**Introduction**

**Information**

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.

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**

We decided to implement “Game of Life” because this game unpredictable. Despite the simple rules, complexity of possible shapes really impress. Also there are a lot of ways to implement this game in different programming languages, so we decided to try make this game with logisim and Cdm8.

**Main part**

There are 2 parts in our implementation of game: hardware part, made with circuits, and software, made with processor Cdm8. We used software to process interaction with game and hardware to implement logic of game and display information. User can place/remove live cells in field with size 16\*16 (LED matrix) with pointer by entering WASD in keyboard to change coordinates of pointer, Enter to place/remove cell and can press button under keyboard to clean input. There is also speed controller (**“Speed changer”** circuit) with input 3 pins, each toggled pin increase speed. And finally there is **“Run”** pin, which executes game logic and hide pointer.

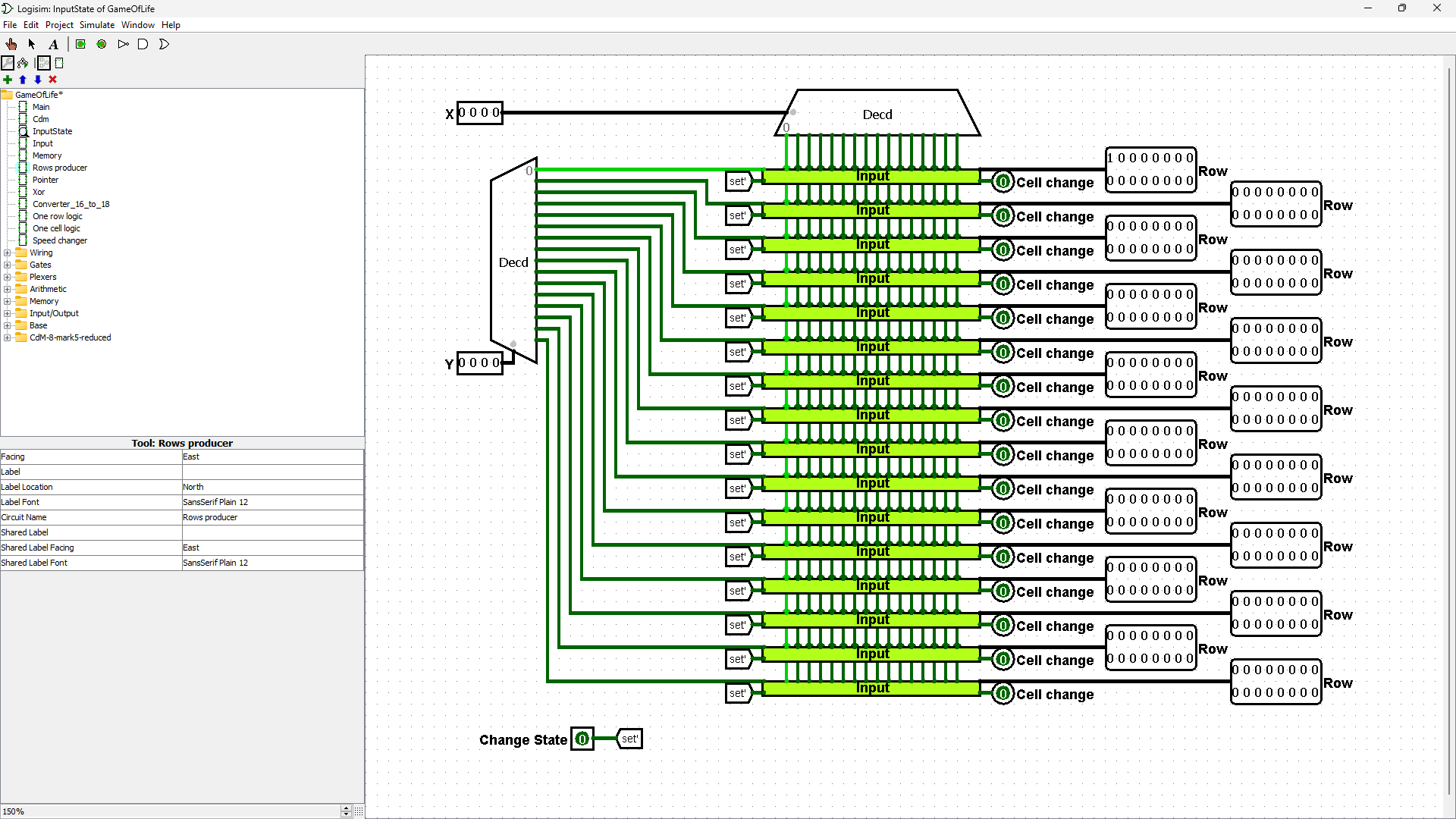
**Software**

**“Cdm”**

To scan input symbols we use keyboard, connected to “Cdm” circuit. From keyboard **“Cdm”** gets 2 inputs: 7-bit number for ASCII symbol (**“Input key”**) and 1-bit flag (**“Was changed”**) which toggled when letter was scanned.

**“Cdm”** output is 2 4-bit numbers: **“X”** and **“Y”** coordinates of a pointer and 1 bit flag (**“Change state”**) which turned on if we need to change cell status in this coordinates(when Enter key was in input). Input character decodes and depending on 7-bit number, which this symbol represents, processor change pointer coordinates or toggle **“Change state”**.

***Hardware***

**“****InputState”**

Circuit input:

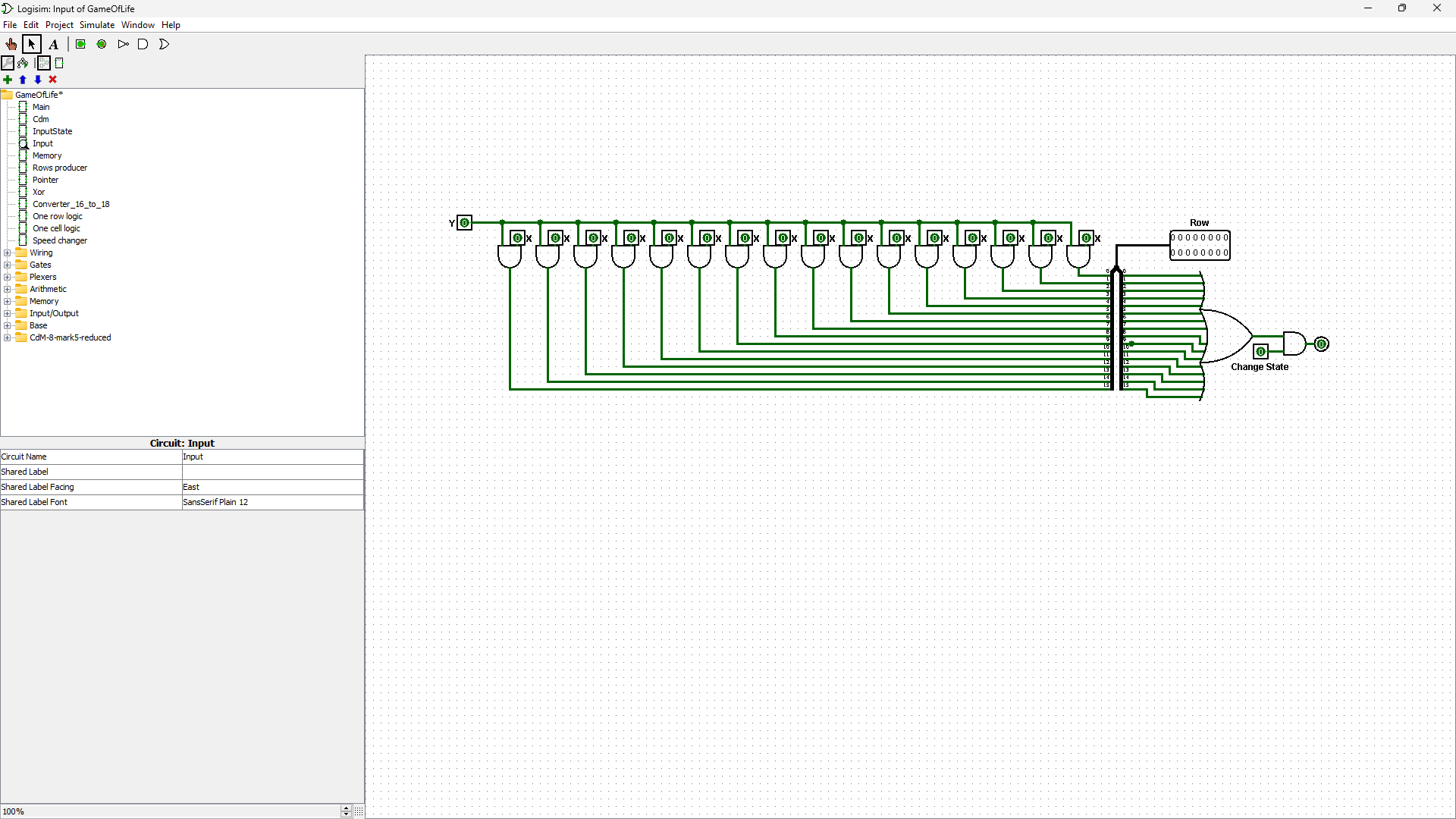
* **“X” “Y”** – 4 bit coordinates (from **“Cdm”**)
* **“Change state”** – 1 bit instruction from **“Cdm”**

Circuit output:

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

Behavior:

From **“Cdm”** pointer coordinates send to **“InputState”** circuit, it converts 4-bit numbers to 16 16-bit rows (field). This field have 0 state in every bit of rows, except pointer coordinates. Decoder convert every possible position of 4 bits in 16 bits, all of them have states 0 excepts one bit, basically it is one row/column of field with one toggled bit – position of pointer.



**“Input”**

Circuit input:

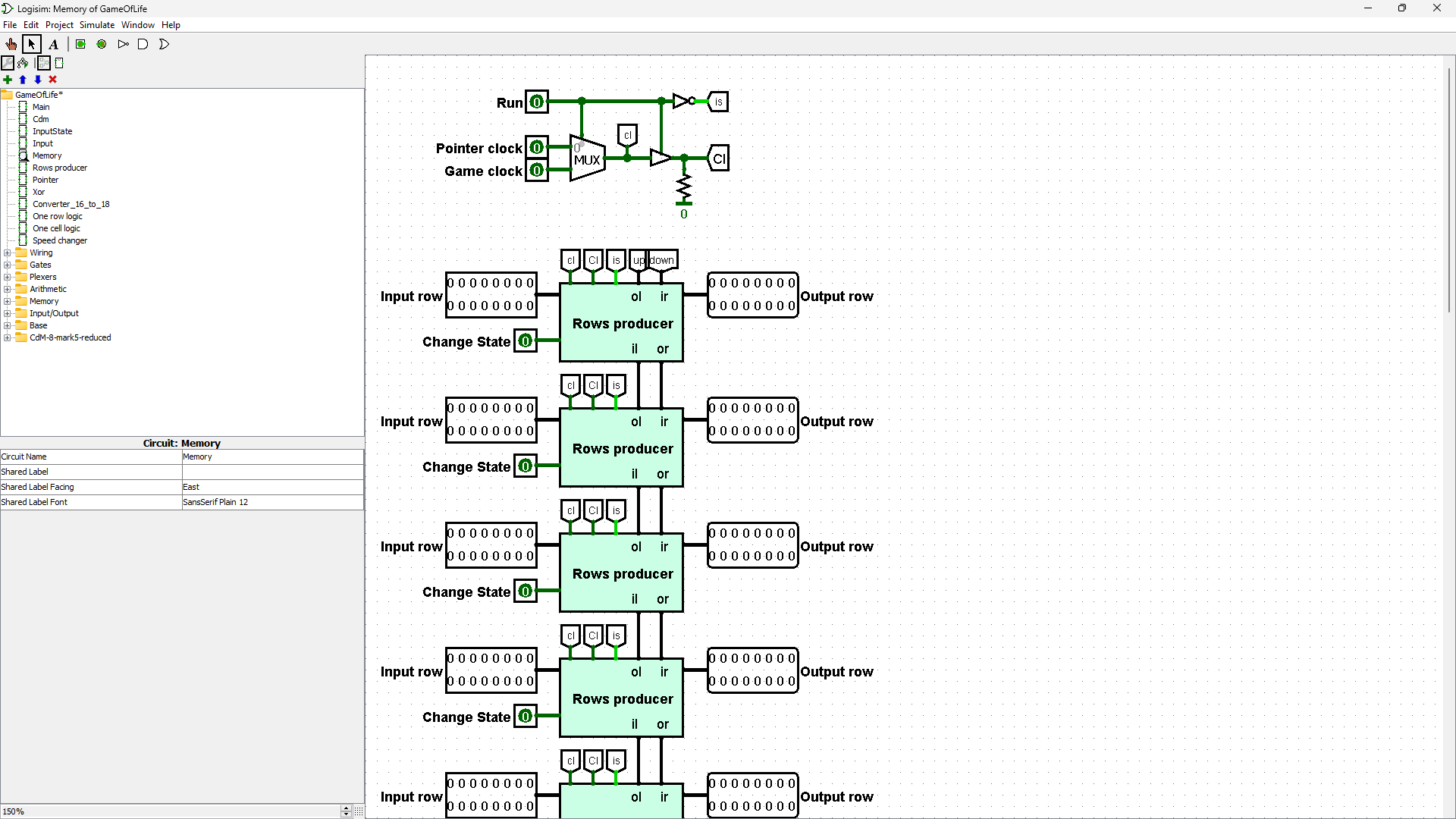
* **“X” “Y”** – 1 bit parsed coordinates (from **“InputState”**)
* **“Change state”** – 1 bit instruction (from **“InputState”**)

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).



**“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”**)

Circuit output:

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

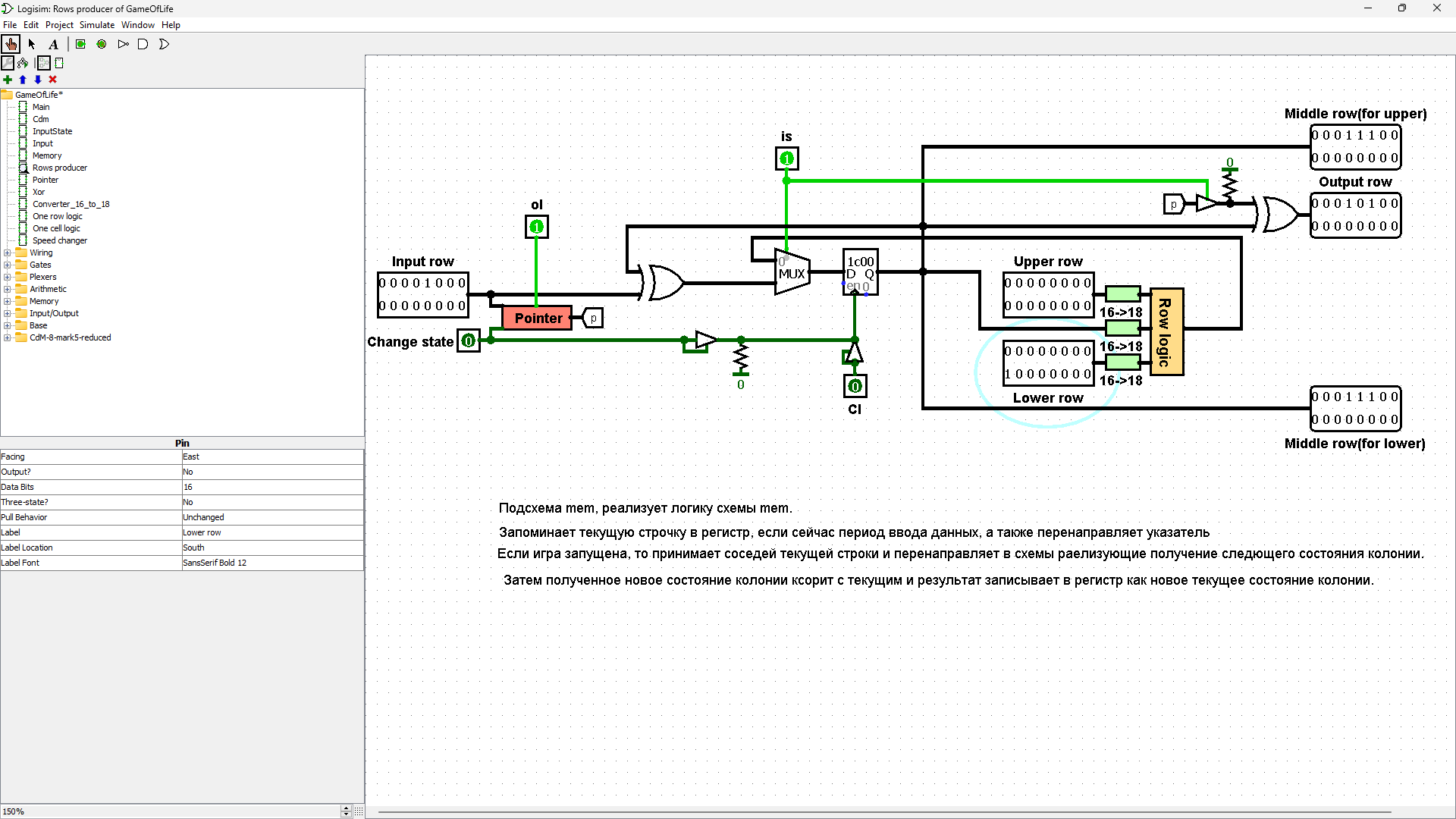
Behavior:

And then in output we have 16 rows and change pins, it send to **“Memory”** circuit with 3 additional pins:**“Run”**, **“Pointer clock”**, **“Game clock”**. **“Memory”** connect 2 states of project: game and data input. We have separate clocks for pointer and game because pointer must blink slow(better visibility), and game must be much quicker. If **“Run”** toggled, **“Memory”** will use ticks from **“Game clock”**, otherwise from **“Pointer clock”**.

This circuit have next gates:

* **“Cl”** – if **“Run”** state is 1
* **“ol”** – if **“Run”** state is 0
* **“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”**

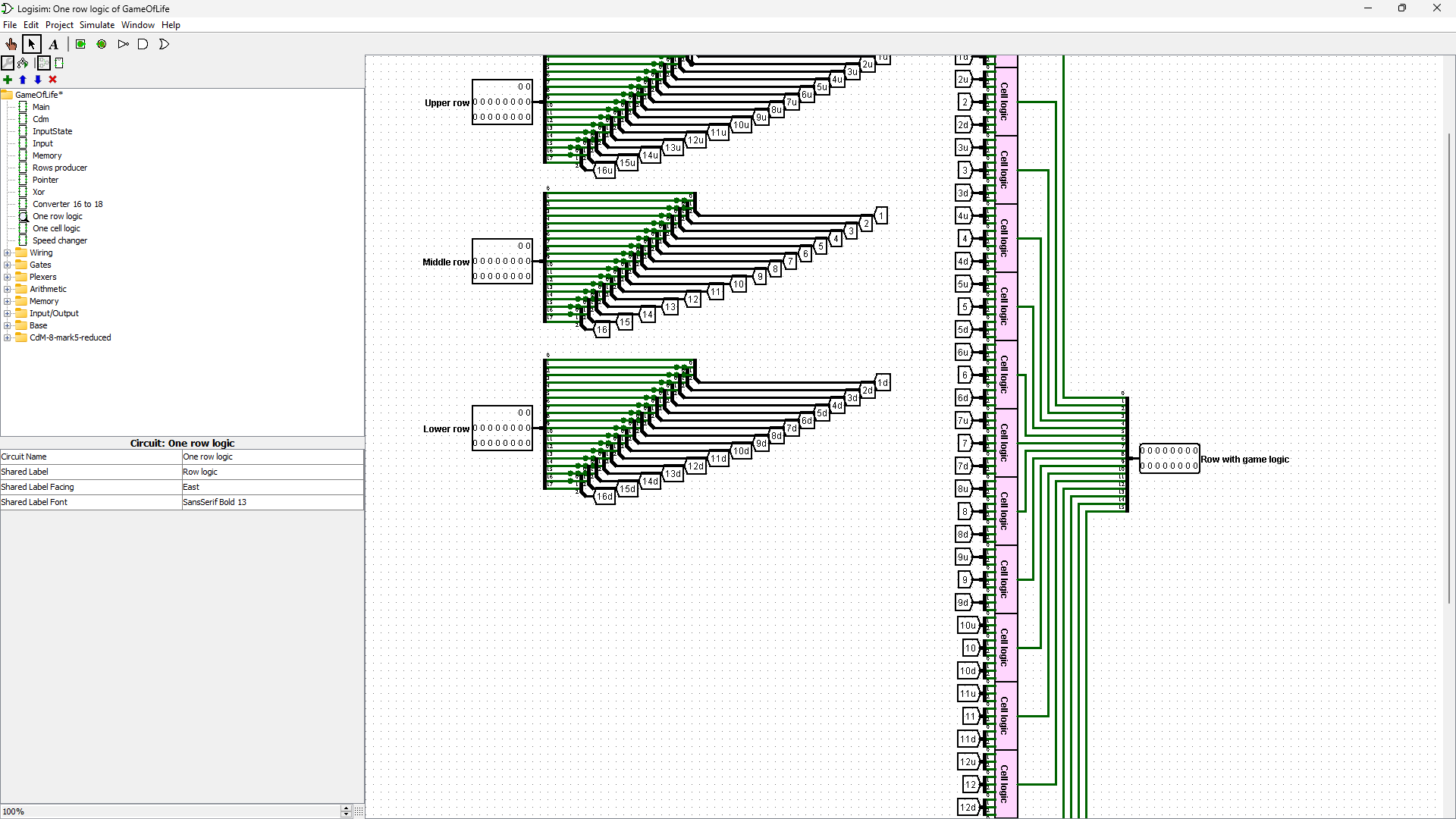
Circuit output:

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

Behavior:

If **“is”** state is 1, we save current state of row in register, send saved row for neighbors, and display pointer. Also we use **“Pointer”** circuit (this circuit make pointer blinking, by using **“XOR”** - circuit which is xor operator with more suitable places of inputs).

If “is” state is 0 we take **“Upper row”**, **“Lower row”** and **“Input row”** and transform them to 18 bits rows with **“Converter 16 to 18”**, Because we need to implement loop field. **“Converter 16 to 18”** converts row “1000000000000001” to “110000000000000011”, so now for cells from one border we have neighbor cells from another border. These 18 bits rows we send in **“One row logic”** circuit, result of it XORes with current state and sends to display.

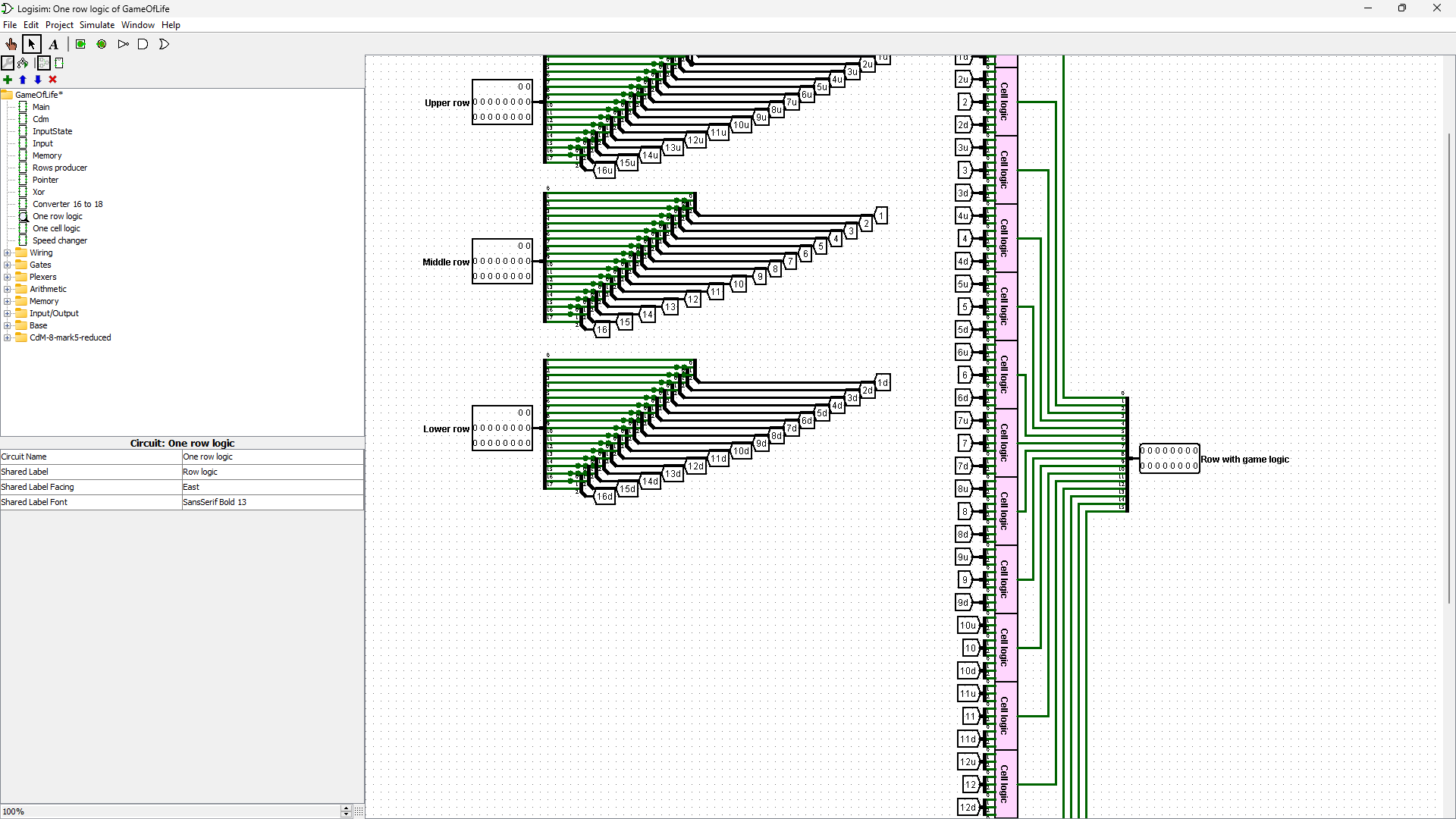
**“****One row logic”**

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.