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)

15.03.06 - Mechatronics and Robotics
Focus (profile): Artificial intelligence

TERM PAPER

Nazarov Danila Alekseevich
Kalugina Anastasia Vladimirovna

The theme of the paper:

“The game of Life”

Novosibirsk, 2023

Contents

Novosibirsk, 2023

Introduction

Games are quite in demand in the 21st century, and more and more people are immersed in them. But very few people think about how they work from the inside. And in order to create a game, you need to understand it. Your own implementation of the game will help to understand this.

We chose “The game of life” as the theme of the project. In previous years there was no such theme, so we became interested in implementing this game. In addition, the documentation for the project is given in Russian).

Rules:

- Any “living” cell with less than two “living” neighbors dies, as if caused by an insufficient population;
- Any “living” cell with two or three “living” neighbors lives on for the next generation;
- Any “living” cell with more than three “living” neighbors is dying, as if due to overpopulation;
- Any “dead” cell that has exactly three “living” neighbors becomes a “living” cell by reproduction.

The games stops if:

- There will not be a single “living” cell left on the field;
- The configuration in the next step will repeat itself exactly (without shifts and rotations) in one of the earlier steps (a periodic configuration is formed);
- At the next step none of the cells changes its state (a special case of the previous rule, a stable configuration is formed).

The player does not actively participate in the game. He only arranges or generates the initial configuration of “living” cells, which then change according to the rules. Despite the simplicity of the rules, a huge variety of shapes can arise in the game.

Analogues

“The game of Life” is not as popular a game as tanks or tennis, but it is the most interesting. The main task is to set up such a configuration at the beginning so that at the end of each epoch there are still living cells that can continue the lineage. (figure 1).

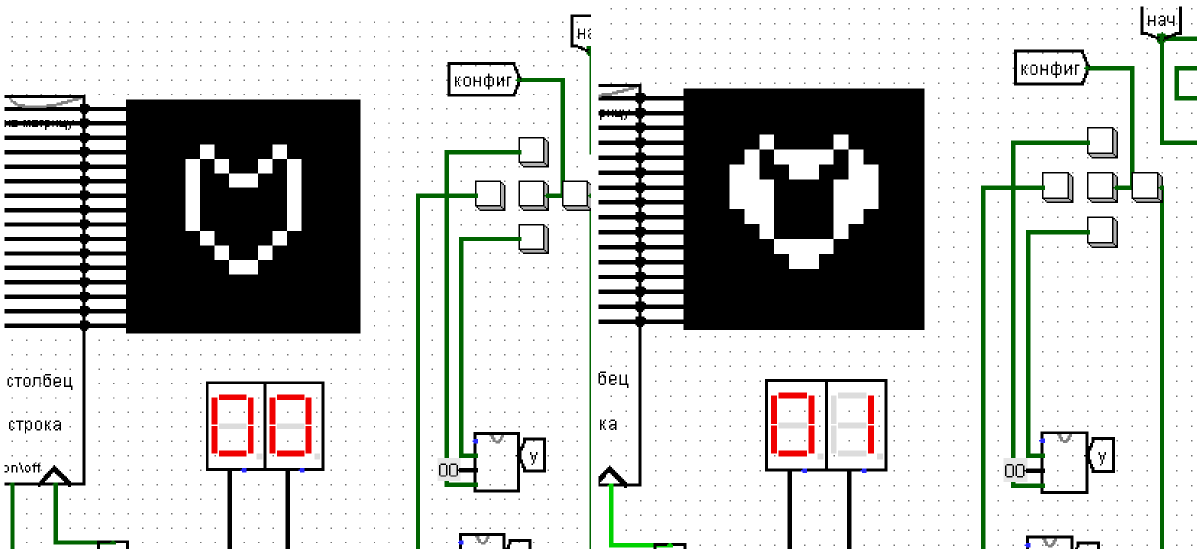


Figure 1 – The birth of a new “living” cell

But there are cases where after a few epochs there may be no living cells left at all and they will die of loneliness. (figure 2).

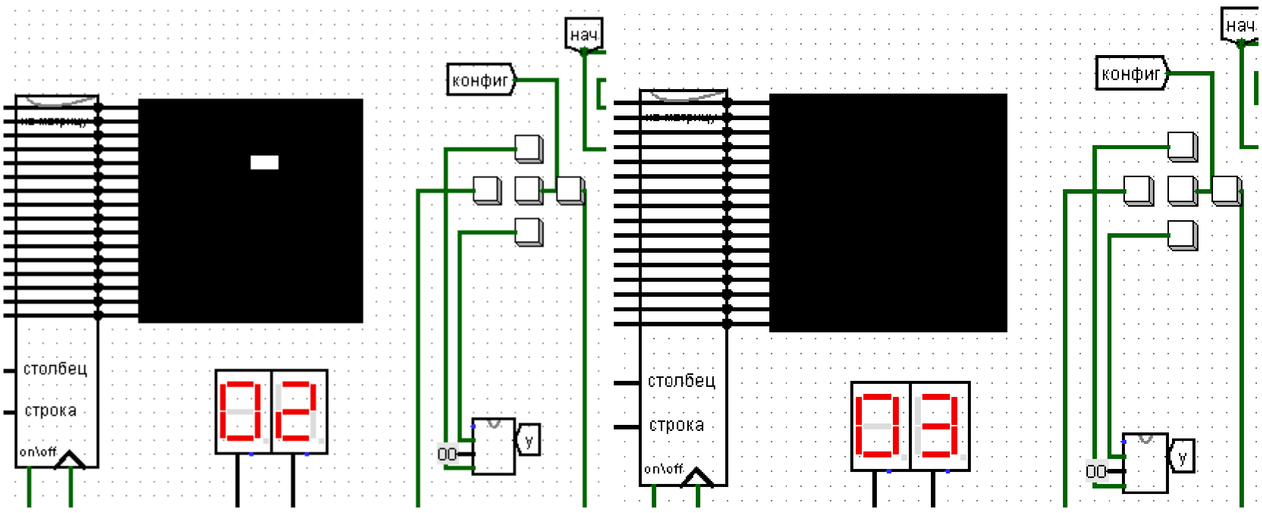


Figure 2 – The cell “dies” because of loneliness

Problem statement and division into subtasks

In order to implement all of the above, we have set tasks that will lead to the realization of “The game of Life”.

Tasks:

1. Simulation of the device, which is built on logic circuits in the program Logisim;
2. Using and connecting to the CDM-8 processor circuitry.

For a clear understanding of all stages of work, we have divided tasks into subtasks.

Subtasks:

1. Dividing the game implementation into Hardware and Software parts;
2. Hardware design and implementation;
3. Development and implementation of the Software;
4. Checking Software and Hardware collaboration;
5. Fixing bugs and optimizing the game (if it’s possible).

Hardware

Start screen

We have implemented the game (figure 3). The player sees the control buttons, the check button and the field of 16x16. The edges of the field are connected to each other as if they were “stitched”.

There are two circuits connected to the matrix, which are responsible for memory and buffer. They are used to write to the cells and output to the matrix. The circuit closest to the matrix is responsible for blinking red of the cell where the player is at the moment. The location of the live cell and its state can be selected using the controls on the right side of the playing field. After setting the initial configuration, the player can start the game by clicking on the button to the right of the navigation. The process will be started.

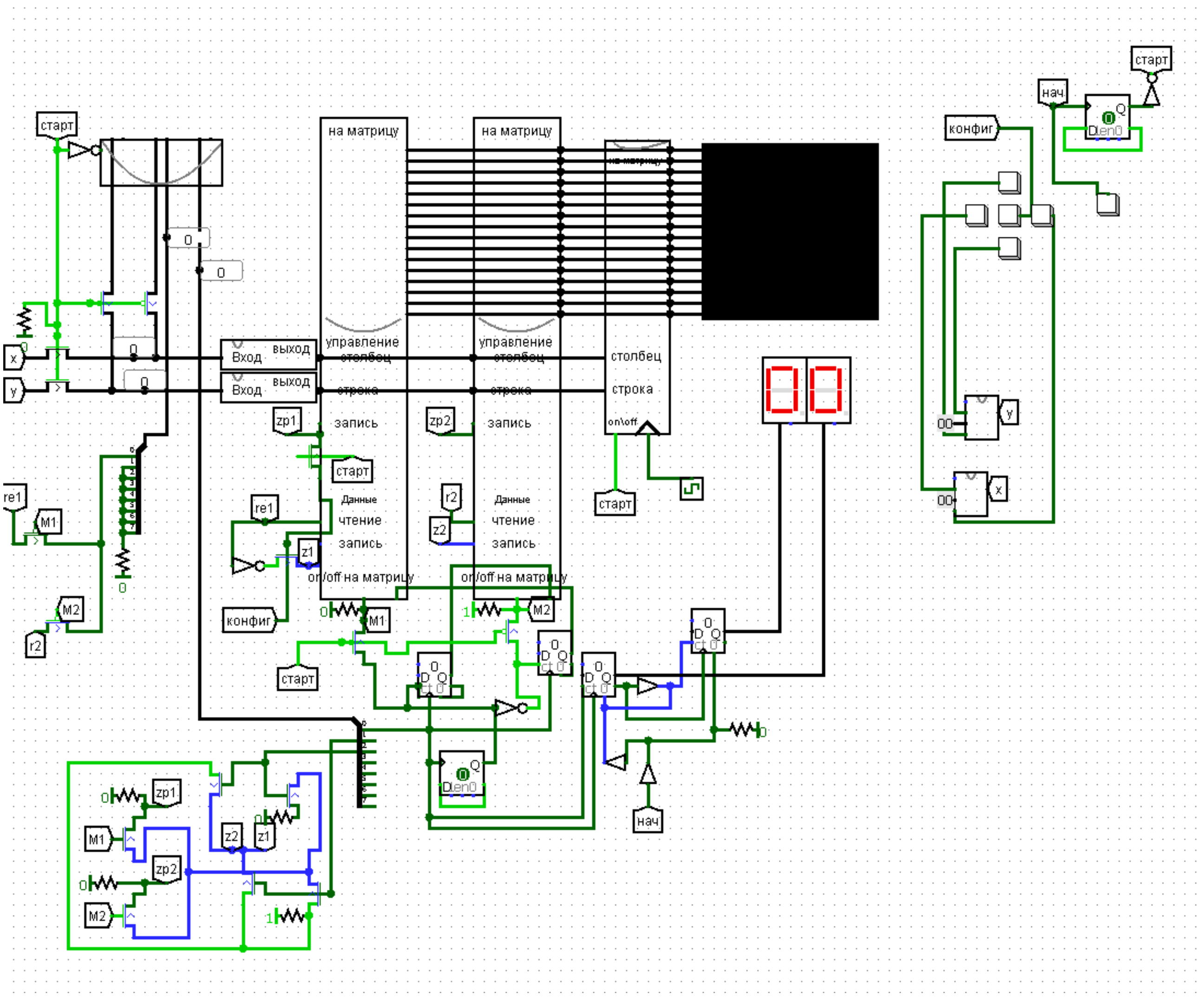


Figure 3 – startup screen

CDM

Here is an implementation of CDM. When an attempt is made to write something to RAM, Hardware intercepts communication with the CDM and RAM. Then certain cell addresses are assigned to the memory and as soon as the address on the address bus matches the address that is defined for a particular device, it appears as a memory cell for the CDM.

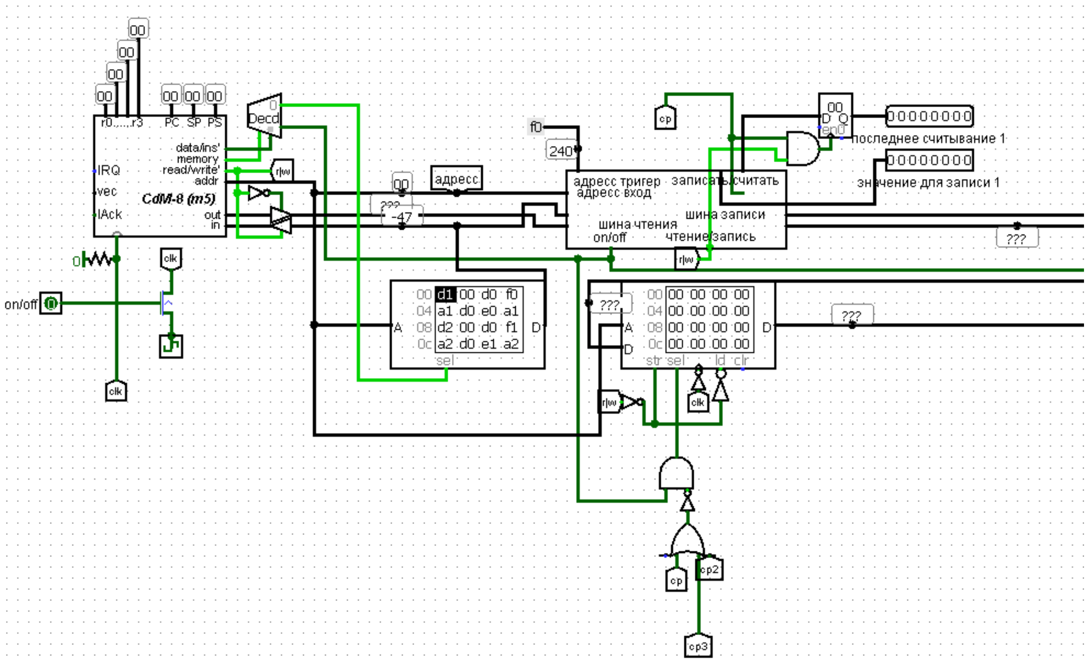


Figure 4 – first part of the CDM

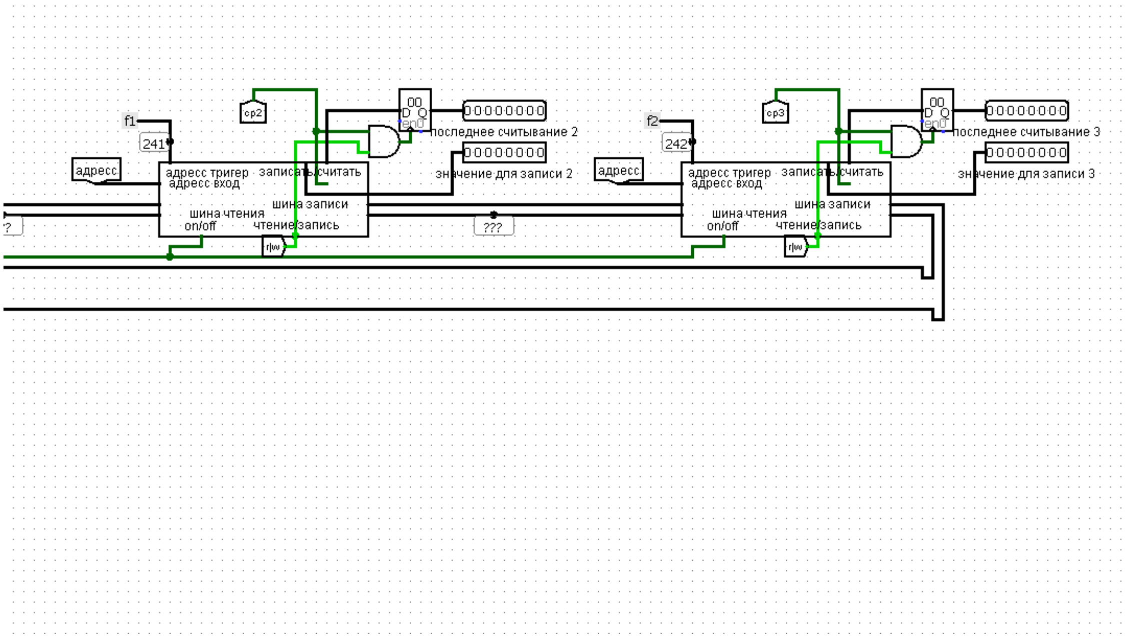


Figure 5 – first part of the CDM

Filed coordinated

In memory the values are stored by coordinates which we set manually at the beginning. They are set by columns and rows. The minimum coordinate value is (0;0) and the maximum is (15:15).

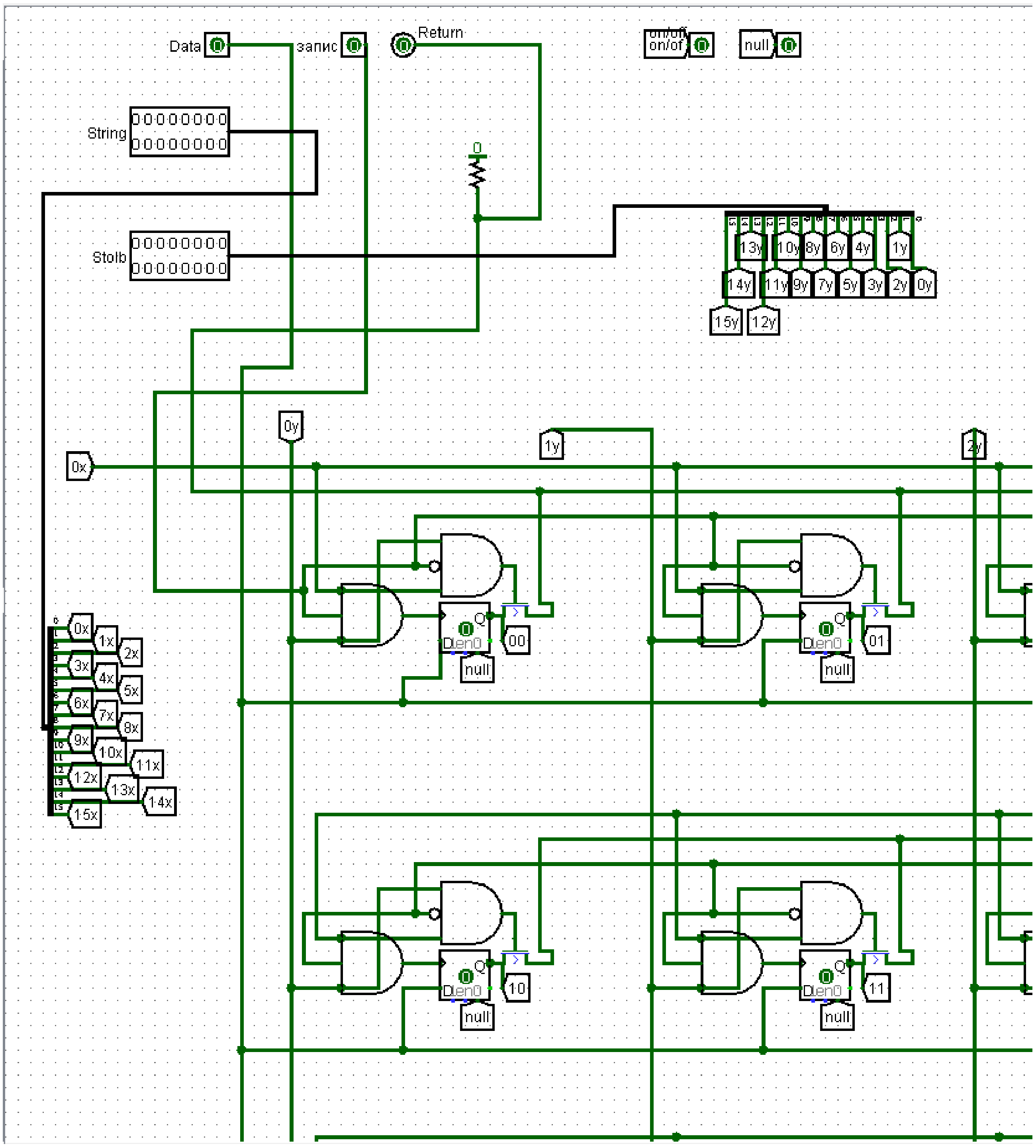


Figure 6 – the state of the field coordinates at the beginning

As an example, let's choose the initial coordinates. Let it be (0;0) (figure 7).
In Data comes the value to be written to the memory cell. Record is the input to which the cycles come. On the rising cycle the value is written. And Return is where the value that is currently in the cell comes in.

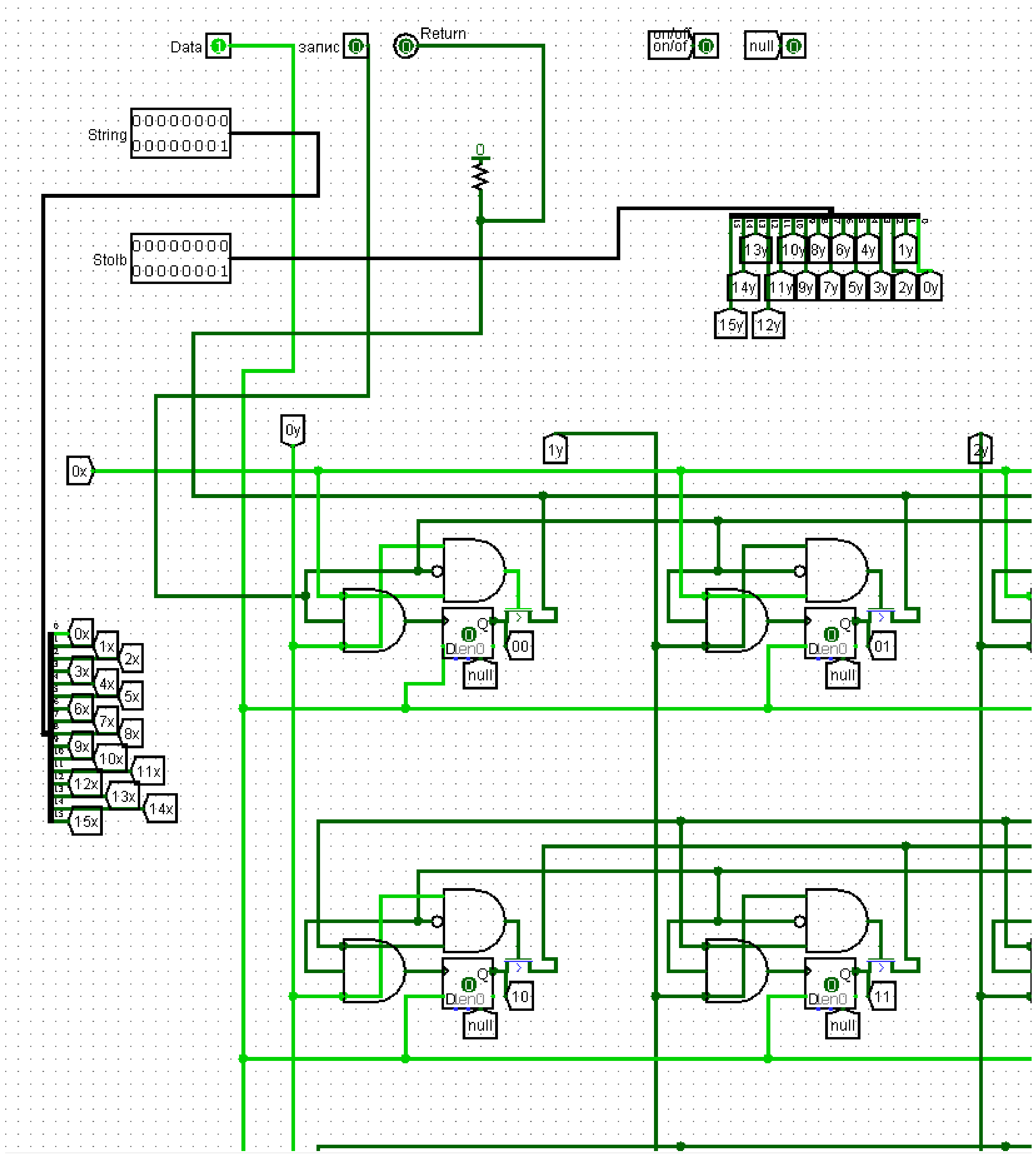


Figure 7 – ehe state of the field cell with the selected coordinate

Conversion from 8 bits to 16 for coordinates

For the correct operation of the game it is necessary to set the coordinate state by a hexadecimal number. In the diagram (figure 8) the number translation from octal to hexadecimal number system is realized.

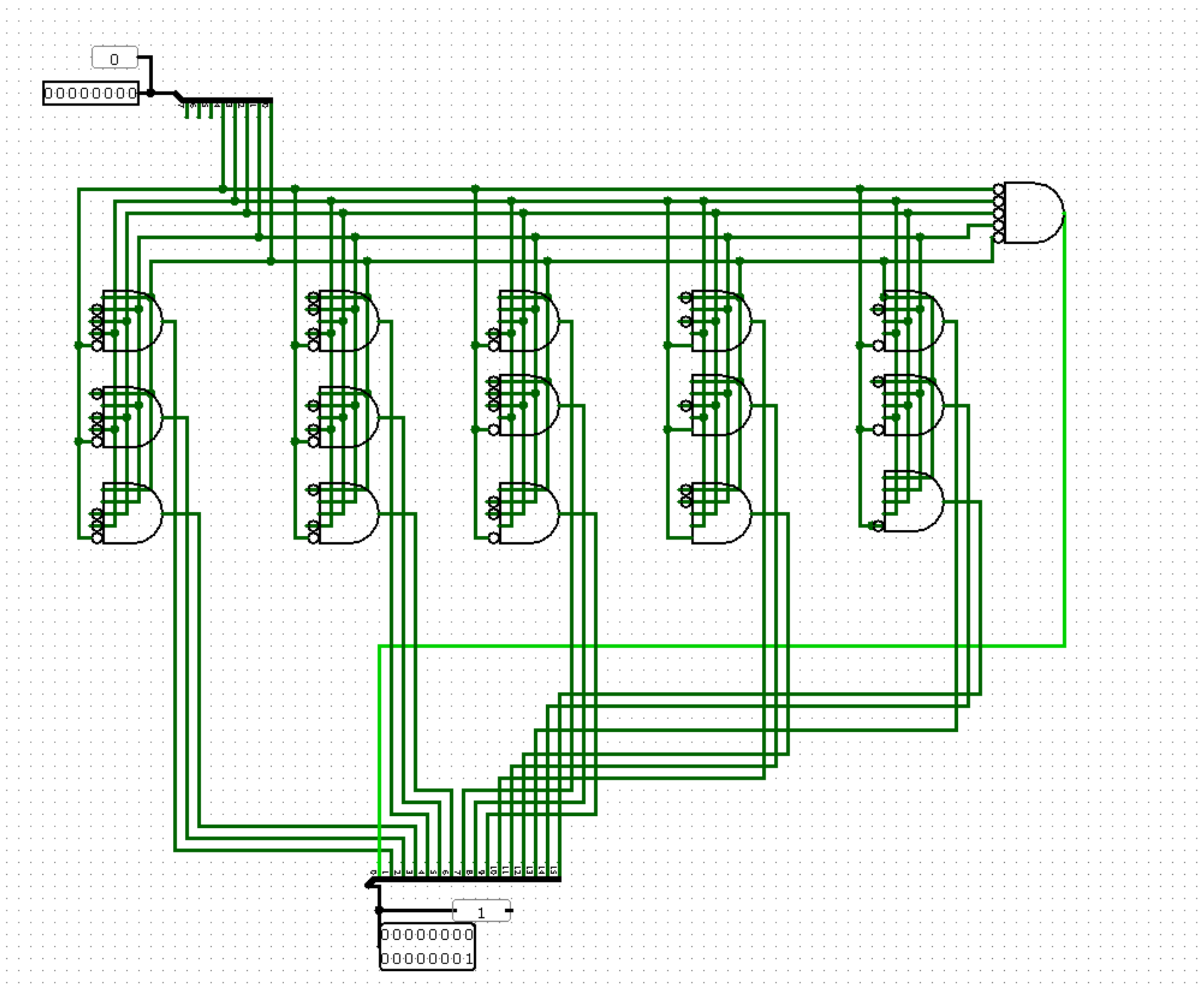


Figure 8 – conversion from octal to hexadecimal

Flashing of the selected cell in red

Flashing a square in red only works when the game is not running. Each “square” is responsible for a cell specified by coordinates. When the beat is 0, no error is caused (figures 9 and 10). When the measure becomes 1, an error occurs and the square becomes red (figures 11 and 12). Since the measures change, the state of the cell will also change.

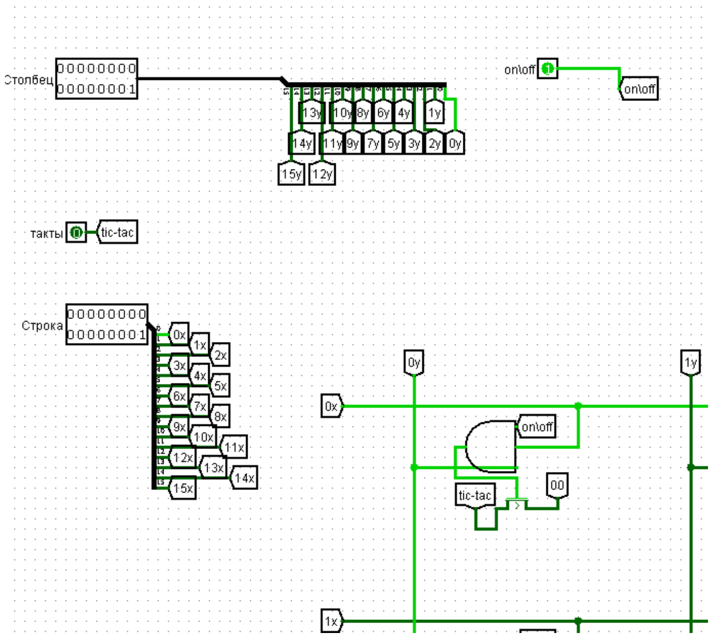


Figure 9 – the cycle is 0

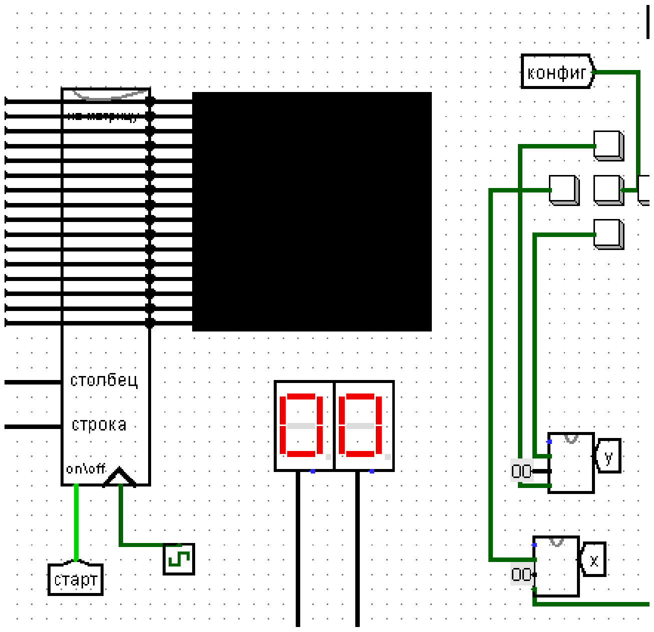


Figure 10 - the state of the cell when the cycle is 0

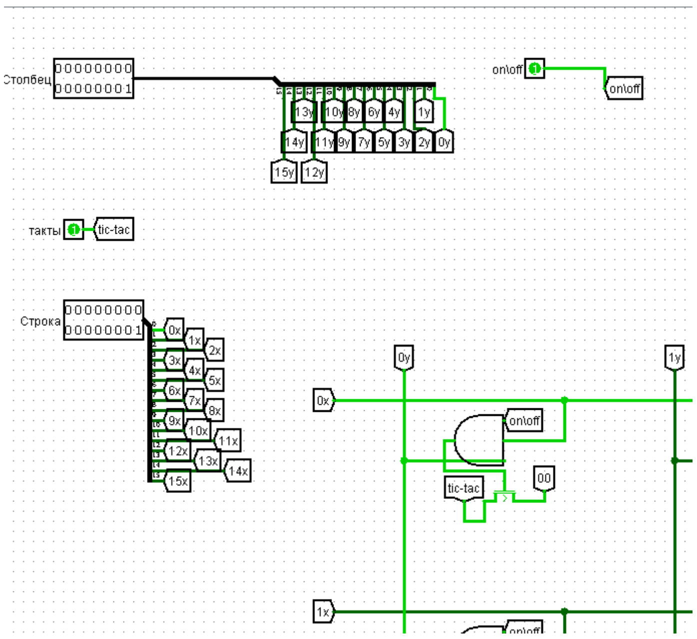


Figure 10 – the tact equals 1

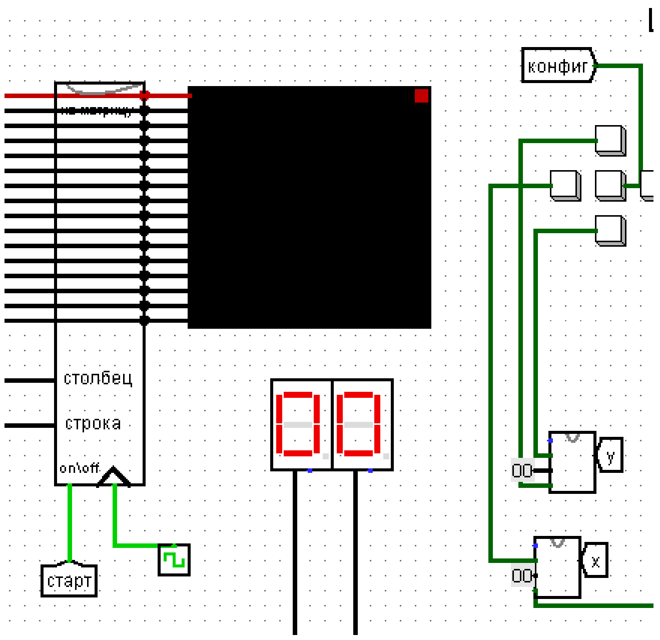


Figure 11 - cell state when the cycle is 1

There are 3 buses from the CDM and the address given to the device. If the device address and the address on the bus match, the transistors lock the bus. They then check to see if it is a read or write. They then write data to the CDM that came to the device or write data to the device from the CDM.

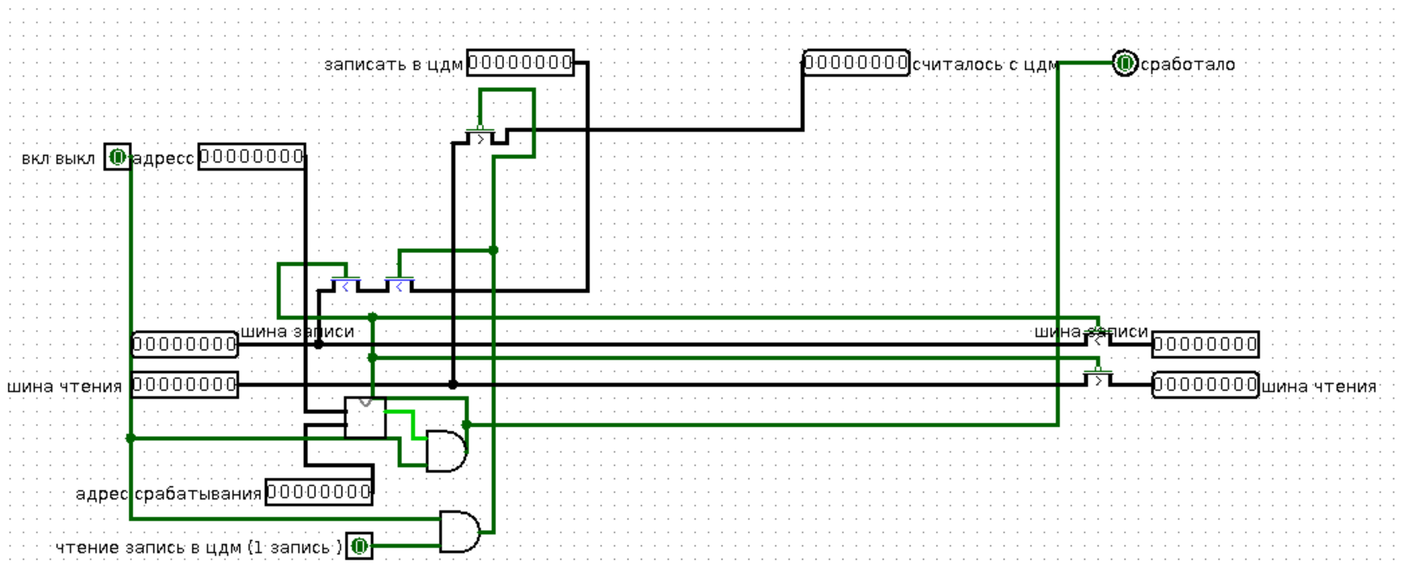


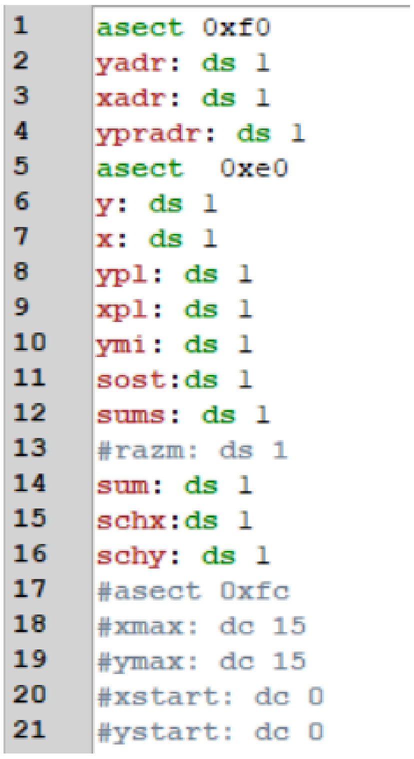
Figure 12 - work with CDM

Software

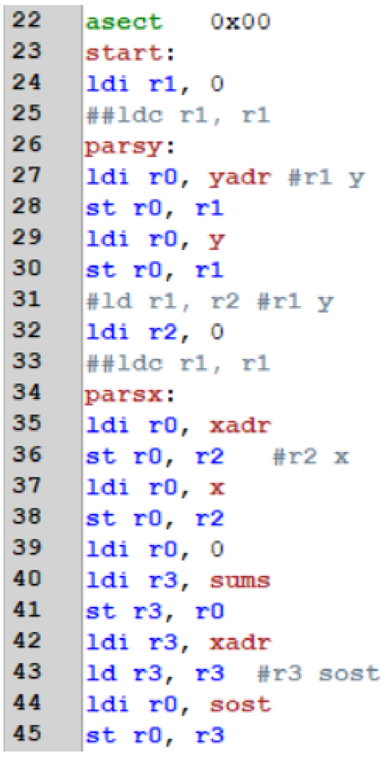
This part of the project is implemented with the CocoIDE development environment. It is based on the Assembler programming language, which is designed to write instructions on the Harvard architecture for the CdM-8 processor from the CdM-8-mark-5-banks library.

Our code starts by assigning memory locations to variables (figure 13). The first 4 lines are the memory cells to which the hard devices are bound.

Then we start the enumeration of cells (figure 14). In the first iteration it zeroes in on X and Y, and in subsequent iterations it only zeroes in on X. It also records the coordinates of the current cell in the X and Y variables, and records the state of that cell (alive or dead) in the “sost” variable.



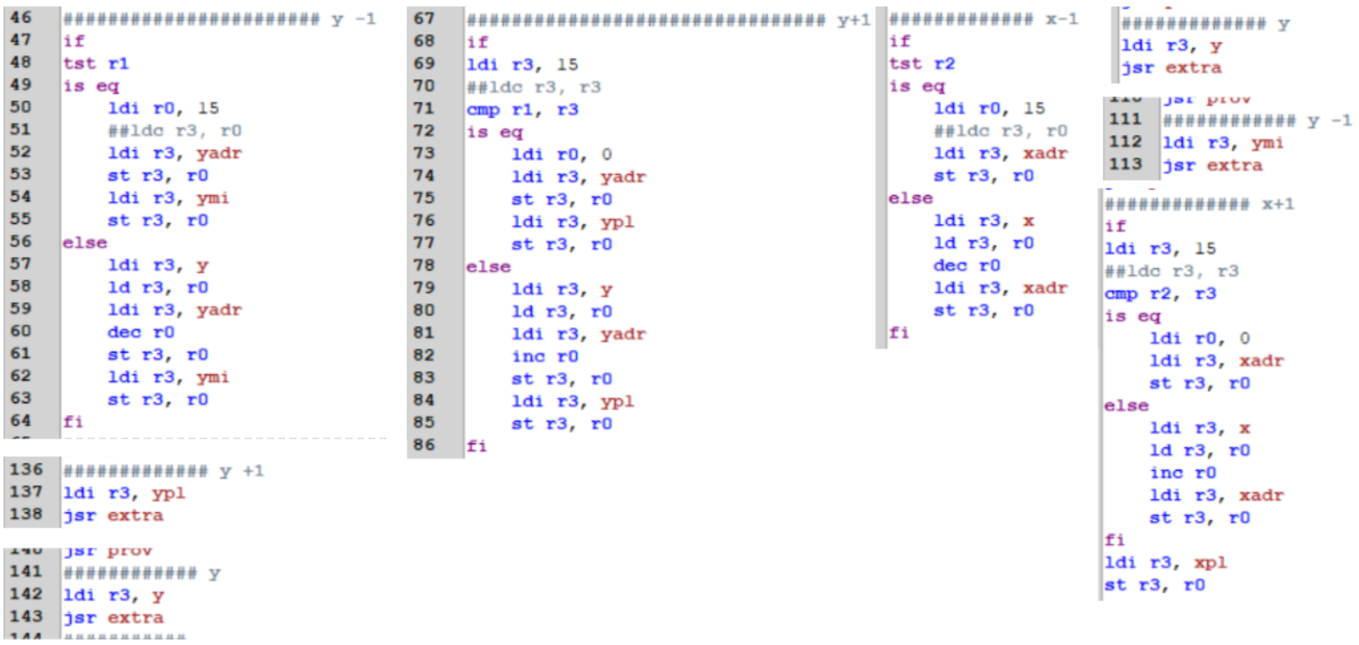
```
asect 0xf0
yadr: ds 1
xadr: ds 1
ypradr: ds 1
asect  0xe0
y: ds 1
x: ds 1
ypl: ds 1
xpl: ds 1
ymi: ds 1
sost:ds 1
sums: ds 1
#razm: ds 1
sum: ds 1
schx:ds 1
schy: ds 1
#asect 0xfc
#xmax: dc 15
#ymax: dc 15
#xstart: dc 0
#ystart: dc 0
```



```
asect   0x00
start:
ldi r1, 0
##ldc r1, r1
parsy:
ldi r0, yadr #r1 y
st r0, r1
ldi r0, y
st r0, r1
#ld r1, r2 #r1 y
ldi r2, 0
##ldc r1, r1
parsx:
ldi r0, xadr
st r0, r2    #r2 x
ldi r0, x
st r0, r2
ldi r0, 0
ldi r3, sums
st r3, r0
ldi r3, xadr
ld r3, r3  #r3 sost
ldi r0, sost
st r0, r3
```

Figure 13, 14 - variable labels

These parts are responsible for enumerating the neighbors of a cell (figure 15). They find neighbor coordinates while checking for edges. For example, if the current cell has a Y coordinate of 15, then its Y+1 coordinate will be 0. This is how the edges are "stitched" together.



```
######################## y -1
if
tst r1
is eq
    ldi r0, 15
    ##ldc r3, r0
    ldi r3, yadr
    st r3, r0
    ldi r3, ymi
    st r3, r0
else
    ldi r3, y
    ld r3, r0
    ldi r3, yadr
    dec r0
    st r3, r0
    ldi r3, ymi
    st r3, r0
fi
```

```
############# y +1
ldi r3, ypl
jsr extra
```

```
jsr prov
############ y
ldi r3, y
jsr extra
```

```
################################## y+1
if
ldi r3, 15
##ldc r3, r3
cmp r1, r3
is eq
    ldi r0, 0
    ldi r3, yadr
    st r3, r0
    ldi r3, ypl
    st r3, r0
else
    ldi r3, y
    ld r3, r0
    ldi r3, yadr
    inc r0
    st r3, r0
    ldi r3, ypl
    st r3, r0
fi
```

```
############# x-1
if
tst r2
is eq
    ldi r0, 15
    ##ldc r3, r0
    ldi r3, xadr
    st r3, r0
else
    ldi r3, x
    ld r3, r0
    dec r0
    ldi r3, xadr
    st r3, r0
fi
```

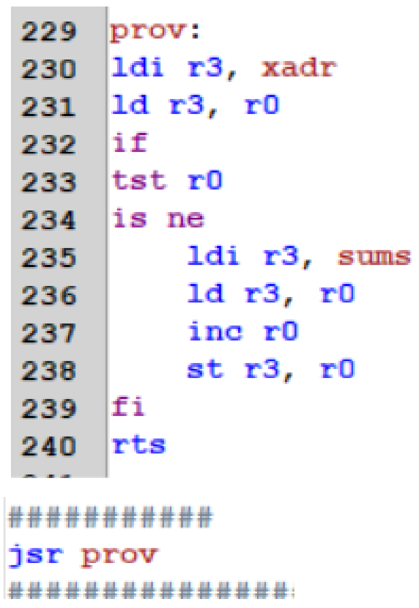
```
############# y
ldi r3, y
jsr extra
```

```
jsr prov
############ y -1
ldi r3, ymi
jsr extra
```

```
############# x+1
if
ldi r3, 15
##ldc r3, r3
cmp r2, r3
is eq
    ldi r0, 0
    ldi r3, xadr
    st r3, r0
else
    ldi r3, x
    ld r3, r0
    inc r0
    ldi r3, xadr
    st r3, r0
fi
ldi r3, xpl
st r3, r0
```

Figure 15 - search for neighbors' coordinates

These parts are used to check the state of the cell neighbors (figure 16). If the neighbor is alive, they increment the variable sums.

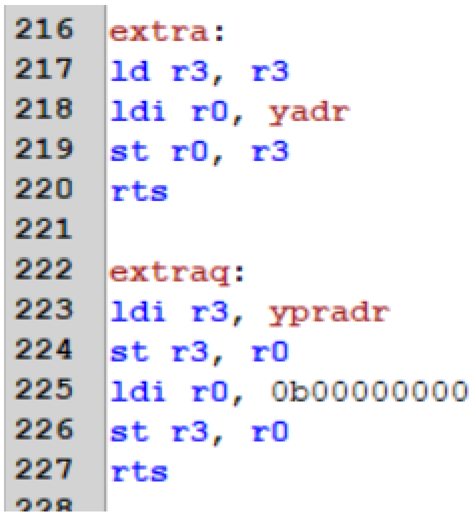


```
prov:
ldi r3, xadr
ld r3, r0
if
tst r0
is ne
    ldi r3, sums
    ld r3, r0
    inc r0
    st r3, r0
fi
rts
```

```
###########
jsr prov
################
```

Figure 16 - neighborhood check

Here we optimized the code to fit into memory (figure 17).



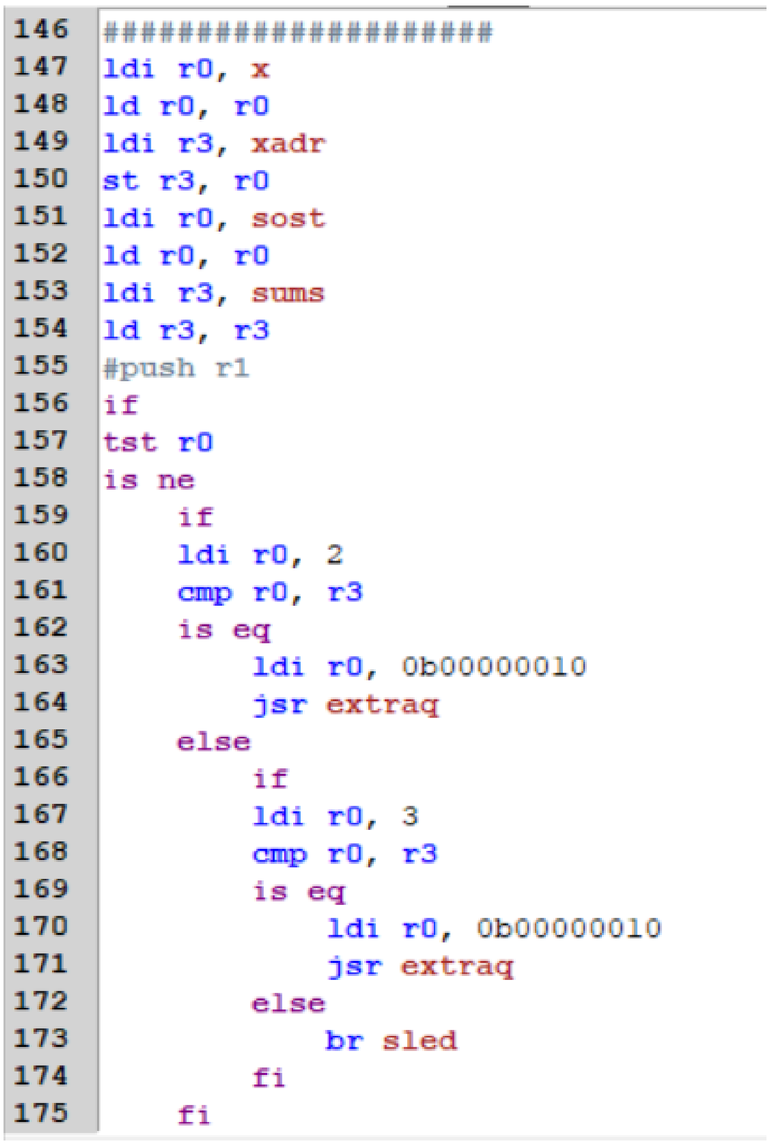
```
extra:
ld r3, r3
ldi r0, yadr
st r0, r3
rts

extraq:
ldi r3, ypradr
st r3, r0
ldi r0, 0b00000000
st r3, r0
rts
```

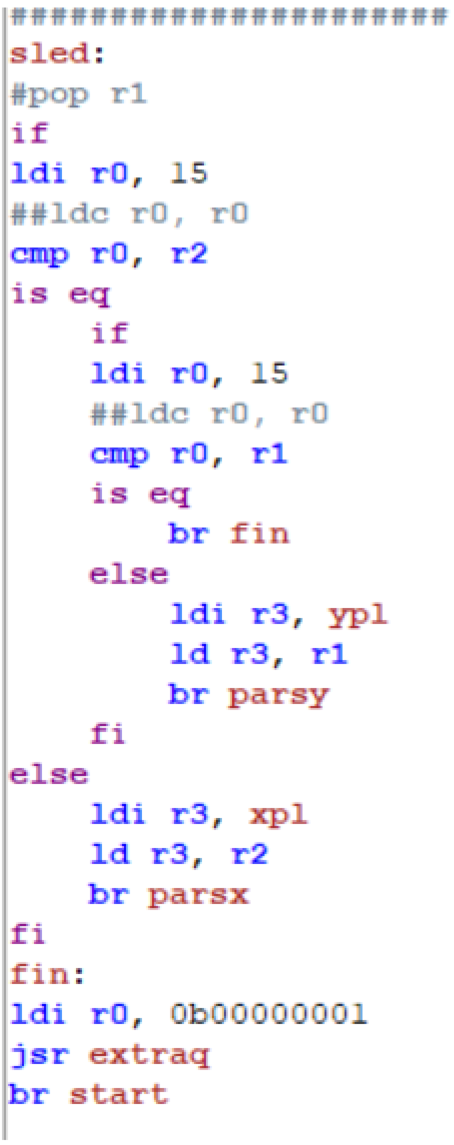
Figure 17 - code optimization

A cell becomes alive if its number of neighbors and current state satisfy the rules of cell life (figure 18).

This is where it goes to the next iteration of the cell enumeration. If all the cells have already been searched, it sends a signal to change the fields and clear the field to be written, and then starts the enumeration again (figure 19).



```
######################
ldi r0, x
ld r0, r0
ldi r3, xadr
st r3, r0
ldi r0, sost
ld r0, r0
ldi r3, sums
ld r3, r3
#push r1
if
tst r0
is ne
    if
    ldi r0, 2
    cmp r0, r3
    is eq
        ldi r0, 0b00000010
        jsr extraq
    else
        if
        ldi r0, 3
        cmp r0, r3
        is eq
            ldi r0, 0b00000010
            jsr extraq
        else
            br sled
        fi
    fi
```



```
########################
sled:
#pop r1
if
ldi r0, 15
##ldc r0, r0
cmp r0, r2
is eq
    if
    ldi r0, 15
    ##ldc r0, r0
    cmp r0, r1
    is eq
        br fin
    else
        ldi r3, ypl
        ld r3, r1
        br parsy
    fi
else
    ldi r3, xpl
    ld r3, r2
    br parsx
fi
fin:
ldi r0, 0b00000001
jsr extraq
br start
```

Figure 18, 19 - "birth" of a cell, go through the cells

Conclusion

The result of this work was the implementation of the "Game of Life". We implemented the project with the help of logic circuits, which we created in the program Logisim, and wrote the software in the programming language Assembler in the development environment CocoIDE.

In the process of implementing the project, we were able to understand how our game works from the inside. Thanks to this we fixed some bugs and errors that were either preventing our game from working or slowing it down.

By implementing the project, we have improved our programming skills and our understanding of circuitry. Also, it is an invaluable experience that will stay with us forever.

Sources

http://www.cburch.com/logisim/docs/2.7/ru/html/libs/index.html

http://www.ccfit.nsu.ru/~fat/