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

Job topic: **‘TETRIS’**

Valeriia Zaichikova, 23930
Egor Kravchenko, 23930
Dmitry Zlobin, 23930

Novosibirsk
2024

ОШИБКА! ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА.

1 TERMS AND ABBREVIATIONS

CdM8 Mark5	Coco de Mer 8 Mark5 – the processor which was made in Logisim
LED	Light-emitting diode
Tetromino (tetrominoes)	Figure for the game of Tetris, geometric shape composed of four squares, connected orthogonally
Randomizer	A device to generate a value from a range randomly
BCD	Binary-coded decimal
UI	User interface

2 INTRODUCTION

The subject of our collaborative project is focused on Tetris, one of the earliest and most renowned video games globally, developed in 1984 by the Soviet programmer Alexey Pazhitnov. This game quickly achieved widespread acclaim internationally.

The project is based on the standard rules of the game Tetris. On a 10x20 playing field, various figures, called "*tetrominoes*", fall from above. If figures are collected in one full line, this line disappears, and the player receives a certain number of points. The number of points awarded for each line is as follows: 1 line – 100 points, 2 – 300, 3 – 700 and 4 – 1500. If a new figure cannot be accommodated on the playing field, the game will be over. The objective for the player is to maintain their position for as long as possible or to achieve the highest score.

The project was implemented on **Logisim** with a **CdM8 Mark5** processor and writing code for it via **CocoIDE**. All of the above project tools are part of our Digital Platforms subject curriculum.

3 PURPOSE AND AREA OF APPLICATION

The objective of the program is to implement it on low-level platforms such as assembler and logic circuits. The program is designed for use in gaming and entertainment spheres.

4 FUNCTIONAL CHARACTERISTICS

Code for our project is used to generate random figures for the gameplay. The main problem with the Random Generator is that a «drought» or «flood» can happen. This means that some figure may not fall out for a long time or, on the contrary, go several times in a row. The Random Generator in our code solves these problems by using the «bag» system. In this system, a list of shapes is placed in a «bag», after which the shapes are randomly removed from it one by one until the «bag» is empty. When it is empty, the pieces return to it and the process repeats. The Random Generator has a «bag» of size 7 (7-bag), that is, a "bag" filled with each of the 7 tetrominoes.

CdM-8 full source code for Harvard architecture

```
asect 0x00

precompile:
  ldi r2, 0b00000001
  ldi r1, time
  st r1, r2
  ldi r1, seed
  st r1, r2
  ldi r1, status
  st r1, r2
  ldi r2, 0b00000111
  ldi r1, mask
  st r1, r2

mainLoop:
  ldi r2, figures
ldi r3, mask
ld r3, r3
while
        tst r3
stays nz
        ld r2, r0
        if
                tst r0
        is z
                br random
        else
                inc r2
                dec r3
        fi
wend
refill:
        ldi r2, figures
        ldi r3, mask
        ld r3, r3
        while
                tst r3
        stays nz
                ldi r0, 0b00000000
                st r2, r0
                inc r2
                dec r3
        wend
random:
        ldi r1, seed
                ld r1, r1
        ldi r0, time
                ld r0, r0
        xor r0, r1
                inc r0
                ldi r2, time
                st r2, r0
                ldi r0, mask
        ld r0, r0
        if
                and r1, r0
        is z
                ldi r0, mask
                ld r0, r0
        fi
                ldi r1, seed
                st r1, r0
        ldi r1, figures
        add r0, r1
        dec r1
        ld r1, r2
        if
```

```
                tst r2
        is nz
                ldi r3, figures
                ldi r2, 0b00000111
                while
                        tst r2
                stays nz
                        ld r3, r1
                        if
                                tst r1
                        is z
                                ldi r0, mask
                                ld r0, r0
                                and r3, r0
                                inc r0
                                add r3, r1
                                br output
                        else
                                inc r3
                                dec r2
                        fi
                wend
                br refill
        fi
output:
        do
                ldi r2, 0b00000001
                ldi r3, 0b11110001
                st r3, r2
                ld r3, r3
                tst r3
        until nz
        ldi r3, 0b00000001
        st r1, r3
        ldi r3, 0b00000000
        ldi r1, figure
        st r1, r0
        ldi r1, 0xF1
        st r1, r3
br mainLoop

asect 0xd0
figures:
asect 0xe0
time:
asect 0xe1
seed:
asect 0xe2
mask:
asect 0xf0
figure:
```

```
asect 0xf1
status:

end
```

Code parts’ description

A section of code for allocating memory for a "bag", components for a random number generator and working with the CdM-8 processor:

```
asect 0xd0
figures:
asect 0xe0
time:
asect 0xe1
seed:
asect 0xe2
mask:
asect 0xf0
figure:
asect 0xf1
status:
```

The precompile section is used for initial data filling

```
precompile:
  ldi r2, 0b00000001
  ldi r1, time
  st r1, r2
  ldi r1, seed
  st r1, r2
  ldi r1, status
  st r1, r2
  ldi r2, 0b00000111
  ldi r1, mask
  st r1, r2
```

The main cycle of the program contains all the necessary functions for the correct operation of the randomizer. Firstly, we check if there are any available shapes left and refill the bag if necessary.

```
  ldi r2, figures
  ldi r3, mask
  ld r3, r3
  while
        tst r3
  stays nz
        ld r2, r0
        if
              tst r0
        is z
              br random
```

```
                else
                        inc r2
                        dec r3
                fi
        wend
        refill:
                ldi r2, figures
                ldi r3, mask
                ld r3, r3
                while
                        tst r3
                stays nz
                        ldi r0, 0b00000000
                        st r2, r0
                        inc r2
                        dec r3
                wend
```

A function for generating a random number by using a timer and seed, as well as a mask that limits the range of numbers from 0 to 7, after which a check is performed if the number is not 0. This is a feature of the program. Finally, the function checks if the resulting figure has not been used yet and replaces it if necessary.

```
    random:
        ldi r1, seed
                ld r1, r1
        ldi r0, time
                ld r0, r0
        xor r0, r1
                inc r0
                ldi r2, time
                st r2, r0
                ldi r0, mask
                ld r0, r0
                if
                        and r1, r0
                is z
                        ldi r0, mask
                        ld r0, r0
                fi
                ldi r1, seed
                st r1, r0
                ldi r1, figures
                add r0, r1
                dec r1
                ld r1, r2
                if
                        tst r2
                is nz
                        ldi r3, figures
                        ldi r2, 0b00000111
                        while
                            tst r2
                        stays nz
                            ld r3, r1
                            if
                                tst r1
                            is z
                                ldi r0, mask
                                ld r0, r0
                                and r3, r0
                                inc r0
                                add r3, r1
                                br output
                            else
                                inc r3
                                dec r2
                            fi
                        wend
                        br refill
                fi
```

The next part of the code checks if a new figure has been requested and issues it to the hardware:

```
    output:
        do
            ldi r2, 0b00000001
            ldi r3, 0b11110001
            st r3, r2
            ld r3, r3
            tst r3
        until nz
        ldi r3, 0b00000001
        st r1, r3
        ldi r3, 0b00000000
        ldi r1, figure
        st r1, r0
        ldi r1, 0xF1
        st r1, r3
     br mainLoop
```

5 TECHNICAL CHARACTERISTICS

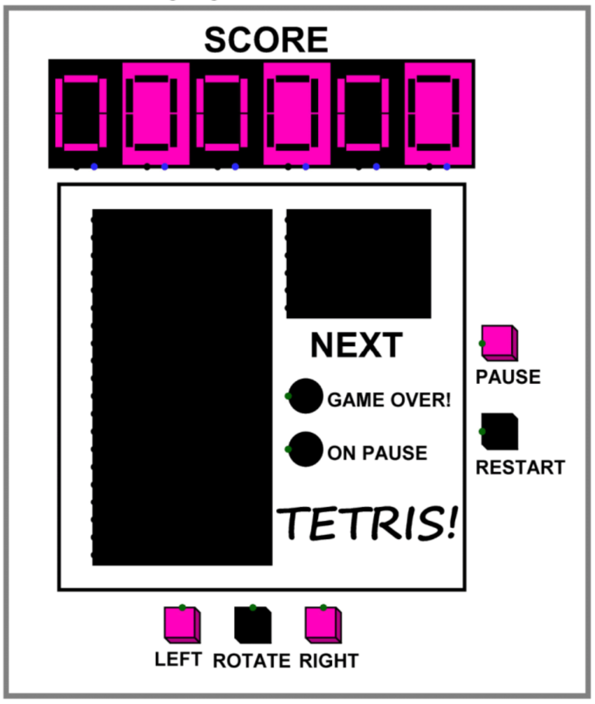
The First section: the Output Display

The primary output component of the program is the display layer (as previously mentioned, its dimensions are **10x20**).

On the smaller matrix **6x8**, the image of the next new output figure is printed from the top left. Additionally, buttons for controlling the game process have been implemented, including buttons for controlling the player (movement to the right, left, turn), as well as "*Pause*" and "*Restart*."

LEDs with corresponding signatures indicate if the game is over (and the player have to restart) and its current status (e.g., whether it is paused). (fig.1) <u>*All components are connected to the field subcircuit.*</u>

fig.1. game ui, the main circuit



The Second Section: the field

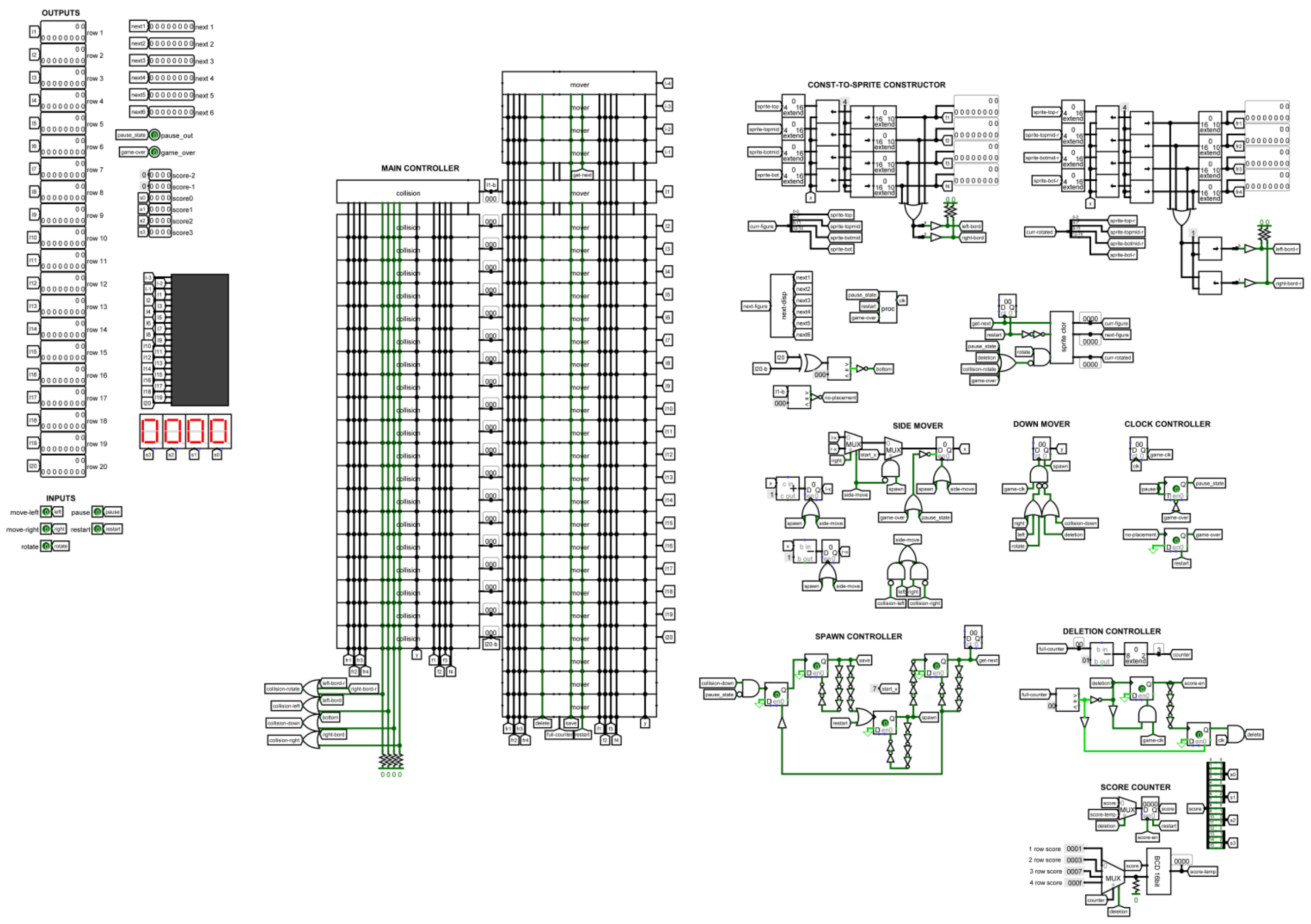


fig.2. the field subcircuit (full)

Inputs:

- **5** input pins for functional buttons (restart, pause, left, right, rotate)

Outputs:

- **20** 10-bit output pins for the main matrix rows
- **6** 8-bit output pins for the “next” matrix rows
- **6** 4-bit output pins for the score hex digit displays
- **2** 1-bit output pins for the game status (game over, paused)

- ***The field Subcircuit: Main controller***

Output to the main display, as shown in the illustration (fig.2) above, is handled by the main controller block, which incorporates collision and sprite control subcircuits (fig.3).

Subcircuits of the same type are connected in series, so that the field is drawn from top to bottom.

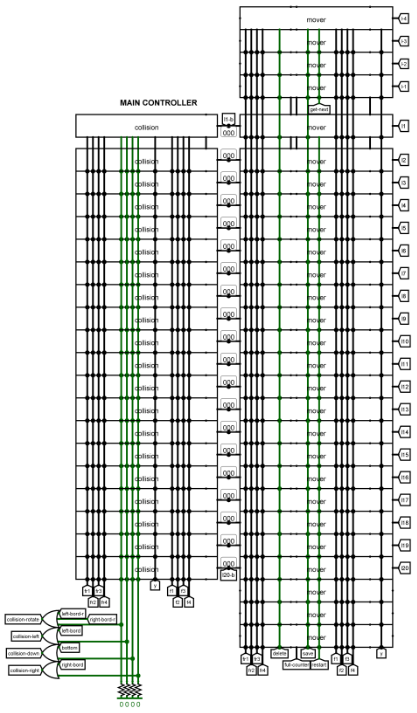


fig.3. main controller block

- ***The field Subcircuit: Const-to-sprite constructor***

The const-to-sprite constructor block performs sprite drawing from a pre-generated and specified constant using bit shifts (fig.4).

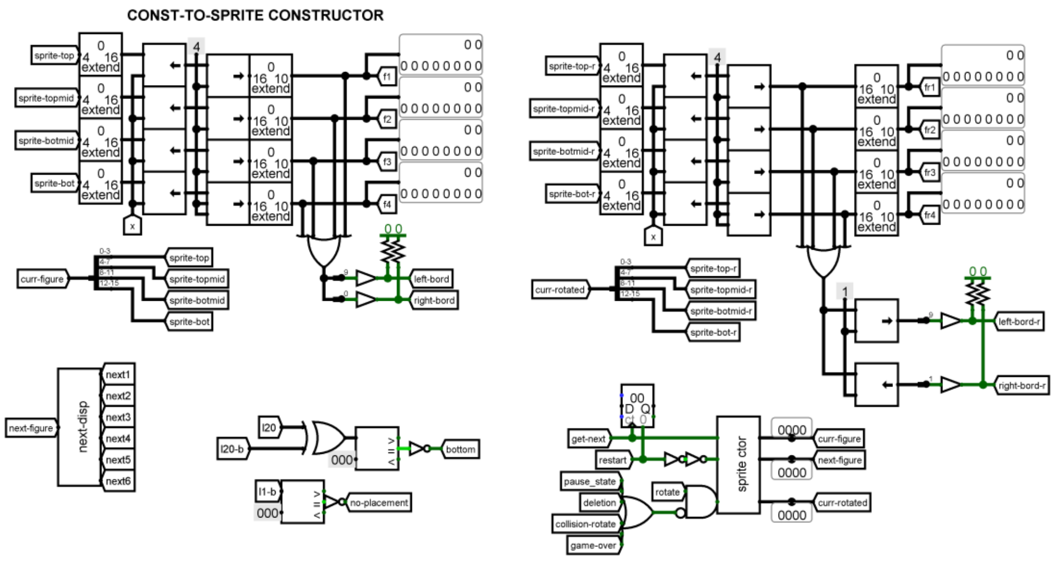


fig.4. const-to-sprite constructor block

• *The field Subcircuit: movement and clock controllers*

The movement of the player's figure on the coordinate axes is controlled using memory elements and arithmetic operations. Using register to store the x value is necessary, since upon the restart the value have to be reset to a specific (in the center of the display), and not to zero. Such behavior is not possible to be implemented with a counter element.

Game process clock must be a lot slower, than the clock for processor, so it is controlled by a counter's overflow. (fig.5)

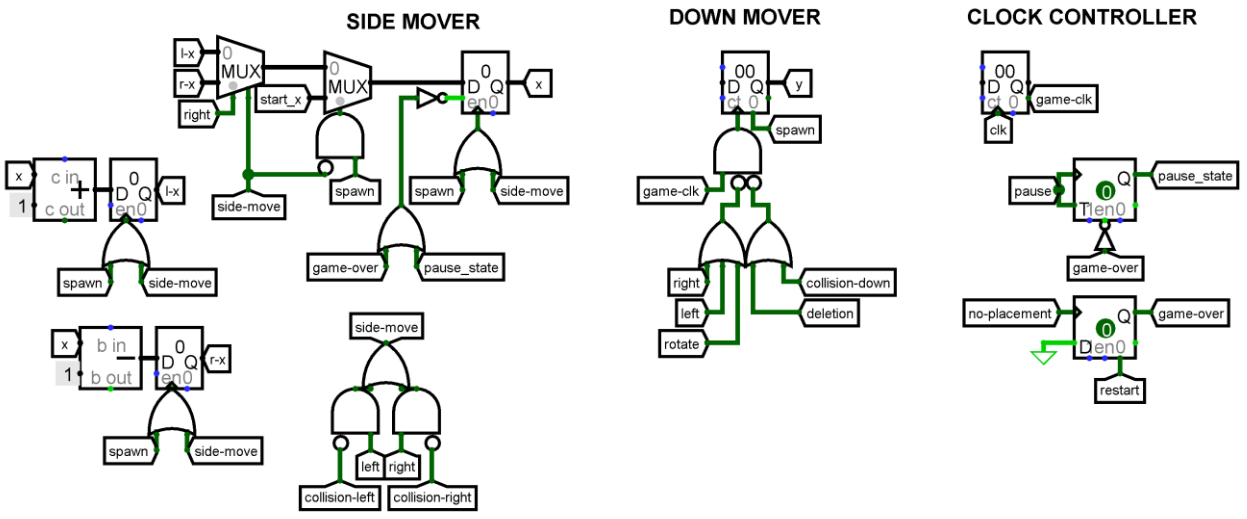


fig.5. movement and clock controllers

• *The field Subcircuit: spawn and deletion controllers, score counter*

In order foe signals to work properly in the gameplay, the sequence of their operation is controlled via D Flip-Flops and delays (made using inverters).

Score counter makes calculation with BCD adder, so it can be displayed on the hex digit displays in the UI circuit. (fig.6)

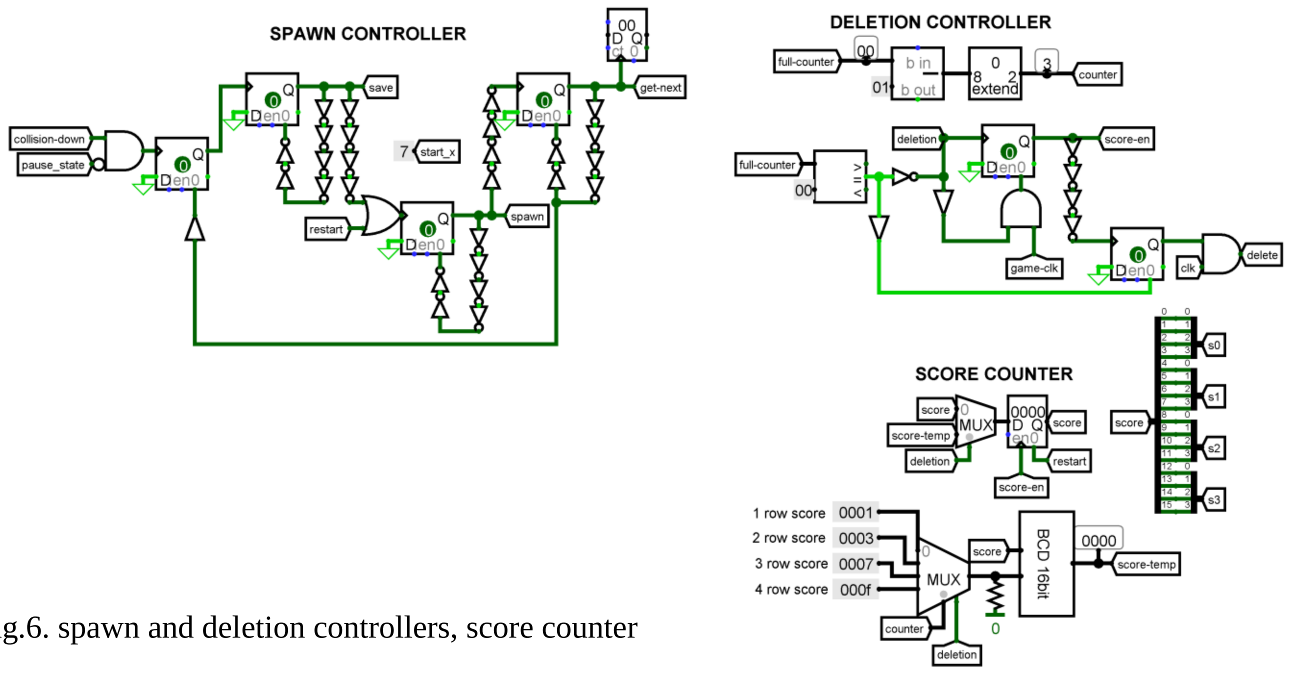


fig.6. spawn and deletion controllers, score counter

The Third Section: Sprite Control subcircuit

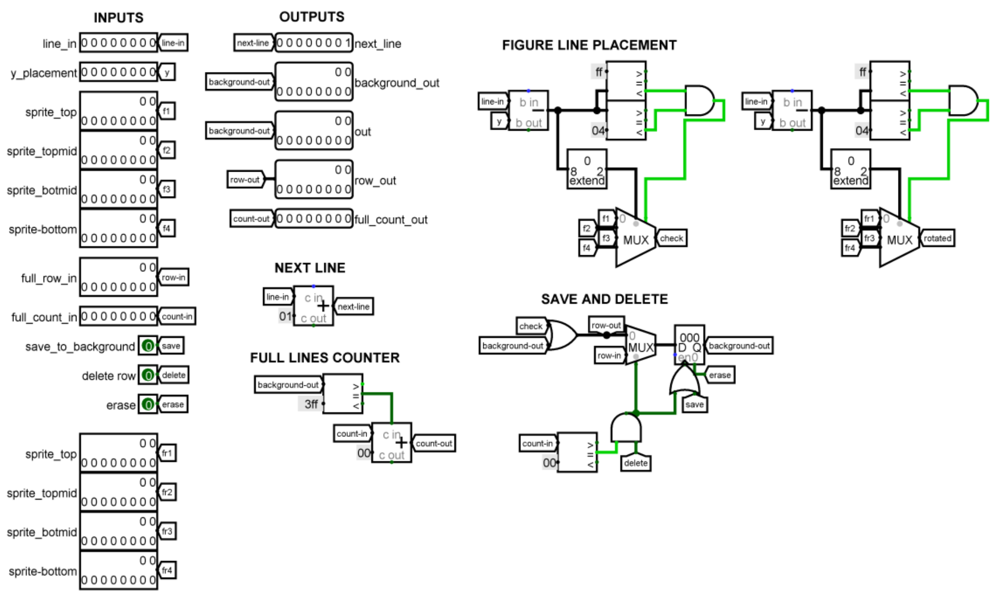


fig.7. sprite controller

The sprite control subcircuit is responsible for controlling the position of sprites by y-position, as well as saving them to the "background" (already set figures), analysing lines for completeness, and deleting complete lines. The value of the background line is stored in a register, whose value is updated when a new shape is saved. The value of both the background and the line being changed is displayed on the playing field.

Lines are analysed sequentially, which allows for the correct implementation of the image shift when collecting a full line in accordance with the game rules and the instant calculation of such lines. (fig.7)

The Fourth Section: Collision subcircuit

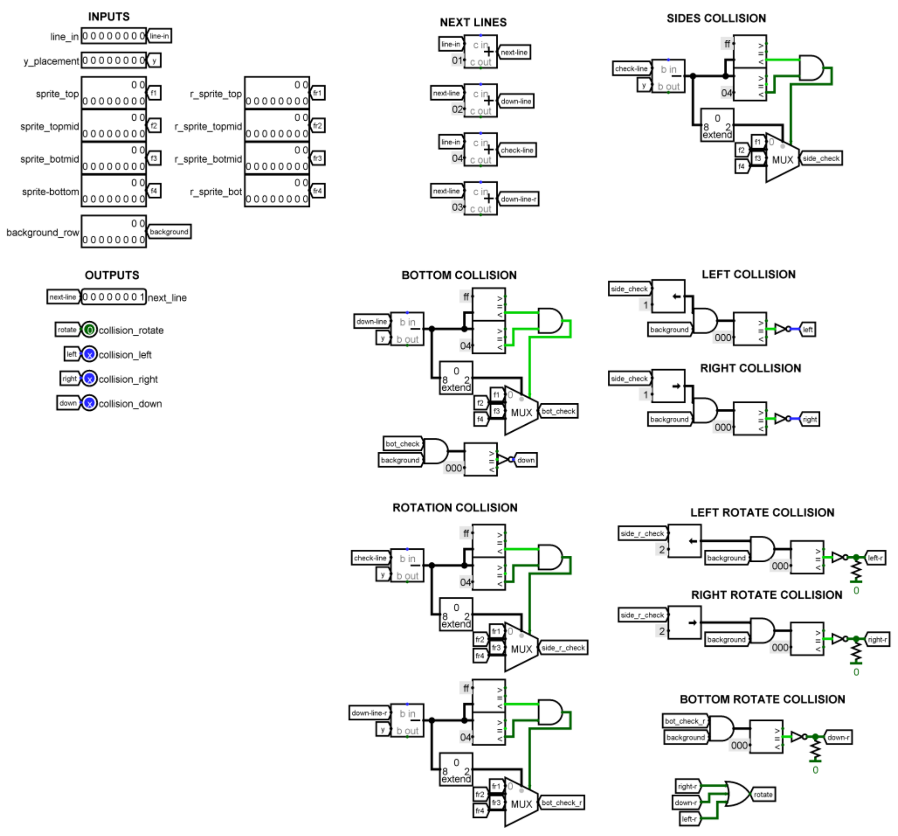


fig.8. collision controller

(fig.8) The collision subcircuit calculates the side and bottom collision using arithmetic operations and compares the changing value of the line with the values of the background, as well as the bottom and side walls of the playing field. The construction is similar in execution to that described above.

The Fifth Section: Get_Figure subcircuit

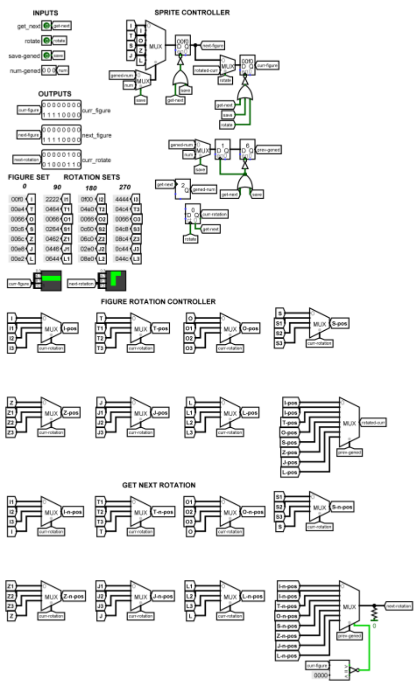


fig.9. get_figure subcircuit

In the get figure subcircuit, the sprite values are managed and rotated based on them. In order to ensure correct and consistent operation of the scheme on restart, a restart value delay is necessary. Furthermore, the process of displaying the sprite on an additional window with the following figure is also set here. *(fig.6)*

The ***get_figure subcircuit*** is responsible for converting a number generated by the ***randomizer*** into one of the seven sprites available in the game. This is stored in the figure set block, along with the values of the numbers generated for them. These values are stored in the corresponding registers.

Additionally, counters and multiplexers are used to realise rotations based on sprite visuals replacement. This is done in a similar way to the existing implementation of the player's figure output, which allows for a cost-effective solution...

The Sixth Section: The Output on the Second Display subcircuit

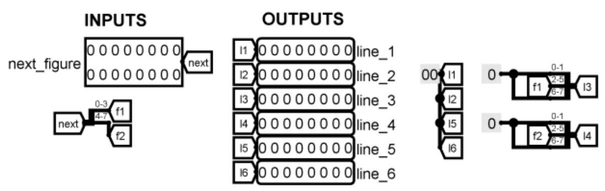


fig.10. second display output

The subcircuit of the output to the additional display *(fig.7)* is created by drawing with the help of constants, as it only requires the display of the figure, not its control or manipulation...

The Seventh Section: BCD 16-bit adder

(fig. 11-13)For this project classic BCD adder was expanded to work correctly with large numbers.

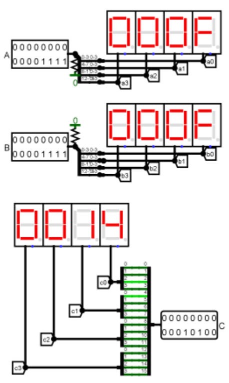
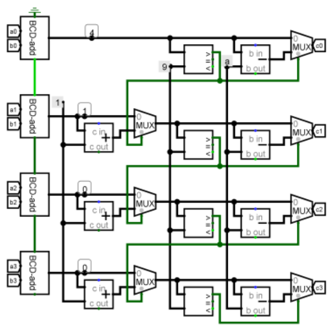



fig.11. 16bit BCD adder

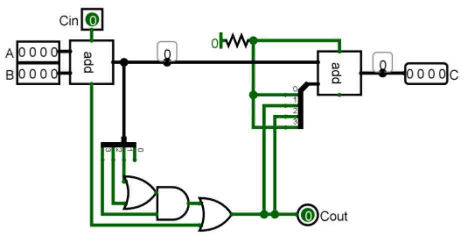


fig.12. BCD adder

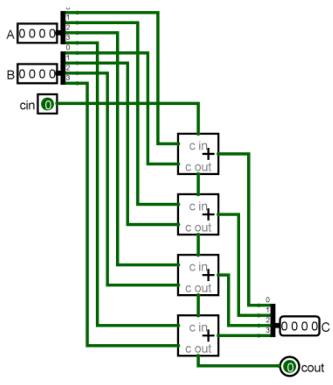


fig.13. full 4-bit adder

The Eighth Section: Processor controller subcircuit

(fig.14) The processor controller circuit is built on Harward architecture with simple IO manager for input and output.

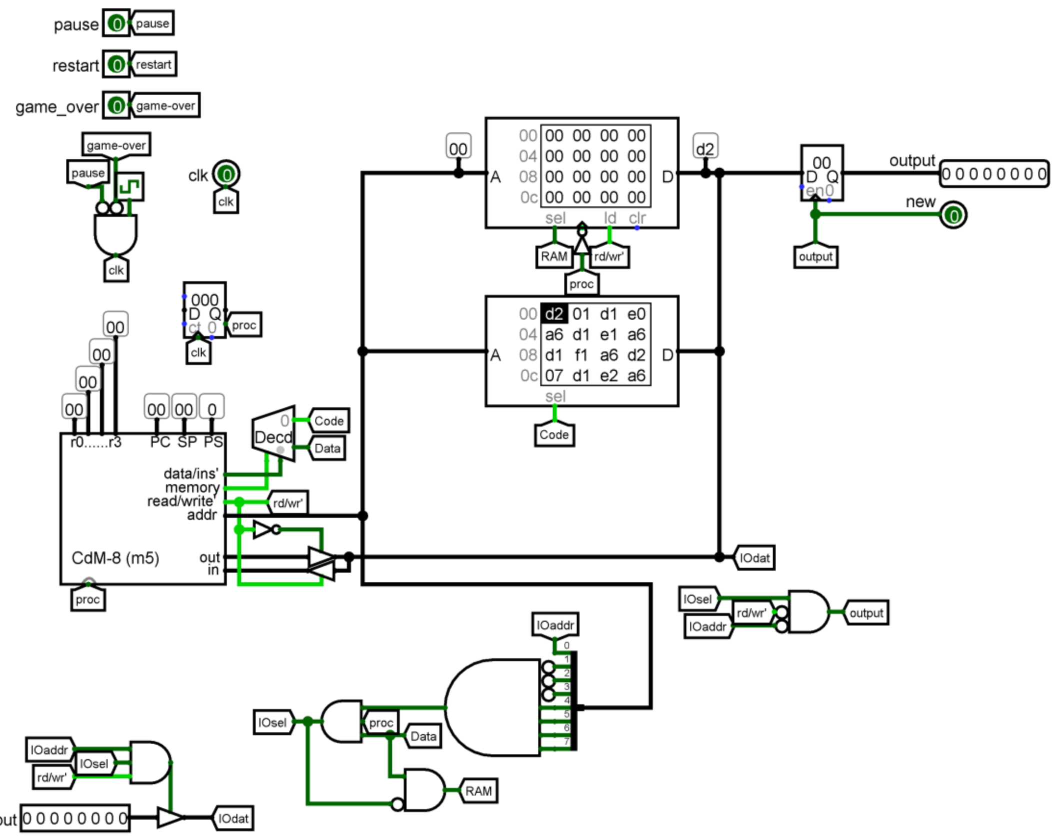


fig.14. processor controller

6 CONCLUSION

The project of implementing the Tetris game on assembler and through logic circuits has been successfully completed. The main goal, to demonstrate the capability of low-level platforms to create a complete gaming application, has been achieved. Functionality and stability of the program have been verified by tests, making it suitable for entertainment purposes.

7 SOURCES USED IN DEVELOPMENT

- http://ccfit.nsu.ru/~fat/Platforms
- http://www.cburch.com/logisim/docs/2.6.0/ru/guide/tutorial/index.html?authuser=1
- *“Computing platforms”, A. Shafarenko and S.P. Hunt, School of Computer Science University of Hertfordshire 2015*
- https://en.wikipedia.org/wiki/Read-only_memory
- https://en.wikipedia.org/wiki/Random-access_memory
- https://tetris.wiki/Super_Rotation_System