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): Computer Science and System Design

TEAM PAPER

Authors:
Kotenkov Maksim
Yakovleva Valeria
Zelenin Pavel

Job topic:

“Conway’s Game of Life”

Novosibirsk, 2023

TABLE OF CONTENTS

INTRODUCTION

Let us talk you through our project – “The Game of Life “. “Life” itself was made in 1970 by British mathematician John Horton Conway – it’s a game main feature of which is unnecessary for a player: he only sets the initial states of first cell generation on the plate. Plate consists of a grid of cells which, based on a few mathematical rules, can live, die or multiply. Depending on the initial conditions, the cells form various patterns throughout the course of the game.

1) For a space that is populated

    a) Each cell with *one or no neighbors* **dies**, as if by solitude

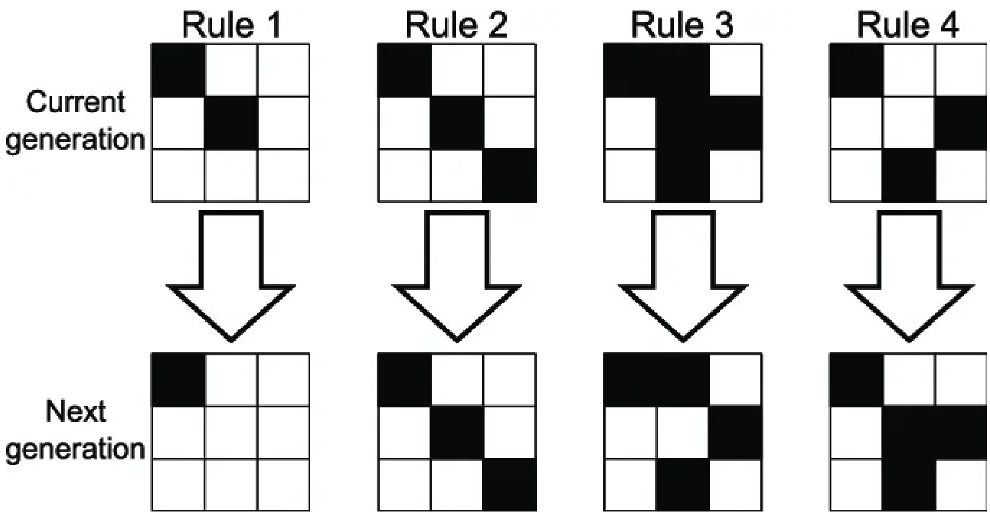
    b) Each cell with *two or three neighbors* **survives**

    c) Each cell with *four or more neighbors* **dies**, as if by overpopulation

2) For a space that is empty or unpopulated

    a) Each cell with *three neighbors* **becomes populated**

More specifically, it does mean *each dead cell with three neighbors* **becomes alive**, *each alive cell with less than two or more than three neighbors* **becomes dead**.



Picture 1.1 – RULES

So, we need to make a display showing our cells, which should change their condition according to the rules, while the game is on.

Also, due to the limited size of memory, as was told in the documentation, we need to handle edge cells. We have decided to make a toroidal plate – it means edge cells are neighbors to the opposite side cells, so the grid is cycled.

1 PROBLEM STATEMENT

Out of the suggested projects, we found “Conway's game of life” to be the most interesting. After doing some research among already existing implementations of this game, we came to the conclusion that the classic version of the game cannot provide a good user experience. In this problem, we saw space for our own ideas and decided to develop the game in Logisim and Assembly, with significant modifications that in our opinion improve the game and make it more interesting.

2 ANALOGUES

Comparing our game with classical version we can find out what makes us different:

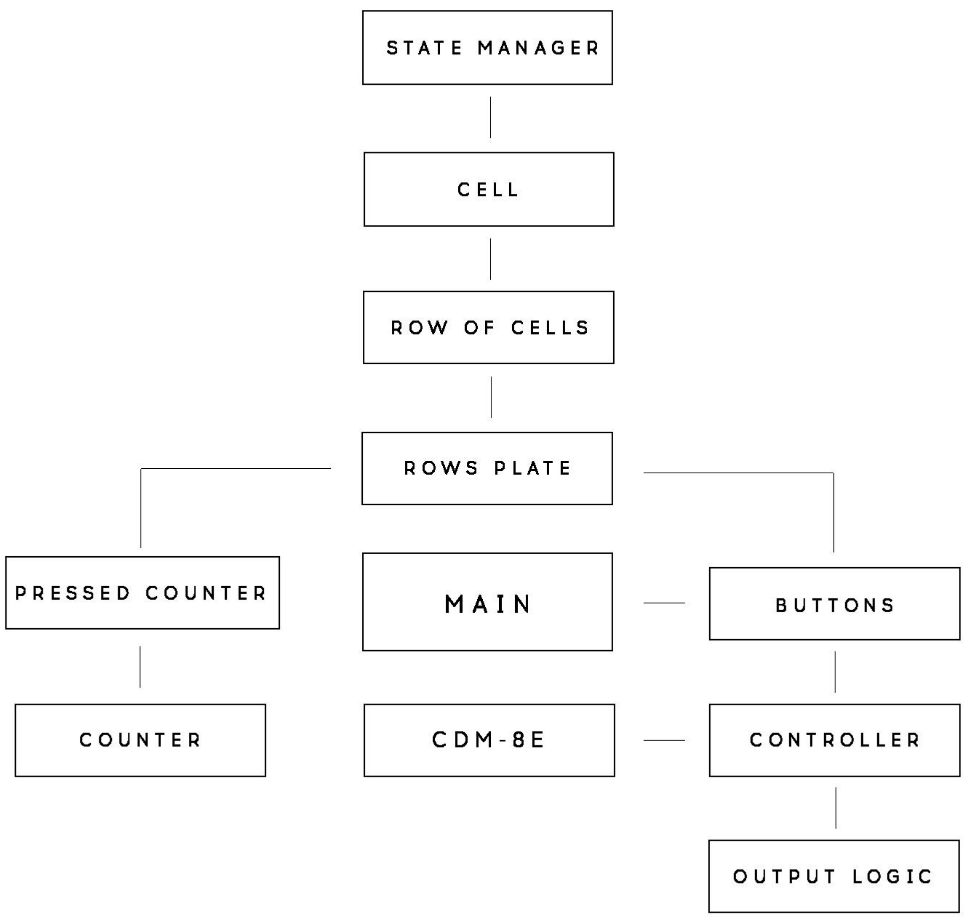
1. Toroidal plate – our cells are neighbors and you can move through the border to another cell on the opposite side
2. Intuitive main scheme - we only have a display, buttons and a few subcircuits. That makes it easy understandable and not so awkward
3. Pattern buttons – using it you can simply and quickly put ready-made blocks on the plate. So there’s no reason to use cursor for placing every cell
4. System control – the possibility to pause and reset the game at every moment. Using relevant buttons you can control simulation

3 HARDWARE

In our project we have 10 schemes including the main one:

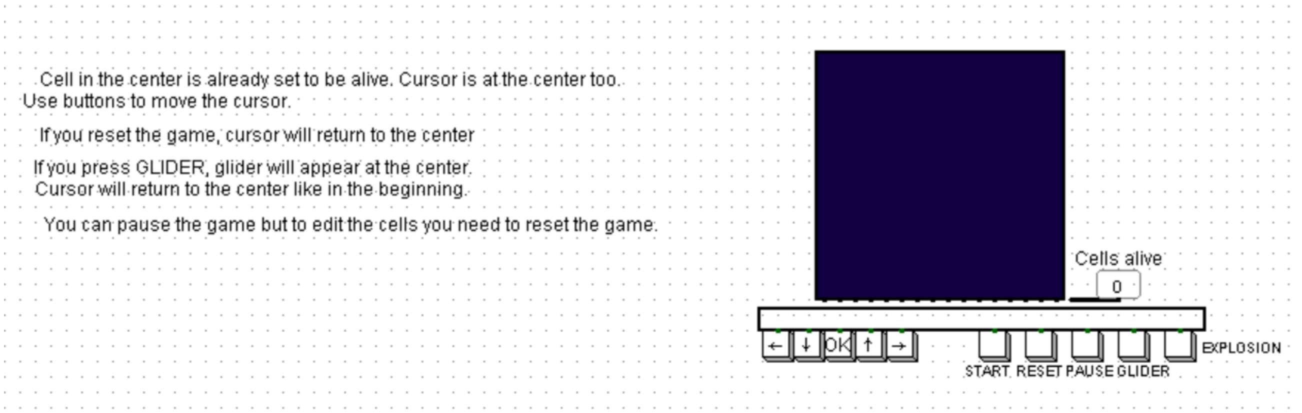
- main
- Cell
- State manager
- Row of cells
- Rows plate
- Controller
- OutputLogic
- counter
- PressedCounter
- BUTTONS

Their interaction is illustrated on the block diagram below:



Picture 2.1 – HARDWARE STRUCTURE

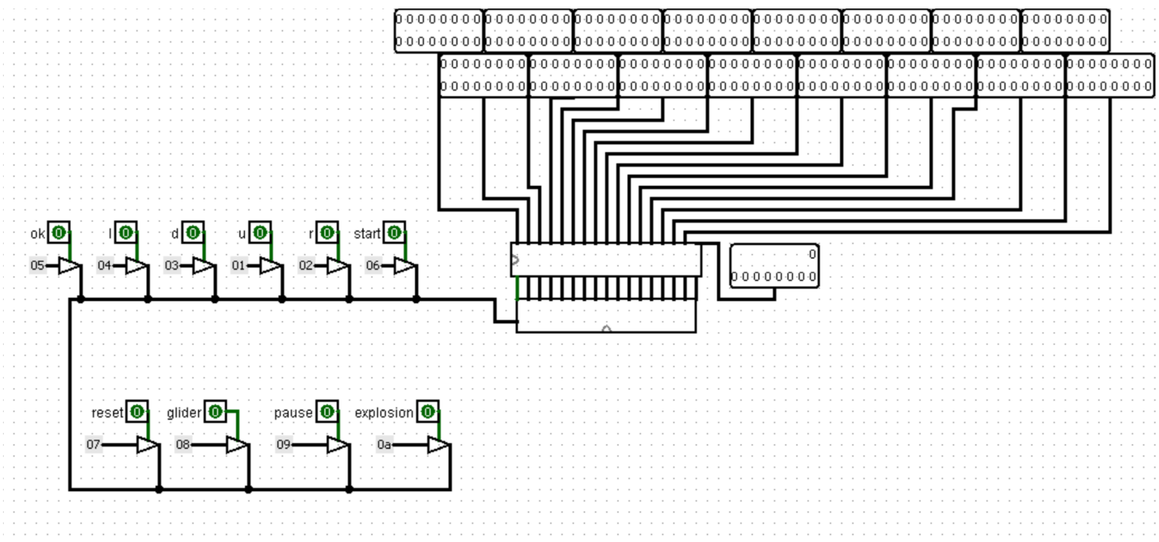
THE MAIN SCHEME



Picture 2.2 – MAIN CIRCUIT

Main scheme in turn contains a display and the ***"BUTTONS"*** subcircuit (it is connected to 9 different buttons). Arrows – to move the cursor cell, ok – to change the state of the cell. Start button starts the game, reset returns us back to the only one cell on the plate – cursor, so it basically does delete all the other cells. Glider is a pattern button which places the glider in the middle of the grid, explosion works the same way – places an explosion on a grid. And the pause button, obviously, stops or continues the game.

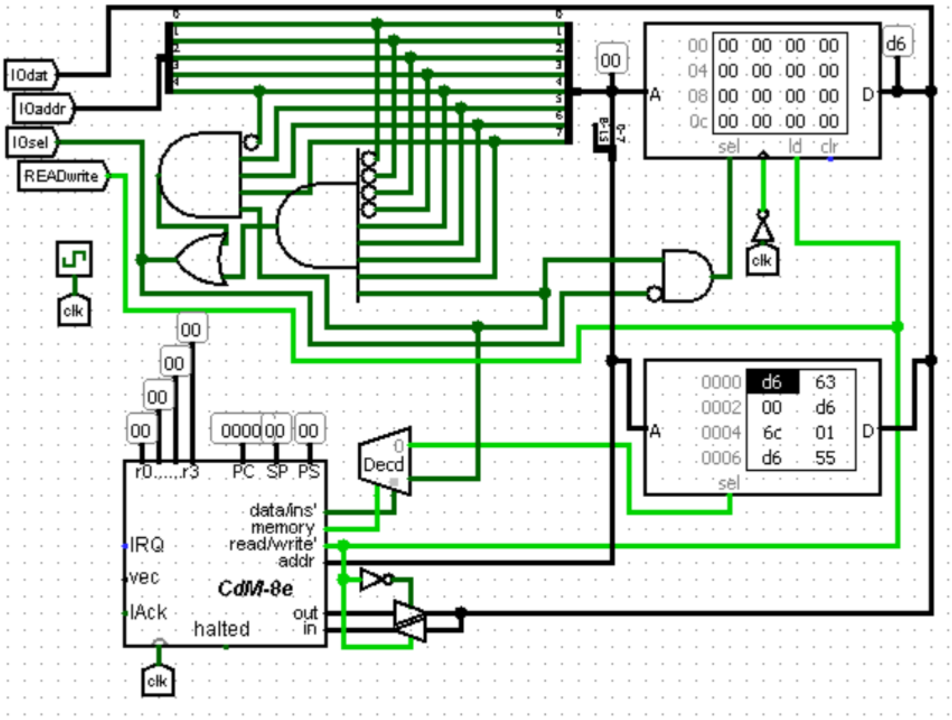
"BUTTONS" itself looks like this:



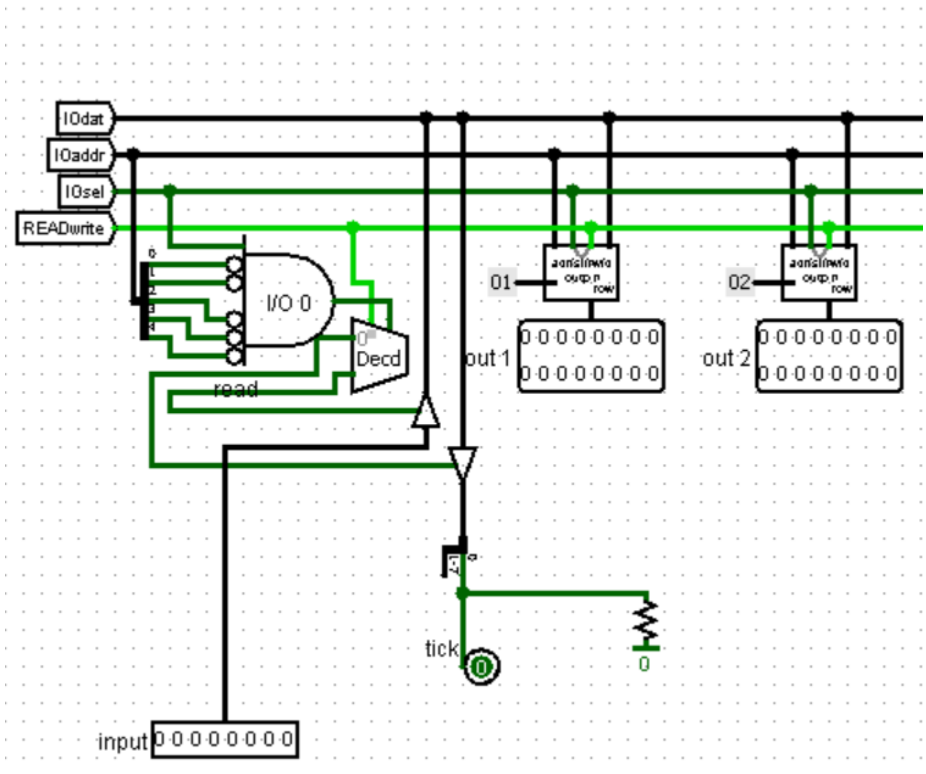
Picture 2.3 – BUTTONS

Each pin on the right side is connected to display – one pin is responsible for one row. Lower right one is for counting alive cells.

This scheme contains 2 more subcircuits – ***"Rows plate"*** and ***"Controller".*** When the player presses some buttons – its code goes into ***"Controller"*** through the input tunnel (IOdat).

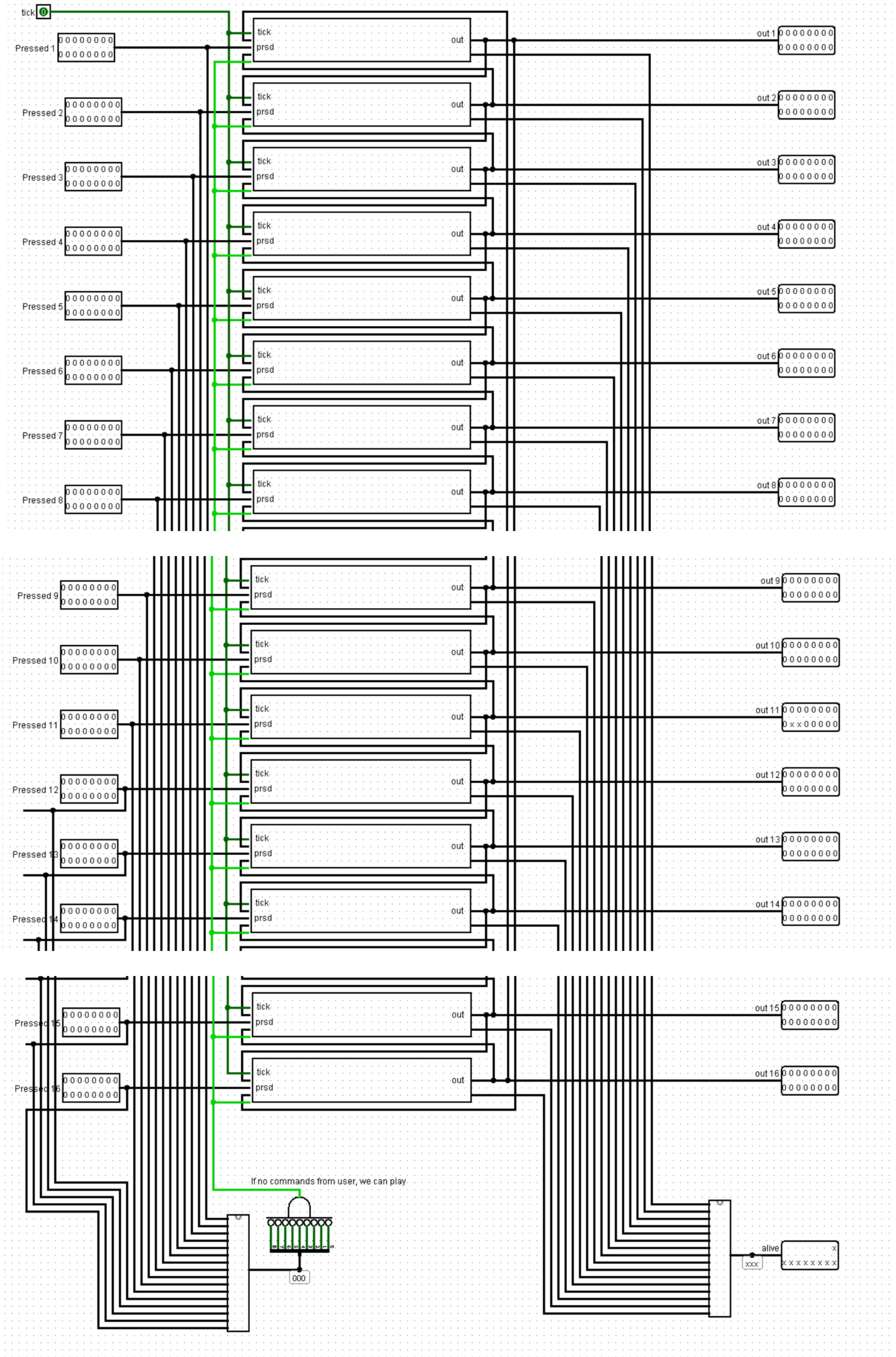


Picture 2.4 – CONTROLLER



Picture 2.5 – CONTROLLER

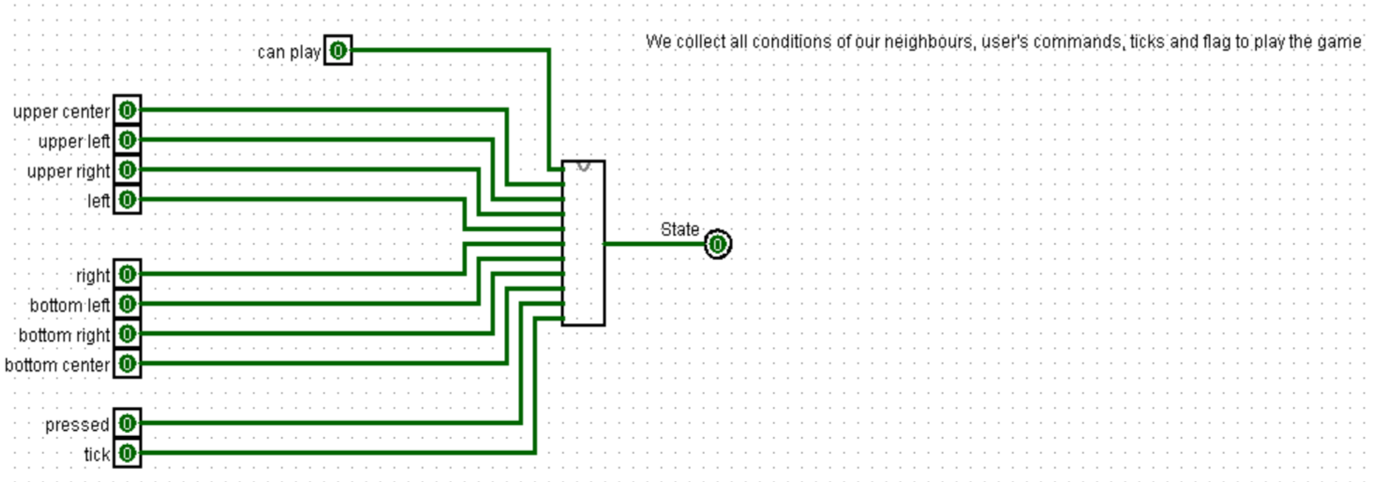
By the end of one tick, we get 16 outputs (IO1 - IO16) which are forming our plate (1 output = 1 row). Each output is going from ***"OutputLogic"*** – that circuit will be considered in "SOFTWARE & HARDWARE"



Picture 2.6 – ROWS PLATE

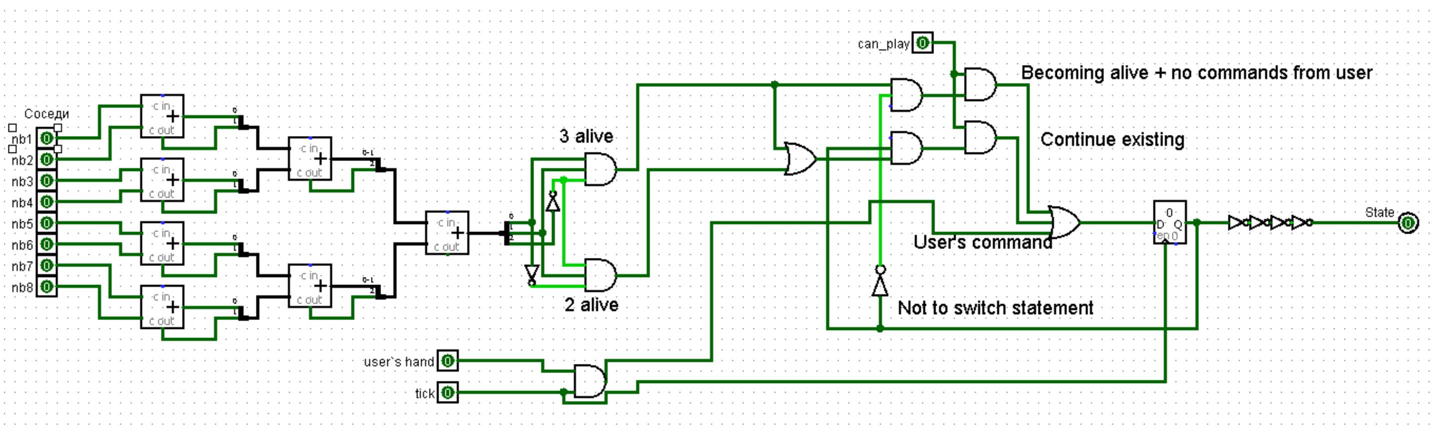
“Rows plate” combines all the rows into the whole grid. It contains 16 ***“Row of cells”*** subcircuits. ***“Row of cell”*** itself is also made of 16 subcircuits called ***“Cell”***. This subcircuit is made of 11 contacts, 8 of which are showing the state of cell’ neighbors. There are also “can play”, “tick” and “pressed” contacts. “Can play” is for controlling placement of already placed cells – they should not change condition when we are putting another cell on a plate - it states 1 only when the user gives us no commands at all (all cells become not pressed when we press "start"). “Pressed” button is responsible for the condition of a cell in the sense of user’s intervention – did the player change the cell's condition or it changed due to the start of the game. “Tick” is tick, when it changes states 2 times, it means one iteration has happened.

In the bottom left we have a subcircuit calculating the number of pressed cells – and if that number is zero (so the user did not change the state of any cells), we set “can play” contact at 1 showing that player is glad with placing.



Picture 2.7 – CELL

All 11 contacts are connected to the ***“State manager”:***



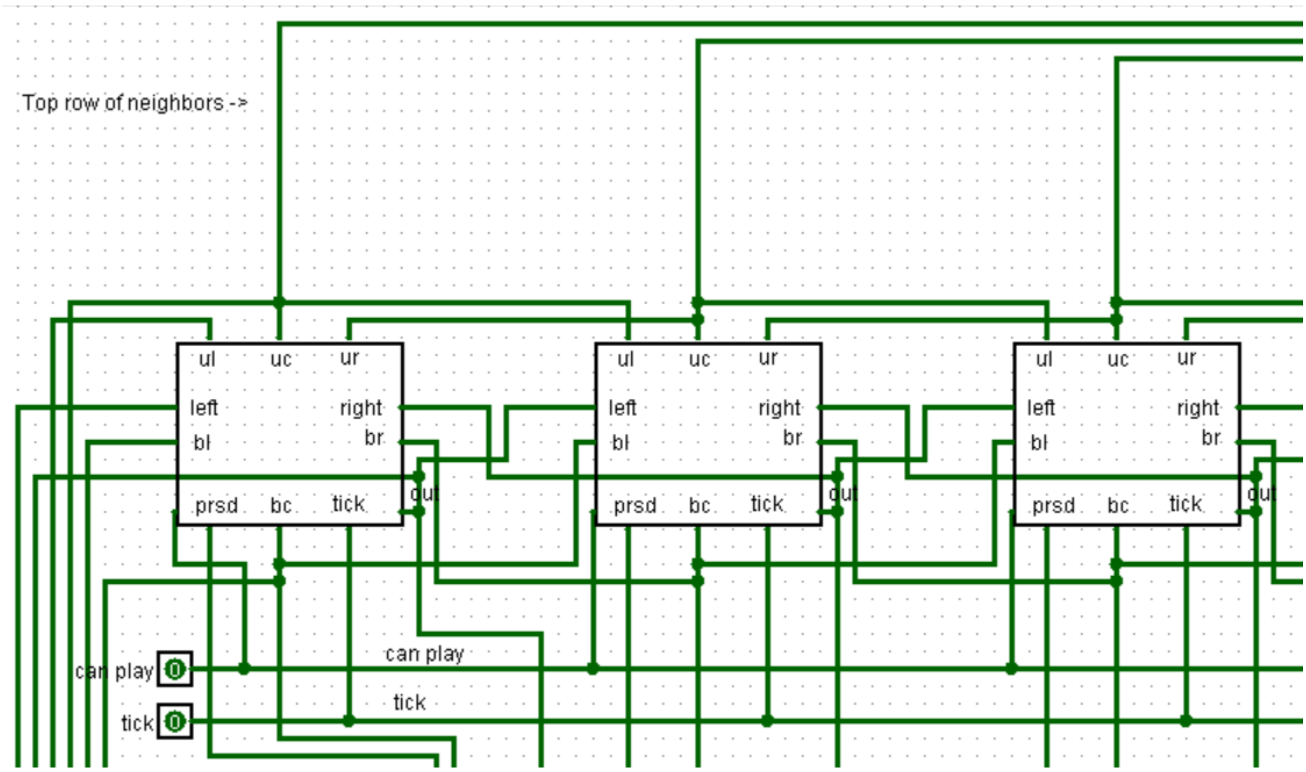
Picture 2.8 – STATE MANAGER

This circuit is calculating the state of the cell according to the rules. Using adders we get the final number of neighbors – if there are 3 or 2 alive cells around – the cell survives, otherwise – dies. If the cell was dead but now it has 3 neighbors – the cell becomes alive.

Turning now back to ***"Row of cells"*** :

We get 2 rows on input – the top row of neighbors and the bottom one. Each of them is sequentially connected to 16 ***"Cell"*** subcircuits.

Here's a piece of ***"Row of cells"*** circuit, for instance

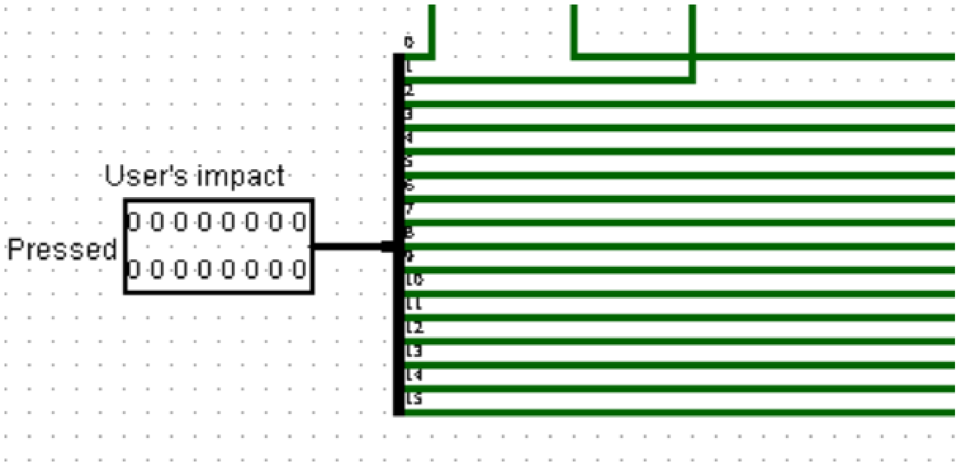


Picture 2.9 – ROW OF CELLS

The second cell is connected to its upper center neighbor, then its upper left neighbor (ul) connects to the first cell's upper center, its upper right connects to the third one upper center, etc. So, it is clear that all cells which are neighbors to each other are connected. Because of that we connect each "Row of cells" subcircuit on the "Rows plate" scheme – any row has upper and down neighbors cells.

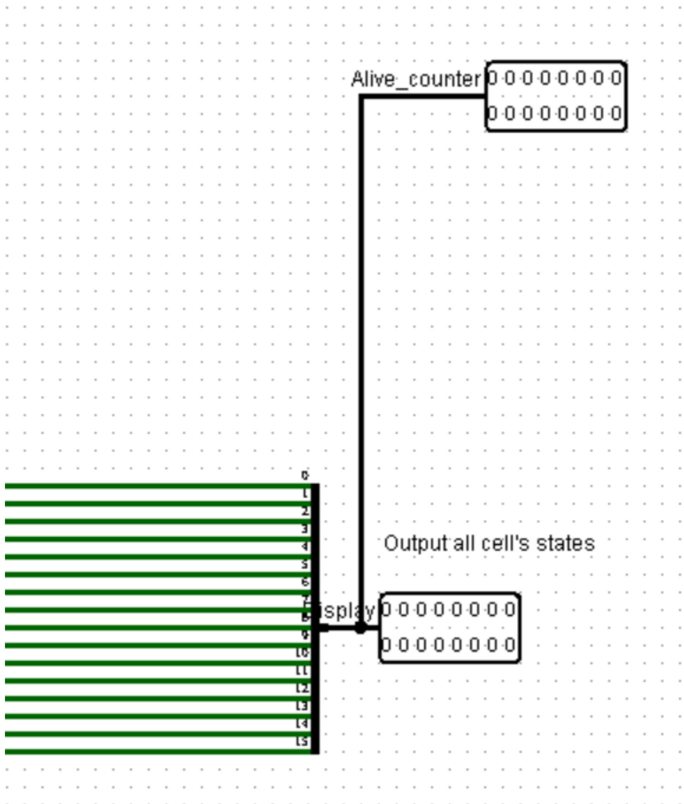
First row upper is the last row, last row down is, obviously, first row.

It also has a "Pressed" pin which is defining user's impact on cells – each of its' 16 wires are connected to cells "pressed" (prsd) contacts



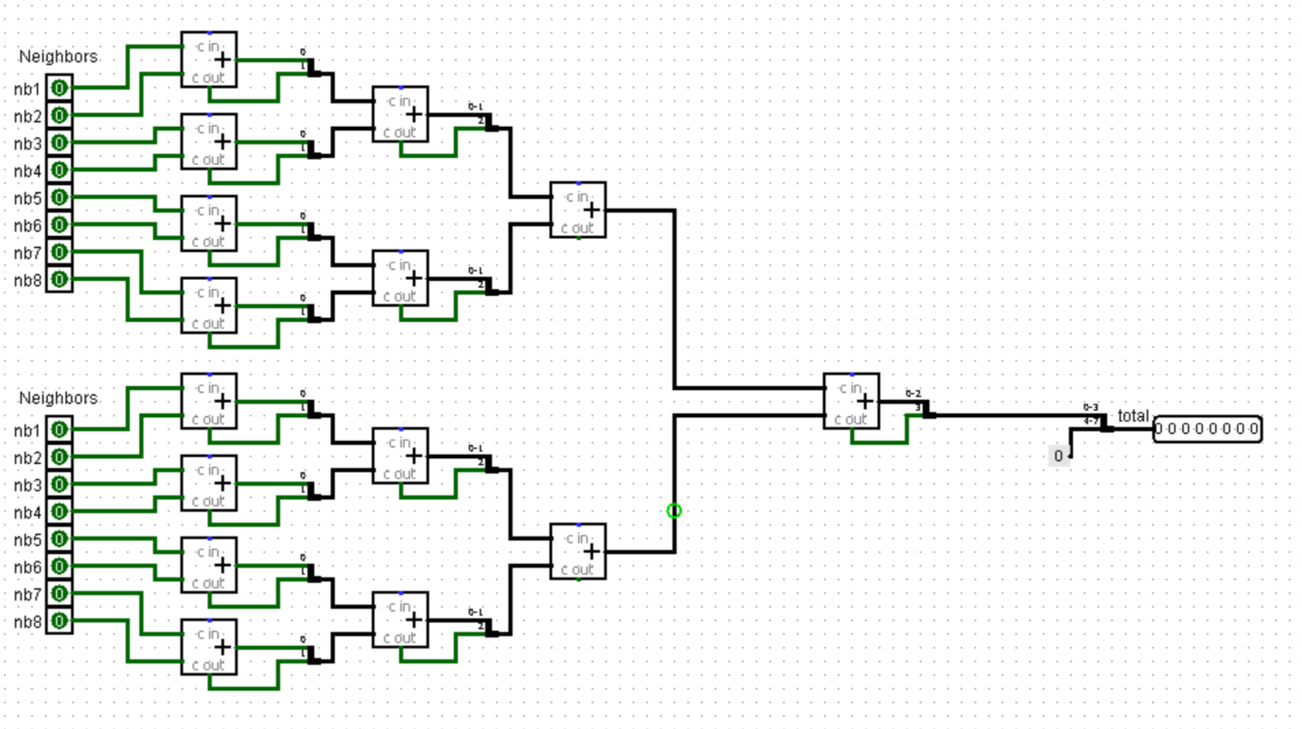
Picture 2.10 – ROW OF CELLS

At the end of the tick all cell outputs (their conditions) are going on display. “Alive_counter” pin also connects to it



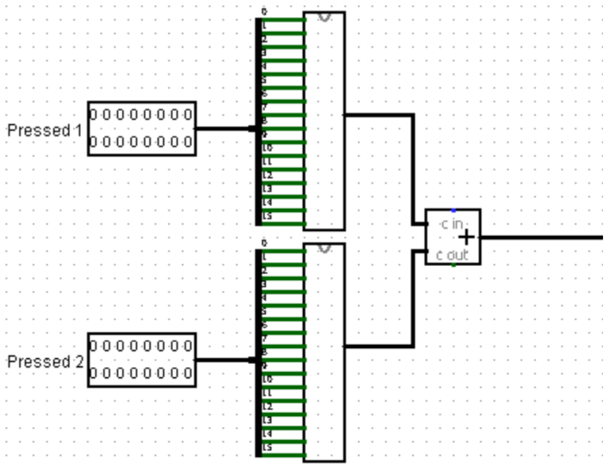
Picture 2.11 – ROW OF CELLS

Each alive_counter connects to ***“PressedCounter”*** on the ***“Rows plate”*** circuit. PressedCounter is made of 16 pins, each bit of whose is, going through splitter, added in ***“counter”***

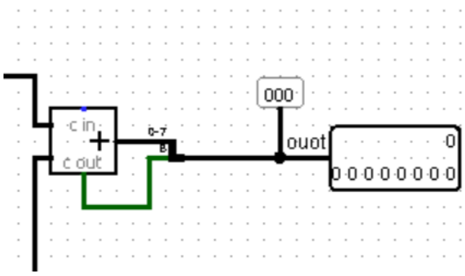


Picture 2.12 – COUNTER

Using adders we calculate the total number of alive cells. In ***"PressedCounter"*** we use 9 bits to determine this number – max number of alive cells is 256 (16x16), so 8 bits are not enough.



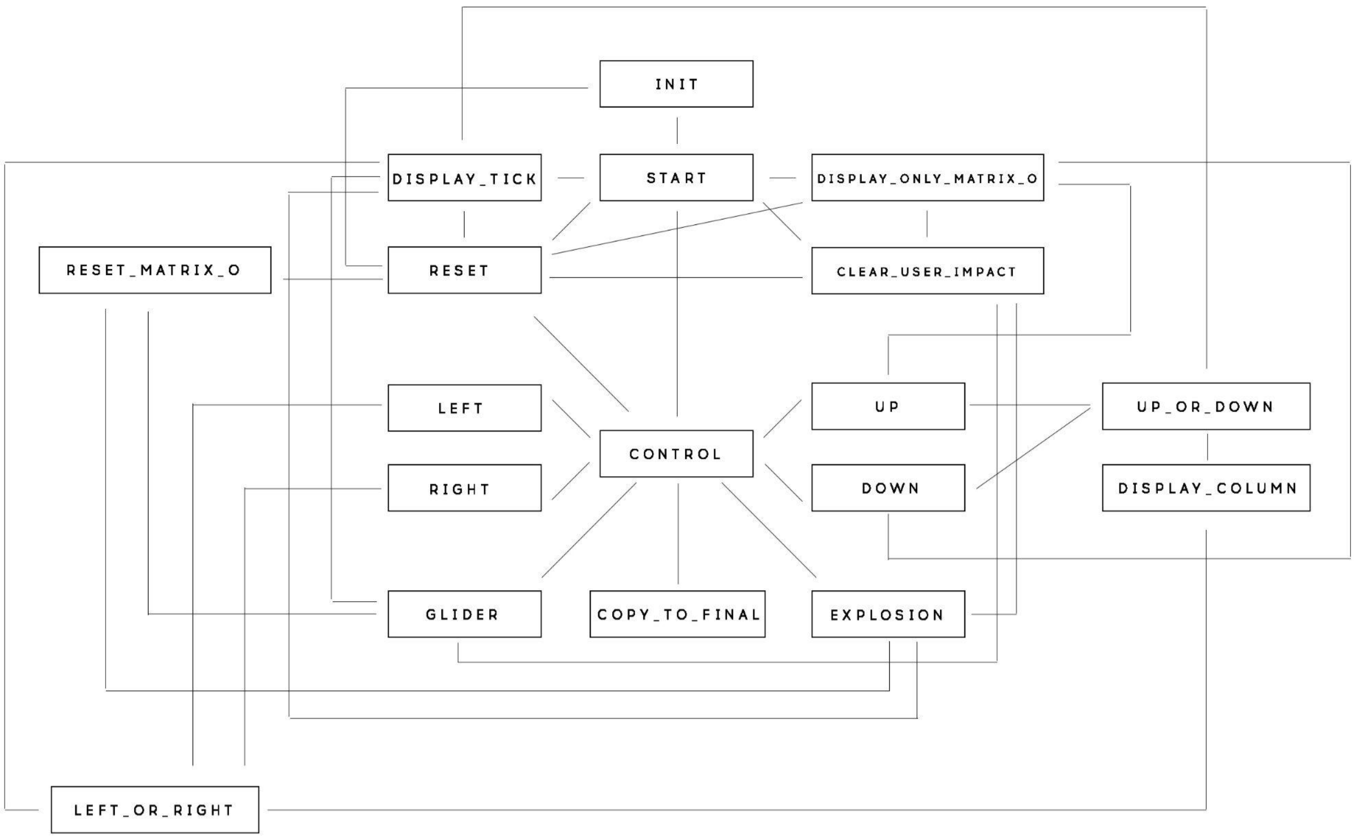
Picture 2.13 – PRESSED_COUNTER



Picture 2.14 – PRESSED_COUNTER

4 SOFTWARE

We have an assembly Controller Program which is basically responsible for all user`s commands. It consists the following subroutines:



Picture 3.1 – SOFTWARE STRUCTURE

- start (actually is main and is never called)
- init
- reset
- clear_user_impact
- reset_matrix_o
- control
- copy_to_final
- display_only_matrix_o
- display_column
- right
- left
- left_or_right

- up
- down
- up_or_down
- display_tick
- glider
- explosion

How they all interact is shown on a block diagram above.

Firstly, we give symbolic names for different addresses:

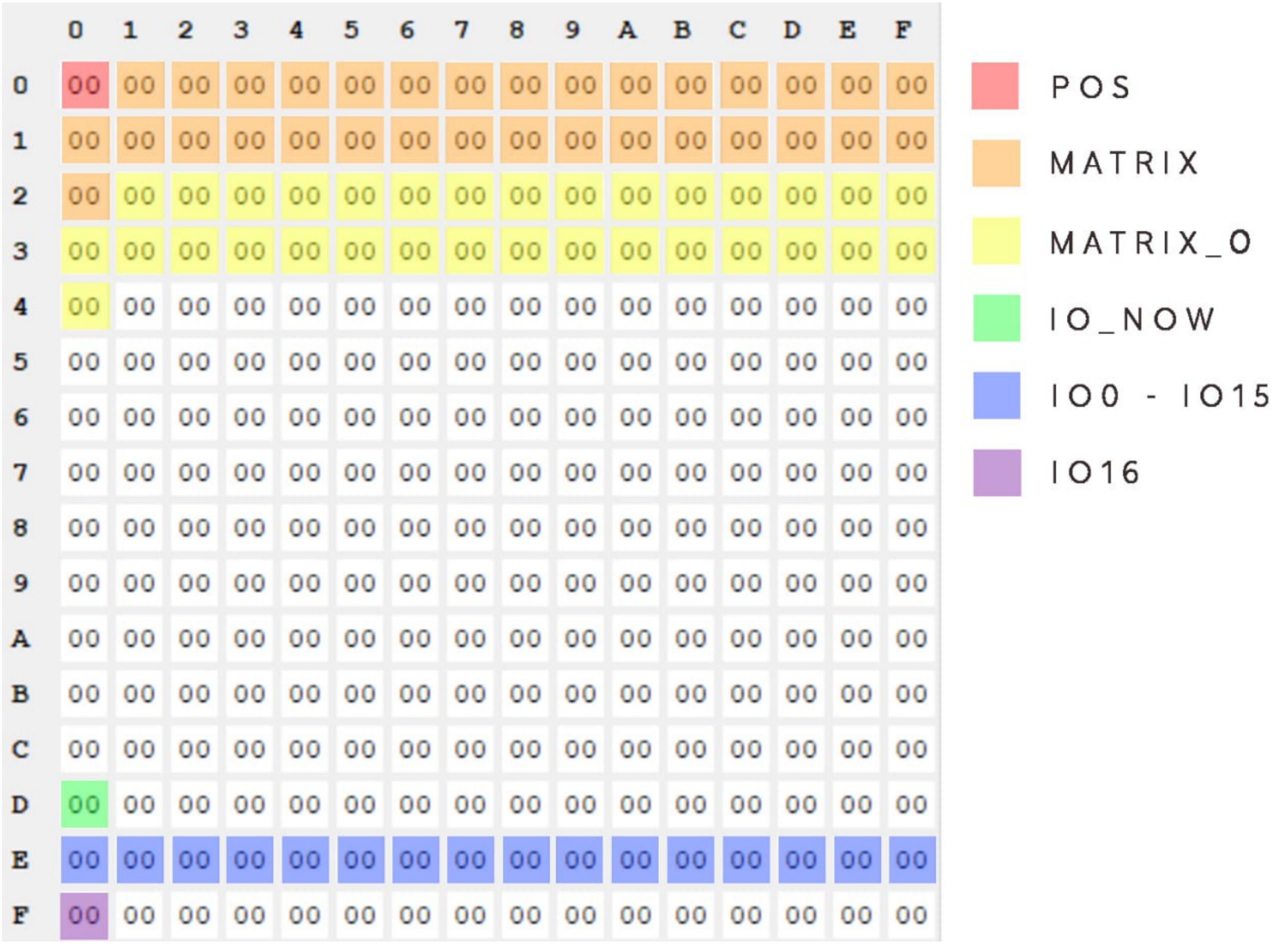
```
# store output to use at the moment
    asect 0xd0
IO_NOW:

# output adresses
    asect 0xe0
IO0:
    asect 0xe1
IO1:
    asect 0xe2
IO2:
    asect 0xe3
IO3:
    asect 0xe4
IO4:
    asect 0xe5
IO5:
    asect 0xe6
IO6:
    asect 0xe7
IO7:
    asect 0xe8
IO8:
    asect 0xe9
IO9:
    asect 0xea
IO10:
    asect 0xeb
IO11:
    asect 0xec
IO12:
    asect 0xed
IO13:
    asect 0xee
IO14:
    asect 0xef
IO15:
    asect 0xf0
IO16:
```

```
# current POS in MATRIX and MATRIX_O
    asect 0x00
POS:
    asect 0x01
MATRIX:
    asect 0x21
MATRIX_O:
```

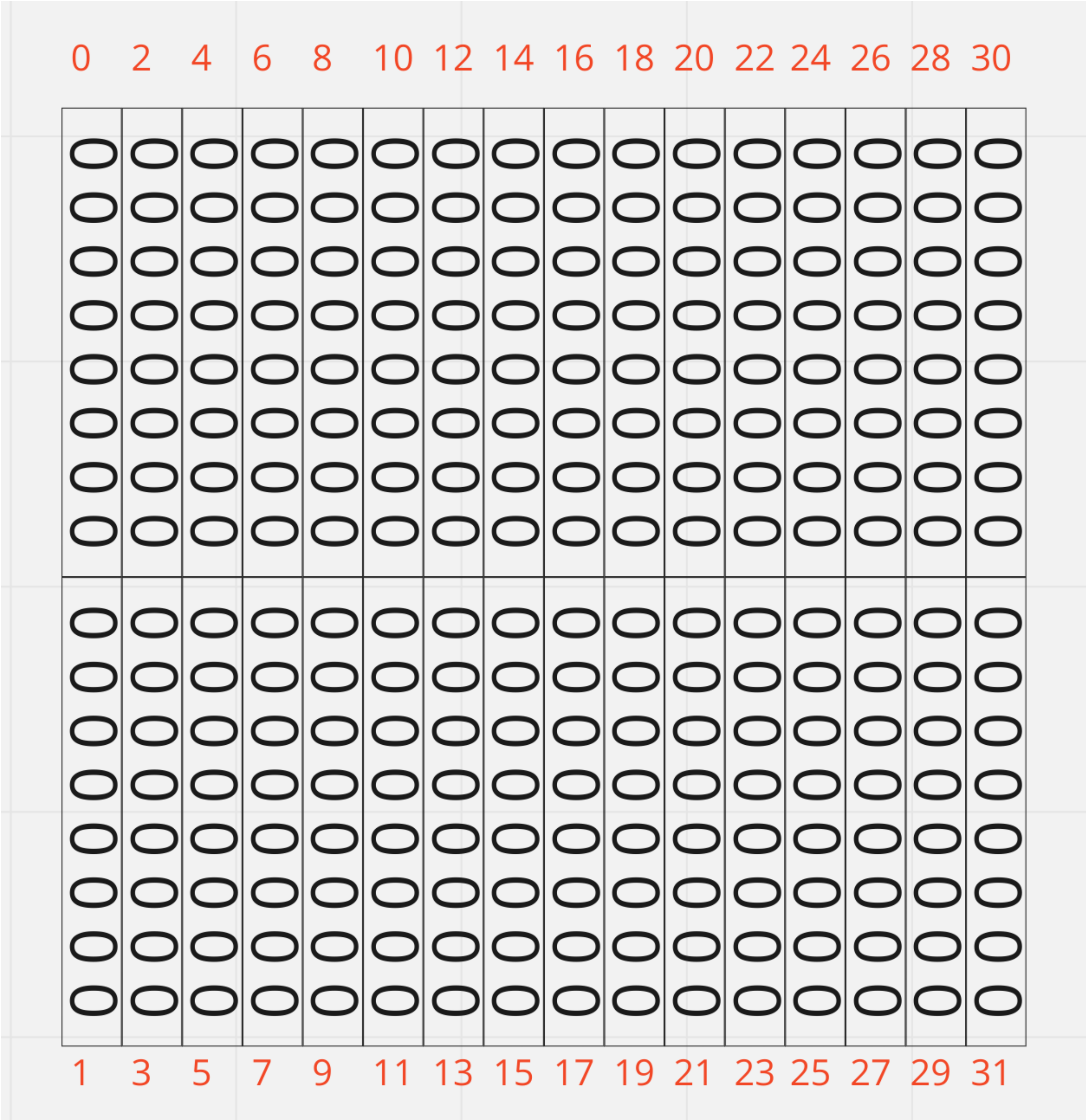
Picture 3.2 – OUTPUT TO USE

This is how all the labels are distributed in memory:



Picture 3.3 – MEMORY ALLOCATION

MATRIX and MATRIX_O are arrays of 32 halves of columns (the whole column is 16 cells, 16 bits, but we only can handle 8 bits in one register, so we split 16 bits into 8 and 8 bits and store these halves).



Picture 3.4 – MATRICES VISUALIZATION

We use MATRIX only to determine the position of the cursor and to change its position by shifts in a row (shla and shr) or by copying the whole byte (half of full column, 8 cells) to another position in MATRIX. MATRIX_O is used for output – it contains cells that will go on display.

Then, after starting point – asect 0, we start our program:

```
asect 0
start:
    # Center cell, POS = 14
    jsr init
    # display first alive cell
    jsr display_only_matrix_o
    jsr display_tick

    # let user control the game while he's not tired
    jsr control

    # now clear user's impact
    jsr clear_user_impact
```

Picture 3.5 – START OF THE PROGRAMM

There're a few subroutines from the list above – **init** (initialisation), **display_only_matrix_o**, **display_tick**, **control** and **clear_user_impact**.

```
# initialize POS, cursor in MATRIX and one cell alive
# in MATRIX_O
init:
    ldi r0, 14  # we place first cell in matrix[14]
    ldi r1, 1
    ldi r2, MATRIX_O
    ldi r3, POS
    st r3, r0
    add r0, r2
    st r2, r1
    ldi r2, MATRIX
    add r0, r2
    st r2, r1

    # set current output adress
    ldi r3, IO8
    ldi r2, IO_NOW
    st r2, r3
    rts
```

Picture 3.6 – INIT

We place the cell in the 14th half of column (first half of 8 column)– almost in the middle of MATRIX. Now we have 1 cell alive. Also, we set IO_NOW to IO8, where the cursor is, to display movements of the cursor faster.

```
# if we need to remove cursor from current column
# we need only to display matrix_o to current column IO
display_only_matrix_o:
    ldi r1, POS
    ld r1, r1

    # make r1 even
    ldi r2, 254
    and r2, r1

    #first half
    ldi r0, MATRIX_O
    add r1, r0
    ld r0, r0
    ldi r2, IO_NOW
    ld r2, r2
    st r2, r0

    #second half
    inc r1
    ldi r0, MATRIX_O
    add r1, r0
    ld r0, r0
    ldi r2, IO_NOW
    ld r2, r2
    st r2, r0

    rts
```

Picture 3.7 – DISPLAY_ONLY_MATRIX_O

Display_only_matrix_o subroutine helps us save the row (column in our case - they are rotated) cells while removing the cursor.

```
display_tick:
    #display tick
    ldi r2, IO0
    ldi r3, 1
    st r2, r3
    dec r3
    st r2, r3
    rts
```

Picture 3.8 – DISPLAY_TICK

The subroutine above is just sequentially putting 1 and 0 in the IO0 address - so 1 tick is done and we update the screen.

Control subroutine processes the button code and according to that code starts other subroutines.

```
# buttons processing
control:
    while
        ldi r3, IO0
        ld r3, r3
        tst r3
    stays nz
        ldi r0,POS
        ld r0,r1 #POS in r1
        ldi r2, MATRIX #MATRIX adress in r2
        add r1, r2 # MATRIX adr + POS to r2

        # checking button codes
        if
            dec r3
        is eq  # 1
            jsr up
        fi
        if
            dec r3
        is eq  # 2
            jsr right
        fi
```

Picture 3.9 – CONTROL

Other calls are the same - we decrease the value by 1 and compare it to 0, if it is equal - we call the relevant subroutine. Only if code is 6 we do nothing but quitting the subroutine using rts.

```
if
    dec r3
is eq  # 6
    #start playing
    rts
```

Picture 3.10 – GAME START

For called by control subroutines we'll be using another one:

```
# display whole column (two halfs)
display_column:
    ldi r1, POS
    ld r1, r1

    ldi r2, 254
    and r2, r1

    #first half
    ldi r2, MATRIX
    add r1, r2
    ld r2, r2
    ldi r0, MATRIX_O
    add r1, r0
    ld r0, r0
    or r2, r0
    ldi r2, IO_NOW
    ld r2, r2
    st r2, r0

    #second half
    inc r1

    ldi r2, MATRIX
    add r1, r2
    ld r2, r2
    ldi r0, MATRIX_O
    add r1, r0
    ld r0, r0
    or r2, r0
    ldi r2, IO_NOW
    ld r2, r2
    st r2, r0
    rts
```

Picture 3.11 – DISPLAY_COLUMN

The subroutine above just refreshes a single row (2 halves of it) considering the cursor position.

```
# set all output to 0
clear_user_impact:
    jsr display_only_matrix_o
    jsr display_tick

    ldi r3, 0
    ldi r0, IO1
    ldi r2, 16
    while
        tst r2
    stays pl
        st r0, r3
        st r0, r3
        inc r0
        dec r2
    wend
    rts
```

Picture 3.12 – CLEAR_USER_IMPACT

Here we're showing the display state using the first subroutine. Then we clear all the rows – set them to 0.

Again, basically our rows are columns due to their connection, so we renamed our buttons - right is actually up, up - left, down - right and left - down. It is shown on Picture 3.4.

We just changed pin labels in BUTTONS to make control understandable.

Subroutine below - "right" - move cursor one position right using bitwise shift considering possible overflow - then we should move to another byte in MATRIX, so we call a subprogram that will be explained later.

```
# right button
right:
    if
        ld r2,r3
        shr r3 # right shift
    is cs #if problems and we crossed the border
        ldi r3, 128
        jsr left_or_right
    else
        ldi r0, 128
        xor r0, r3
        st r2, r3
        jsr display_column
        jsr display_tick
    fi
    rts
```

Picture 3.13 – RIGHT

With "left" button we do almost the same thing and if we cross the border between halves, we call the same subprogram - left_or_right:

```
# left button
left:
    if
        ld r2,r3
        shla r3 # left shift
    is cs
        ldi r3, 1
        jsr left_or_right
    else
        st r2, r3
        jsr display_column
        jsr display_tick
    fi
    rts
```

Picture 3.14 – LEFT

The difference between left and right is in the shift we use: shla and shr, and the value we put in r3: 128 and 1. If we have moved right and the overflow happened - which means we should go to another byte - we place 128 in r3. That's because in the second byte that 1 will be in the 7th bit. With the left case and 1 there's the same thing - we need to move to the next byte - in that byte our 1 will be in the 0 bit.

Also, in the "right" subroutine we xor r0 and our shifted r3. That was made to avoid possible issues with the 7th bit (sometimes it has a value of 1, sometimes of 0).

```
# to make code not so big, merge left and right
# we set r3 to choose where to move
left_or_right:
    ldi r1, 0
    st r2, r1

    if
        ldi r0, POS
        ld r0, r0
        ldi r1, 1
        and r0, r1 #check even POS or not
        tst r1
    is z # even
        inc r2
        inc r0
    else # odd
        dec r2
        dec r0
    fi
    ldi r1, POS
    st r1, r0

    ldi r2, MATRIX
    add r0, r2
    st r2, r3 # now 1 on new POS (-2 or +2)

    jsr display_column
    jsr display_tick
    rts
```

Picture 3.15 – LEFT_OR_RIGHT

In left and right subroutines we store a value into r3, which helps us determine what overflow happened - where we should move. At first we check even POS or odd to understand should we increase or decrease POS (at what half of the column we'll move). Then we store in the MATRIX value of r3 - in the certain column determined by POS. Earlier, in “left” or “right” subroutines, r3 was loaded.

Here’s the “up” subroutine:

```
# up button
up:
    jsr display_only_matrix_o  # remove cursor from old POS
    if
        ldi r3, 1
        ldi r0, POS
        ld r0, r1 # POS in r1
        cmp r1, r3
    is gt
        # regular
        ldi r0, IO_NOW
        ld r0, r1
        dec r1
        ldi r3, -2
    else
        ldi r0, IO_NOW
        ldi r1, IO16
        ldi r3, 30
    fi
    st r0, r1
    jsr up_or_down
    rts
```

Picture 3.16 – UP

Firstly, we need to remove the cursor to another column saving all its cells. For that we use display_only_matrix_o. Then we compare POS with 1 and if it is greater - so we need to move in the left column (in our case) - we put (-2) into r3. That means we should “go up” in 2 bytes. And if it’s not greater than 1 - that means it’s the first column (0 and 1 bytes) - due to the fact we make a toroidal grid - we should go to the 16th column (bytes 30 and 31). That’s why in the second case we store 30 in r3 (we need to go up 30 bytes).

```
# down button
down:
    jsr display_only_matrix_o  # remove cursor from old POS
    if
        ldi r3, 30
        ldi r0, POS
        ld r0, r1 # POS in r1
        cmp r1, r3
    is ge
        # bottom row
        ldi r0, IO_NOW
        ldi r1, IO1
        ldi r3, -30
    else
        # regular
        ldi r0, IO_NOW
        ld r0, r1
        inc r1
        ldi r3, 2
    fi
    st r0, r1
    jsr up_or_down
    rts
```

Picture 3.17 – DOWN

Down button does the opposite thing - if POS is the 16th column and we want to go down (in the right column) we're supposed to be in the 1th column (bytes 0 and 1) - so we put (-30) in r3 . And in all other ways we just go down for a single column i.e. for 2 bytes.

In the end of up of down subroutines we call up_or_down:

```
# same, like with right/left, but there r3 help us to move cursor to MATRIX[POS+r3]
up_or_down:
    ldi r0,POS
    ld r0,r1 #POS in r1
    ldi r2, MATRIX #MATRIX adress in r2
    add r1, r2 # MATRIX adr + POS to r2
    ld r2, r0
    ldi r1, 0
    st r2, r1 #overwrite MATRIX[POS] with 0
    add r3, r2
    st r2, r0 #MATRIX[POS] data saved in r0 to MATRIX[POS+SHIFT]
    ldi r0, POS
    ld r0, r2
    add r3, r2
    st r0, r2 #overwrite POS

    jsr display_column
    jsr display_tick
    rts
```

Picture 3.18 – UP_OR_ DOWN

Here we just shift MATRIX[POS] by adding the value in r3 (it was loaded in "up" or in "down"). Then overwrite MATRIX[POS] - because the cursor has moved. And lastly remember the final byte that POS points on.

When user is done with placing cells, the life itself starts:

```
# and now infinite loop of life
while
     ldi r3, IO0
     ld r3, r3
     tst r3
stays nz
     if
          ldi r1, 9
          cmp r1, r3
     is eq
          # waiting for pause button to be pressed again
          while
               ldi r3, IO0
               ld r3, r3
               ldi r1, 9
               cmp r1, r3
          stays ne
               # here is possible to reset the game
               if
                    ldi r1, 7
                    cmp r1, r3
               is eq
                    jsr reset
                    jsr control
                    jsr clear_user_impact
                    ldi r3, 9
               fi
          wend
     else
          # reset while game is in process
          if
               ldi r1, 7
               cmp r1, r3
          is eq
               jsr reset
               jsr control
               jsr clear_user_impact
          else
               ldi r0, IO0
               ldi r1, 1
               ldi r2, 0
               st r0, r1
               st r0, r2
          fi
     fi
wend
```

Picture 3.19 – MAIN

Code 9 is responsible for pause, so we stop the game until the user presses that button again. Then we check if the buttons' code is 7 – it's responsible for reset. In other cases, we put 1 and 0 in the IO0 address to refresh the display.

```
# reset button
reset:
    jsr reset_matrix_o
    jsr clear_user_impact

    jsr init
    jsr display_only_matrix_o
    jsr display_tick
    rts
```

Picture 3.20 – RESET

```
# nothing special, just clear the matrix_o
reset_matrix_o:
    ldi r0, MATRIX_O
    ldi r2, 32
    ldi r3, 0
    while
        tst r2
    stays pl
        st r0, r3
        inc r0
        dec r2
    wend
    ldi r0, MATRIX
    ldi r1, POS
    ld r1, r1
    add r1, r0
    st r0, r3
    rts
```

Picture 3.21 – RESET_MATRIX_O

Reset clears MATRIX_O, then all the rows. After it calls an initialization and places the cursor on the display (last 2 subroutines).

```
glider:
    jsr reset_matrix_o
    jsr clear_user_impact

    jsr init # cursor`ll be in center
    ldi r3, IO7
    ldi r0, 12
    ldi r1, MATRIX_O
    add r0, r1
    ldi r2, 3
    st r1, r2
    st r3, r2
    inc r1
    ldi r2, 0   # second half
    st r3, r2
    inc r1

    # next IO
    inc r3
    ldi r2, 5
    st r1, r2
    st r3, r2
    inc r1
    ldi r2, 0   # second half
    st r3, r2

    #next IO
    inc r3
    inc r1
    ldi r2, 1
    st r1, r2
    st r3, r2
    dec r2
    st r3, r2   # 0 to the second half

    jsr display_tick
    rts
```

Picture 3.22 – GLIDER

That is just a hard-coded pattern of a glider - we put certain cells on a plate to make a diagonally moving game figure.

```
explosion:
    jsr reset_matrix_o
    jsr clear_user_impact

    jsr init # cursor`ll be in center
    ldi r3, IO6
    ldi r0, 10
    ldi r1, MATRIX_O
    add r0, r1
    ldi r2, 3
    st r1, r2
    st r3, r2
    inc r1
    ldi r0, 128  # second half
    st r3, r0
    st r1, r0
    inc r1

    # next IO
    inc r3
    ldi r2, 2
    st r1, r2
    st r3, r2
    inc r1
    st r3, r0
    st r1, r0
    inc r1

    # next IO
    inc r3
    ldi r2, 3
    st r1, r2
    st r3, r2
    inc r1
    st r3, r0
    st r1, r0
    inc r1

    # shift
    inc r1
    inc r1
    inc r1
    inc r3
    inc r3

    ldi r2, 3
    st r1, r2
    st r3, r2
    inc r1
    st r3, r0
    st r1, r0
    inc r1

    # next IO
    inc r3
    ldi r2, 2
    st r1, r2
    st r3, r2
    inc r1
    st r3, r0
    st r1, r0
    inc r1

    # next IO
    inc r3
    ldi r2, 3
    st r1, r2
    st r3, r2
    inc r1
    st r1, r0
    st r3, r0

    jsr display_tick
    rts
```

Picture 3.23 – EXPLOSION

This subroutine has a similar function - placing the pattern on the display. Now - the explosion pattern.

There's the last of listed subroutines:

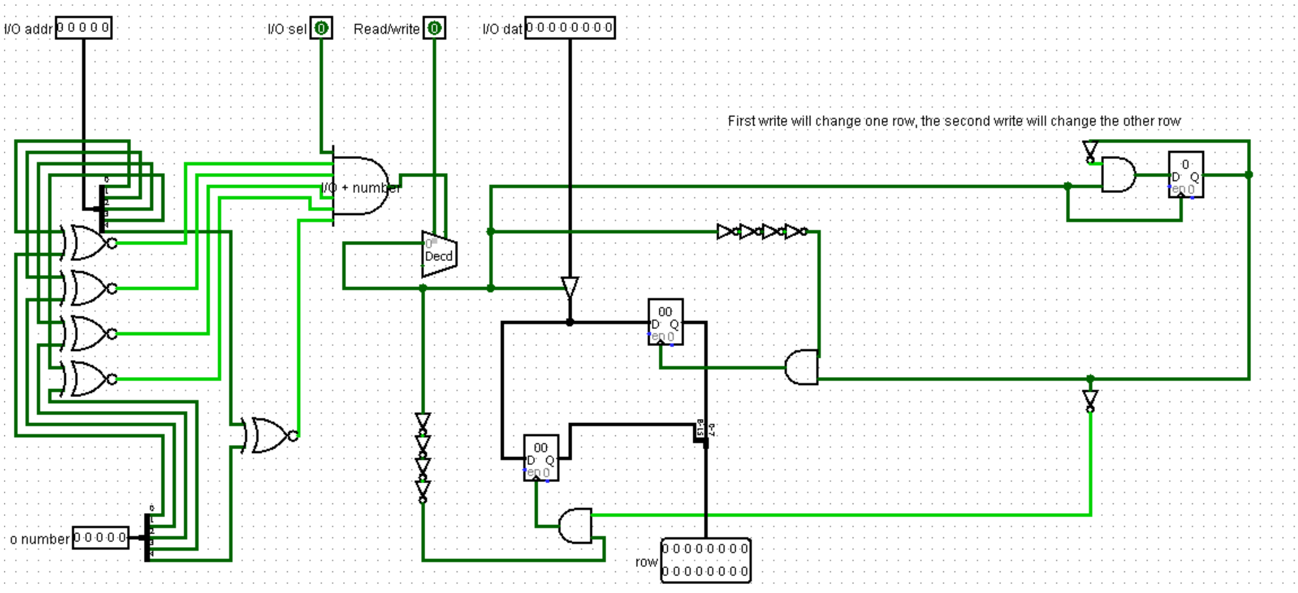
```
# copy-paste MATRIX[POS] to MATRIX_O[POS]
copy_to_final:
    ldi r0, MATRIX
    ldi r1, MATRIX_O
    ldi r2, POS
    ld r2, r2
    add r2, r0
    add r2, r1
    ld r0, r0
    ld r1, r2
    xor r0, r2
    st r1, r2
    rts
```

Picture 3.24 – COPY_TO_FINAL

All it does is copy the cursor from MATRIX to MATRIX_O. In the BUTTONS circuit it's the "OK" button (code 5). That's how we save our placed cells.

5 SOFTWARE & HARDWARE

OutputLogic



Picture 4.1 – RESET_MATRIX_O

This subcircuit forms a 16 bit row with two 8 bit outputs to the same IO that is possible because of the third register (let's call it flag-register because it sets to 1 when we got the first half and to 0 when we got the 2nd half) . First storage command to this IO places content to the first 8 bit of output pin from OutputLogic and changes flag-register to 1. The next storage command places content to the second half because flag-register is 1 and then sets flag-register to 0. After two storage commands we have a full 16 bit row to display and flag-register set to 0.

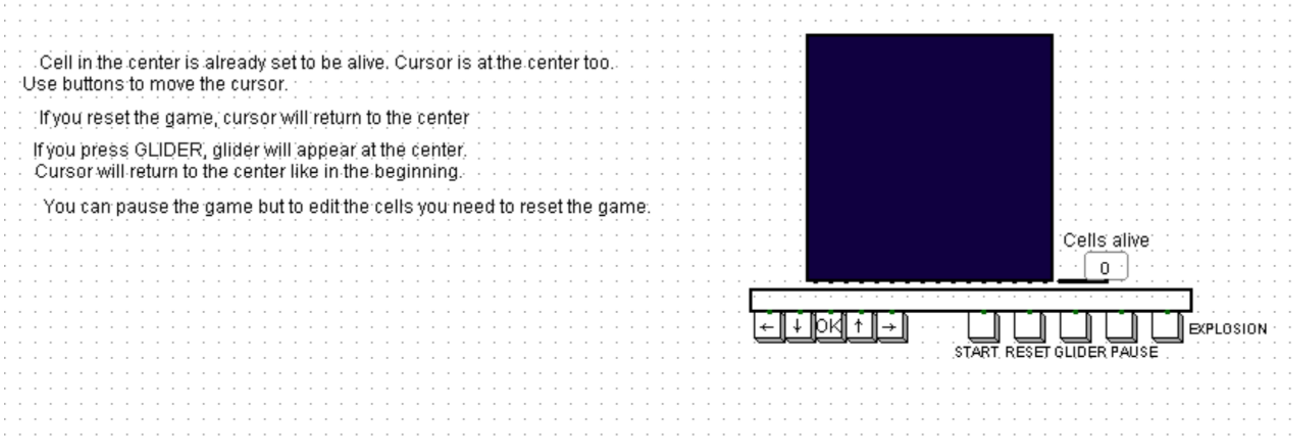
Controller

In the picture 1.3 - "Controller" we have 2 AND gates connected to one OR (upper-left corner), which are used for bit masking. Left one is for masking all the 0xe* addresses (0xe0, 0xe1 and etc.). Right one - only for one address - 0xf0 - in which IO16 is placed.

6 USER MANUAL

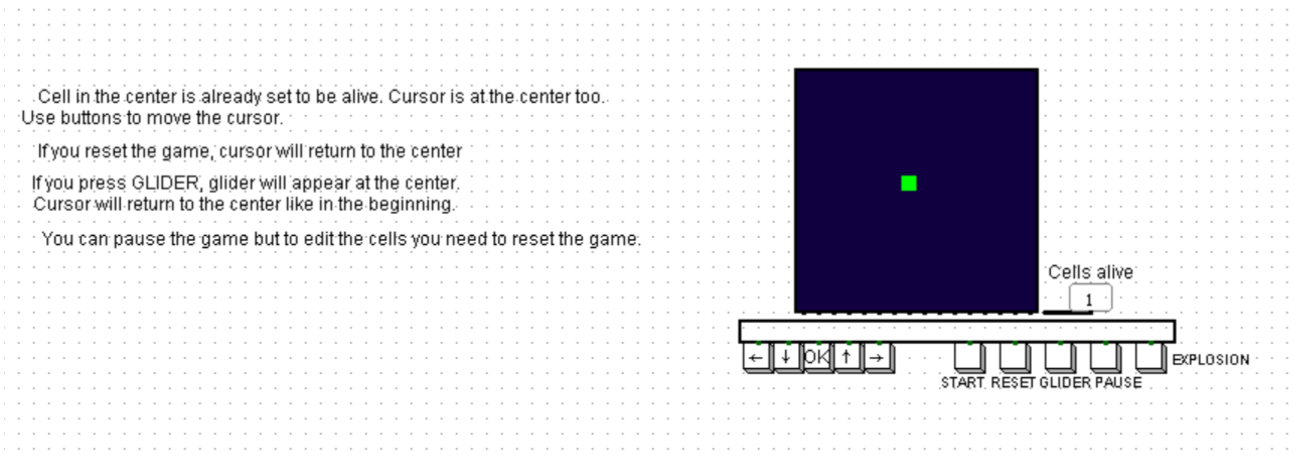
To run the game follow this steps:

1. Install Logisim.
2. Download and extract archive with our project.
3. Open TheLife.circ file in Logisim.
4. Enable the simulation. (Top bar -> Simulate -> Simulation Enabled (Ctrl+E)).
5. Enable ticks (Top bar -> Simulate -> Ticks Enabled (Ctrl+K)).
6. Enjoy!



Picture 5.1 – START SCREEN

At the beginning of simulation, the cursor cell is going to appear:



Picture 5.2 – SIMULATION BEGINS

Using arrows, the player can move it wherever he wants. To fix the position of a new cell or to change the condition of an already placed one – he presses the OK button. "START" button surprisingly starts the game. “RESET” was made for convenience – not to disable simulation every time. “PAUSE” stops or continues the game. “GLIDER” button places pattern of glider to the middle of grid, “EXPLOSION” places the explosion pattern.

CONCLUSION

We have created a functional circuit and Assembly program that are connected and faithfully simulate the behavior of the game with additional features that distinguish our program from the classic version of the game.

Circuit includes 10 subcircuits in total. Program for CdM-8e is written in Assembly and has 572 lines.

APPENDIX 1

```
# store output to use at the moment
        asect 0xd0
IO_NOW:

# output adresses
        asect 0xe0
IO0:
        asect 0xe1
IO1:
        asect 0xe2
IO2:
        asect 0xe3
IO3:
        asect 0xe4
IO4:
        asect 0xe5
IO5:
        asect 0xe6
IO6:
        asect 0xe7
IO7:
        asect 0xe8
IO8:
        asect 0xe9
IO9:
        asect 0xea
IO10:
        asect 0xeb
IO11:
        asect 0xec
IO12:
        asect 0xed
IO13:
        asect 0xee
IO14:
        asect 0xef
IO15:
```

```
	asect 0xf0
IO16:

# current POS in MATRIX and MATRIX_O
	asect 0x00
POS:
	asect 0x01
MATRIX:
	asect 0x21
MATRIX_O:


##########################################################################
### main code
asect 0
start:
	# Center cell, POS = 14
	jsr init
	# display first alive cell
	jsr display_only_matrix_o
	jsr display_tick

	# let user control the game while he's not tired
	jsr control

	# now clear user's impact
	jsr clear_user_impact

	# and now infinite loop of life
	while
		ldi r3, IO0
		ld r3, r3
		tst r3
	stays nz
		if
			ldi r1, 9
			cmp r1, r3
		is eq
			# waiting for pause button to be pressed again
			while
```

```
                        ldi r3, IO0
                        ld r3, r3
                        ldi r1, 9
                        cmp r1, r3
                stays ne
                        # here is possible to reset the game
                        if
                                ldi r1, 7
                                cmp r1, r3
                        is eq
                                jsr reset
                                jsr control
                                jsr clear_user_impact
                                ldi r3, 9
                        fi
                wend
        else
                # reset while game is in process
                if
                        ldi r1, 7
                        cmp r1, r3
                is eq
                        jsr reset
                        jsr control
                        jsr clear_user_impact
                else
                        ldi r0, IO0
                        ldi r1, 1
                        ldi r2, 0
                        st r0, r1
                        st r0, r2
                fi
        fi
    wend

# set all output to 0
clear_user_impact:
    jsr display_only_matrix_o

    ldi r3, 0
```

```
    ldi r0, IO1
    ldi r2, 16
    while
        tst r2
    stays pl
        st r0, r3
        st r0, r3
        inc r0
        dec r2
    wend
    rts

# initialize POS, cursor in MATRIX and one cell alive in MATRIX_O
init:
    ldi r0, 14  # we place first cell in matrix[14]
    ldi r1, 1
    ldi r2, MATRIX_O
    ldi r3, POS
    st r3, r0
    add r0, r2
    st r2, r1
    ldi r2, MATRIX
    add r0, r2
    st r2, r1

    # set current output adress
    ldi r3, IO8
    ldi r2, IO_NOW
    st r2, r3
    rts

# buttons processing
control:
    while
        ldi r3, IO0
        ld r3, r3
        tst r3
    stays nz
        ldi r0,POS
        ld r0,r1 #POS in r1
```

```
ldi r2, MATRIX #MATRIX adress in r2
add r1, r2 # MATRIX adr + POS to r2

# checking button codes
if
        dec r3
is eq  # 1
        jsr up
fi
if
        dec r3
is eq  # 2
        jsr right
fi
if
        dec r3
is eq  # 3
        jsr down
fi
if
        dec r3
is eq  # 4
        jsr left
fi
if
        dec r3
is eq  # 5
        jsr copy_to_final
fi
if
        dec r3
is eq  # 6
        #start playing
        rts
fi
if
        dec r3
is eq  # 7
        jsr reset
fi
```

```
		if
			dec r3
		is eq  # 8
			jsr glider
		fi
		if
			dec r3
			dec r3
		is eq  # 10
			jsr explosion
		fi
	wend
	rts

# nothing special, just clear the matrix_o
reset_matrix_o:
	ldi r0, MATRIX_O
	ldi r2, 32
	ldi r3, 0
	while
		tst r2
	stays pl
		st r0, r3
		inc r0
		dec r2
	wend
	ldi r0, MATRIX
	ldi r1, POS
	ld r1, r1
	add r1, r0
	st r0, r3
	rts

# reset button
reset:
	jsr reset_matrix_o
	jsr clear_user_impact

	jsr init
	jsr display_only_matrix_o
```

```
	jsr display_tick
	rts

# glider button
glider:
	jsr reset_matrix_o
	jsr clear_user_impact

	jsr init # cursor`ll be in center
	ldi r3, IO7
	ldi r0, 12
	ldi r1, MATRIX_O
	add r0, r1
	ldi r2, 3
	st r1, r2
	st r3, r2
	inc r1
	ldi r2, 0  # second half
	st r3, r2
	inc r1

	# next IO
	inc r3
	ldi r2, 5
	st r1, r2
	st r3, r2
	inc r1
	ldi r2, 0  # second half
	st r3, r2

	#next IO
	inc r3
	inc r1
	ldi r2, 1
	st r1, r2
	st r3, r2
	dec r2
	st r3, r2  # 0 to the second half

	jsr display_tick
```

```
        rts

# make pattern of explosion
explosion:
        jsr reset_matrix_o
        jsr clear_user_impact

        jsr init # cursor`ll be in center
        ldi r3, IO6
        ldi r0, 10
        ldi r1, MATRIX_O
        add r0, r1
        ldi r2, 3
        st r1, r2
        st r3, r2
        inc r1
        ldi r0, 128  # second half
        st r3, r0
        st r1, r0
        inc r1

        # next IO
        inc r3
        ldi r2, 2
        st r1, r2
        st r3, r2
        inc r1
        st r3, r0
        st r1, r0
        inc r1

        # next IO
        inc r3
        ldi r2, 3
        st r1, r2
        st r3, r2
        inc r1
        st r3, r0
        st r1, r0
        inc r1
```

```
	# shift
	inc r1
	inc r1
	inc r1
	inc r3
	inc r3

	ldi r2, 3
	st r1, r2
	st r3, r2
	inc r1
	st r3, r0
	st r1, r0
	inc r1

	# next IO
	inc r3
	ldi r2, 2
	st r1, r2
	st r3, r2
	inc r1
	st r3, r0
	st r1, r0
	inc r1

	# next IO
	inc r3
	ldi r2, 3
	st r1, r2
	st r3, r2
	inc r1
	st r1, r0
	st r3, r0

	jsr display_tick
	rts

# copy-paste MATRIX[POS] to MATRIX_O[POS]
copy_to_final:
```

```
        ldi r0, MATRIX
        ldi r1, MATRIX_O
        ldi r2, POS
        ld r2, r2
        add r2, r0
        add r2, r1
        ld r0, r0
        ld r1, r2
        xor r0, r2
        st r1, r2
        rts

# if we need to remove cursor from current column
# we need only to display matrix_o to current column IO
display_only_matrix_o:
        ldi r1, POS
        ld r1, r1

        # make r1 even
        ldi r2, 254
        and r2, r1

        #first half
        ldi r0, MATRIX_O
        add r1, r0
        ld r0, r0
        ldi r2, IO_NOW
        ld r2, r2
        st r2, r0

        #second half
        inc r1
        ldi r0, MATRIX_O
        add r1, r0
        ld r0, r0
        ldi r2, IO_NOW
        ld r2, r2
        st r2, r0

        rts
```

```
# display whole column (two halfs)
display_column:
        ldi r1, POS
        ld r1, r1

        ldi r2, 254
        and r2, r1

        #first half
        ldi r2, MATRIX
        add r1, r2
        ld r2, r2
        ldi r0, MATRIX_O
        add r1, r0
        ld r0, r0
        or r2, r0
        ldi r2, IO_NOW
        ld r2, r2
        st r2, r0

        #second half
        inc r1

        ldi r2, MATRIX
        add r1, r2
        ld r2, r2
        ldi r0, MATRIX_O
        add r1, r0
        ld r0, r0
        or r2, r0
        ldi r2, IO_NOW
        ld r2, r2
        st r2, r0
        rts

# right button
right:
        if
                ld r2,r3
```

```
        shr r3 # right shift
    is cs #if problems and we crossed the border
        ldi r3, 128
        jsr left_or_right
    else
        ldi r0, 128
        xor r0, r3
        st r2, r3
        jsr display_column
        jsr display_tick
    fi
    rts

# left button
left:
    if
        ld r2,r3
        shla r3 # left shift
    is cs
        ldi r3, 1
        jsr left_or_right
    else
        st r2, r3
        jsr display_column
        jsr display_tick
    fi
    rts

# to make code not so big, merge left and right
# we set r3 to choose where to move
left_or_right:
    ldi r1, 0
    st r2, r1

    if
        ldi r0, POS
        ld r0, r0
        ldi r1, 1
        and r0, r1 #check even POS or not
        tst r1
```

```
	is z # even
		inc r2
		inc r0
	else # odd
		dec r2
		dec r0
	fi
	ldi r1, POS
	st r1, r0

	ldi r2, MATRIX
	add r0, r2
	st r2, r3 # now 1 on new POS (-2 or +2)

	jsr display_column
	jsr display_tick
	rts

# up button
up:
	jsr display_only_matrix_o  # remove cursor from old POS
	if
		ldi r3, 1
		ldi r0, POS
		ld r0, r1 # POS in r1
		cmp r1, r3
	is gt
		# regular
		ldi r0, IO_NOW
		ld r0, r1
		dec r1
		ldi r3, -2
	else
		ldi r0, IO_NOW
		ldi r1, IO16
		ldi r3, 30
	fi
	st r0, r1
	jsr up_or_down
	rts
```

```
# down button
down:
	jsr display_only_matrix_o  # remove cursor from old POS
	if
		ldi r3, 30
		ldi r0, POS
		ld r0, r1 # POS in r1
		cmp r1, r3
	is ge
		# bottom row
		ldi r0, IO_NOW
		ldi r1, IO1
		ldi r3, -30
	else
		# regular
		ldi r0, IO_NOW
		ld r0, r1
		inc r1
		ldi r3, 2
	fi
	st r0, r1
	jsr up_or_down
	rts

# same, like with right/left, but there r3 help us to move cursor to MATRIX[POS+r3]
up_or_down:
	ldi r0,POS
	ld r0,r1 #POS in r1
	ldi r2, MATRIX #MATRIX adress in r2
	add r1, r2 # MATRIX adr + POS to r2
	ld r2, r0
	ldi r1, 0
	st r2, r1 #overwrite MATRIX[POS] with 0
	add r3, r2
	st r2, r0 #MATRIX[POS] data saved in r0 to MATRIX[POS+SHIFT]
	ldi r0, POS
	ld r0, r2
	add r3, r2
	st r0, r2 #overwrite POS
```

```
        jsr display_column
        jsr display_tick
        rts

display_tick:
        #display tick
        ldi r2, IO0
        ldi r3, 1
        st r2, r3
        dec r3
        st r2, r3
        rts

end
```