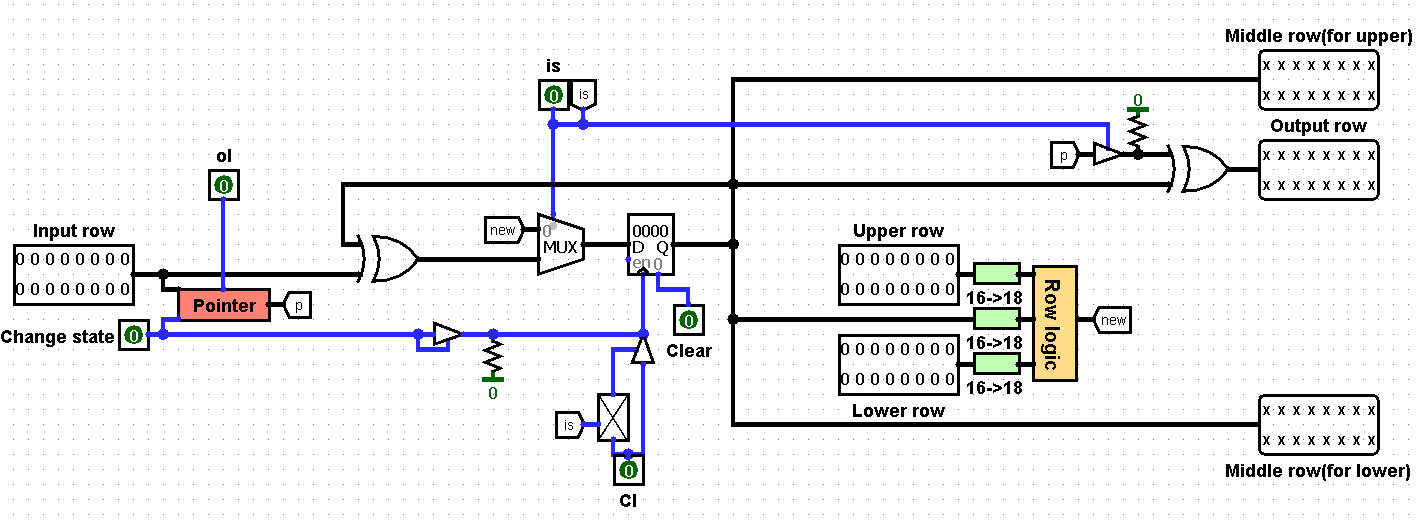
* **“Row”** – 16 bit row with/without pointer

Behavior:

After processing, the output consists of 16 rows, and the pins are changed before sending the data to the **"Memory"** circuit. The **"Memory"** circuit has three additional pins: **"Run"**, **"Pointer clock**", **"Game clock"**, and **"Clear"**. This circuit connects the game and data input parts of the project. Separate clocks are used for the pointer and game, with the pointer clock blinking slowly for better visibility, while the game clock must be much quicker. If **"Run"** is toggled, the multiplexer gives **"Game clock"** to both **"cl"** and **"Cl"**. Otherwise, it gives "Pointer clock" to **"cl"**. Additionally, the controlled buffer gives a value of 0 to **"Cl"**, stopping the game logic when the game has not started and the user is inputting information.

This circuit have next gates:

* **“Cl”** – clock of game logic
* **“cl”** – clock of pointer display
* **“is”** – opposite **“Run”**.
* **“down”** – output from 16th **“Rows producer”**
* **“up”** – output from 1st **“Rows producer”**

We need to process a looped field, so we add **“down”** and **“up”** gates in **“Memory”**, in this way upper row for 1st row is 16th row same on the contrary.

**“Rows producer”**

Circuit input:

* **“Input row”** (same in **“Memory”**) – row that will be changed
* **“Change state”** (same in **“Memory”**) – instruction for pointer
* **“ol”** (from **“Memory”**) – pointer clock
* **“Cl”** (from **“Memory”**) – game clock
* **“is”** (from **“Memory”**) – 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

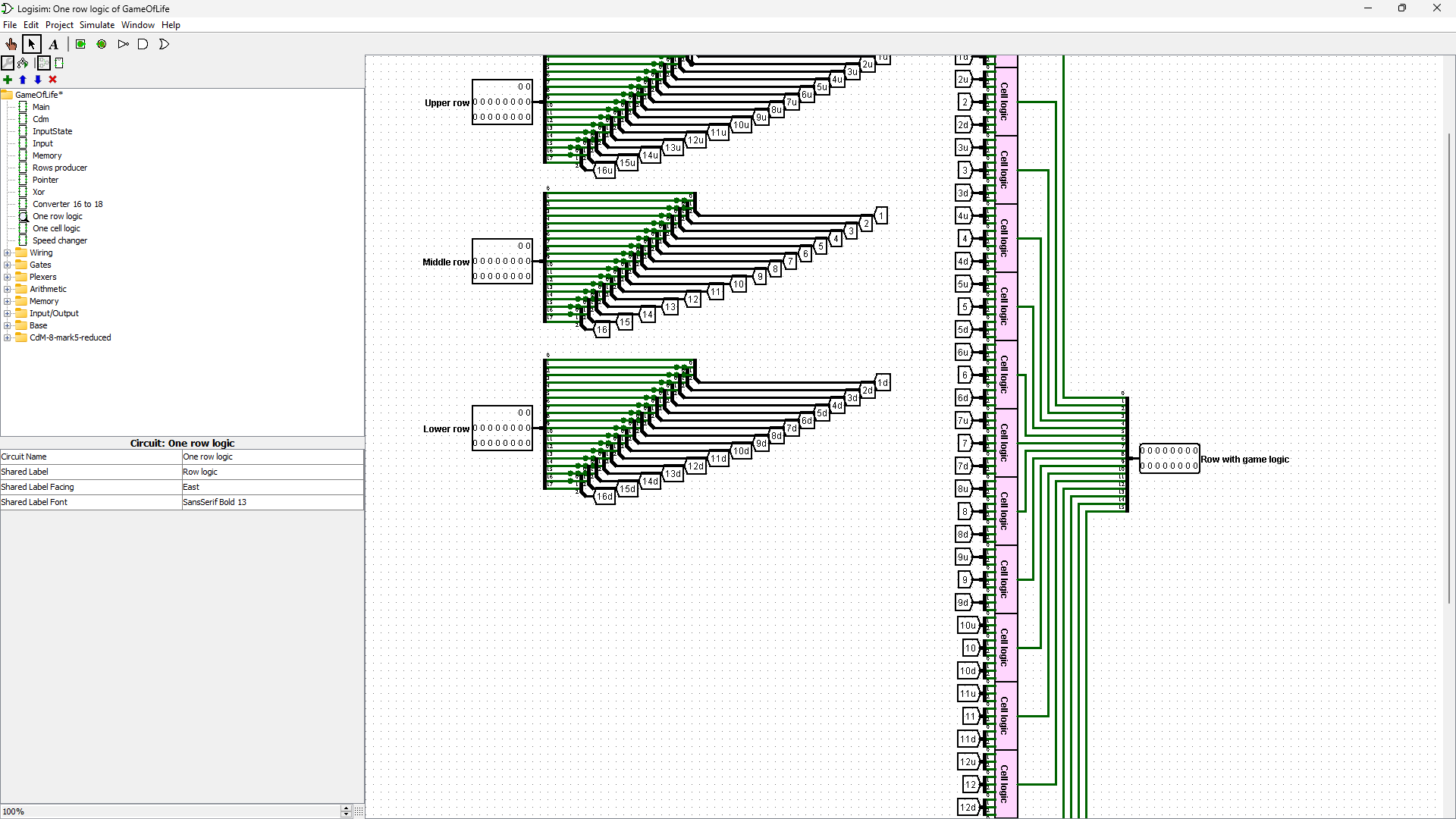
Circuit output:

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

Behavior:

If the **"is"** state is 1, we save the current state of the row in a register, send the saved row to the neighbors, and display the pointer. We also use the **"Pointer"** circuit, which uses an **"And"** operator with suitable input values to make the pointer blink.

If the **"is"** state is 0, we transform the **"Upper row"**, **"Lower row"** and **"Input row"** into 18-bit rows using the "Converter 16 to 18" circuit. This is necessary to implement a loop field, as the **"Converter 16 to 18"** circuit converts a row like "1000000000000001" into "110000000000000011", so that cells from one border have neighboring cells from the opposite border. We then send these 18-bit rows to the "One row logic" circuit, XOR the result with the current state, and display the output.

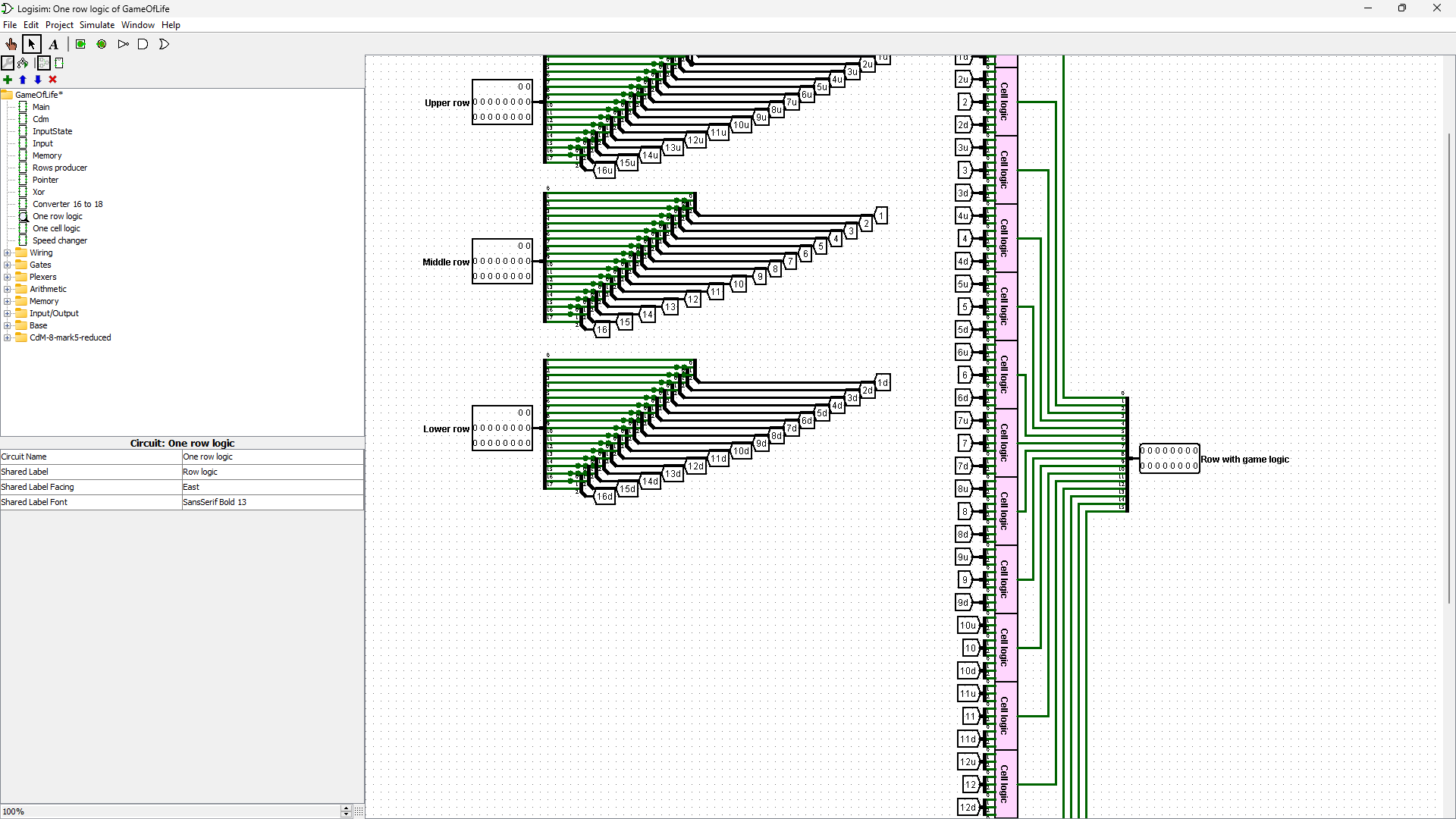
**“One row logic”**

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 we divide cells from rows into groups by three, it is neighbors. Next we send 9 neighboring cell (3 groups) to **“One cell logic”** circuit, it count neighbors for cell in middle of selected groups and change cell state if it must die/alive/stay. All cells calculates in this way and concatenate in one 16 bit row.

**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 their behavior.