MINISTRY OF EDUCATION AND SCIENCE OF THE RUSSIAN FEDERATION

NOVOSIBIRSK STATE UNIVERSITY  
(NSU)

Higher College of Informatics

Department of intelligent systems of thermophysics

Field of study 15.03.06 Deep robotics

Specialty: Mechatronics and robotics

**TERM PAPER**

**Bespalov Sergey Vyacheslavovich**

**Grishchenko Alexander Mikhailovich  
Solopov Ilya Ruslanovich**

The theme of the paper:

**THE «RUSH Space» GAME**

Novosibirsk, 2022

CONTENTS

[INTRODUCTION 3](#_Toc103550056)

[1 REQUIREMENTS DEFINITION 4](#_Toc103550057)

[2 ANALOGUES 5](#_Toc103550058)

[3 HARDWARE 7](#_Toc103550059)

[4 SOFTWARE 38](#_Toc103550060)

[CONCLUSION 45](#_Toc103550061)

[REFERENCES 46](#_Toc103550062)

[APPENDIX 47](#_Toc103550063)

INTRODUCTION

Entertainment software is increasing its demand in the market every year. Video games are a notable example of such software. There are a large number of different genres, and it is constantly updated and enlarged. One such genre is Shoot 'em up, which has a rich history.

Shoot 'em up is a type of video game in which the player-controlled character is most often represented as a spaceship or other vehicle, the main purpose of which is to defeat many enemies using shooting. Enemies in such games are usually various aliens or monsters that attack the player by shooting at him or otherwise. Traditionally, such shooters use a top or side view, and a good reaction is important for a successful passage to dodge enemy fire

The aim of this project is to design and build a Shoot 'em up game on electronic circuits using the CdM-8 processor and its assembly language.

In accordance with the purpose, it was necessary to solve the following problems:

* To study examples of Shoot'em up games (analogues);
* To study and analyze information about the processor, its capabilities, commands and instructions;
* To study and analyze programs for creating electrical circuits;
* To define the functional requirements.

1 REQUIREMENTS DEFINITION

The purpose of this paper term is the design and creation of a Shoot 'em up game based on an electrical circuit with an 8-bit CdM-8 processor included in it.

The following are the functional requirements:

1. Controlled movement of the player;
2. Uncontrolled movement of opponents;
3. Player's shooting;
4. Monster shooting;
5. Destruction of the monsters;
6. Keyboard control (player movement, player shooting);
7. Possibility of winning;
8. Possibility of losing.

2 ANALOGUES

During project creation, we explored Shoot 'em up games. Let's consider some of them.

1. Japanese arcade game "Space Invaders", released in 1978. It can be called one of the first in this genre. This game has all the functional requirements specified in the previous paragraph and additional ones, for example, a life counter and the number of points scored, sound effects, an infinite number of "waves" of enemies, protection in the form of "bunkers" from alien shots.

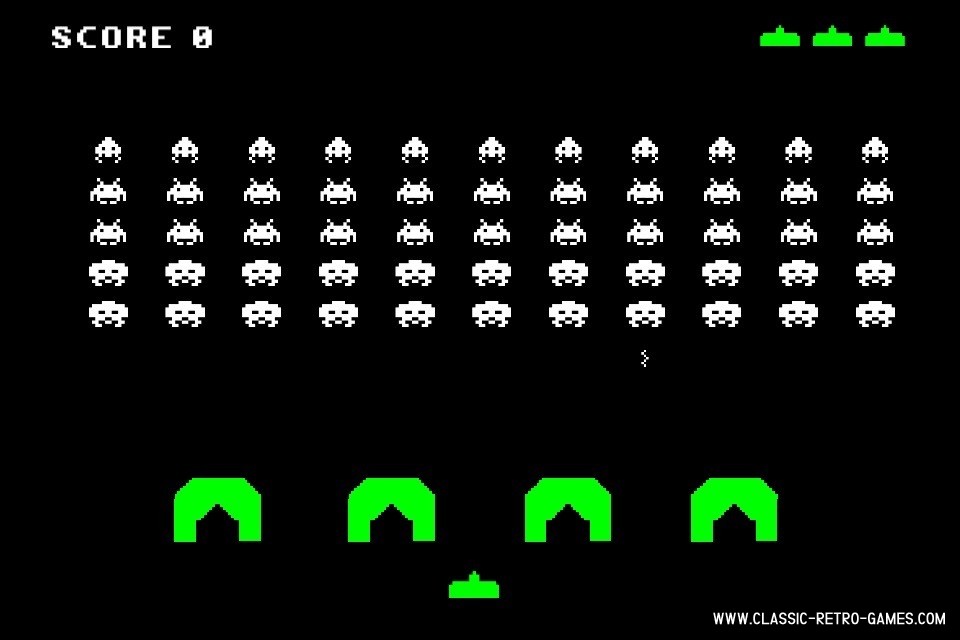


Figure 1 – the gameplay screenshot of "Space Invaders"

1. Another Japanese game released for arcade machines in 1981 is «Galaga». It can be called an improved and more modern version of the previous game. New game mechanics were created in «Galaga», such as an alien tractor beam that can take away the player's control for a short period of time and one life, monsters dive at the player in various trajectories. It is also one of the first games with color RGB graphics.

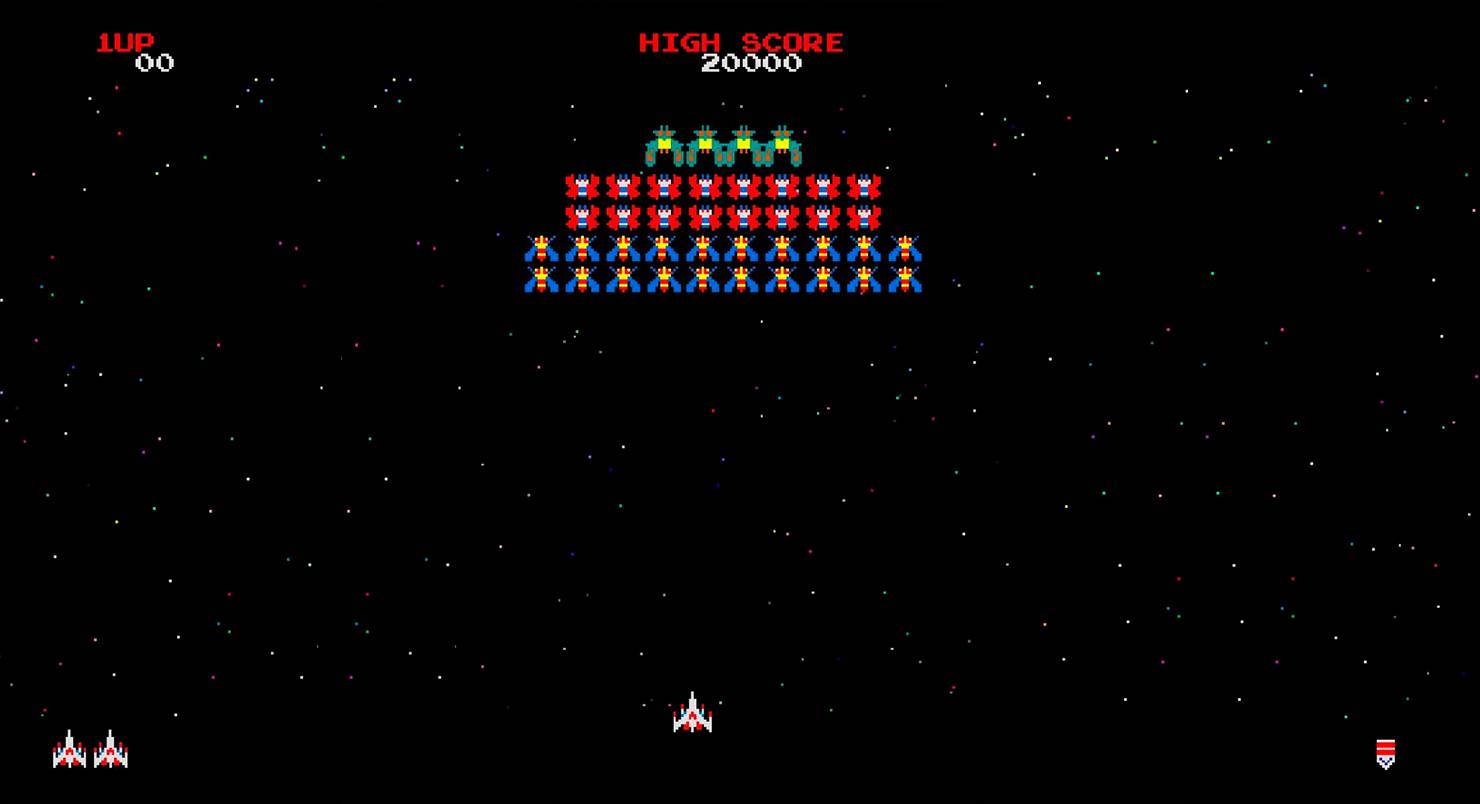


Figure 2 – the gameplay screenshot of "Galaga"

After studying and analyzing these examples, our team decided to create a project based on the design and content of the game "Space Invaders". Due to limited human resources, a not very complex and resource intensive option was needed, which could be done using CdM-8 and our knowledge in the field of circuit design.

3 HARDWARE

The hardware part of our project consists of logical electrical circuits created in the Logisim program, which allows you to model and edit them using a convenient graphical interface. Let’s consider our developments.

**Main Interface**

This is the interface of the game. It is a Pentamenu circuit connected to the matrix and controls: keyboard, "ON / OFF" and "START" buttons. There is also a Cosmsirc circuit here (just a picture). Figure 3 shows the screenshot of the Main Interface circuit.

Изображение выглядит как текст

Автоматически созданное описание

Figure 3 – screenshot of the Main Interface circuit.

To control the game interface, the Pentamenu circuit is used, which contains Menumega and Circ, which contains Gameover and Menu.

**Menu**

The Menu circuit uses only the in input. It is responsible for which of the lines of the "Press Start" screen will be displayed. The decoder receives a five-bit value from the input and raises a bit with a number equal to this value. Then this bit activates one of the 32 buffers, which outputs the required string to out (it will be either zero or a constant) Figure 4 is a screenshot of the Menu circuit.

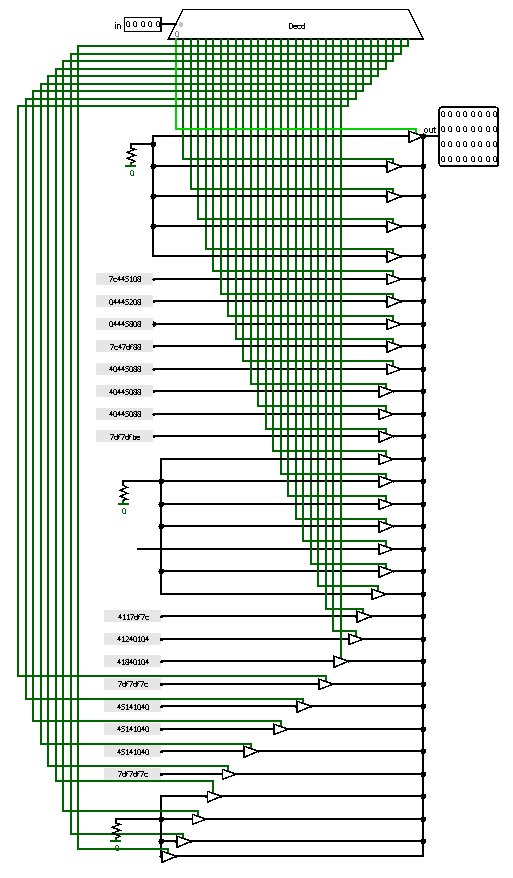


Figure 4 – screenshot of the Menu circuit

**Gameover**

This circuit works similarly to the previous one. A decoder is used that raises one of its outputs depending on the in input. The result is one of the 32 lines of the "Game Over" screen or the "You win" screen. There is also an additional input isgameover, which changes the output value from the multiplexer: if it is raised, then the multiplexers output the value from input 1, otherwise from input 0. This results in either the strings "Game Over" or "You win". Figure 5 is a screenshot of the Gameover circuit.

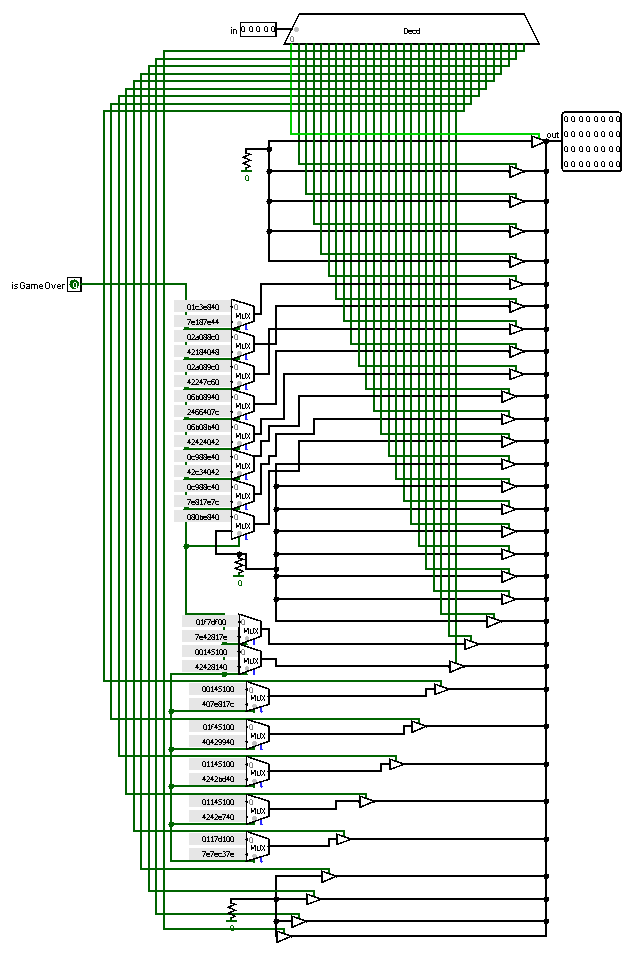


Figure 5 – screenshot of the Gameover circuit

**Circ**

This circuit controls the display of four different lines: the game line, the line from the "Game Over" screen, the line from the "You win" screen, the line from the "Press Start. Four control inputs are used for this: on, start, isgameover and endgame. If the on input is raised, then the “Press Start” screen is displayed, if start, then the 32-bit input game is fed to the output of the circuit. If endgame is raised, then one of the lines of the "You Win" screen is displayed, if both endgame and isgameover are raised, then one of the lines of the "Game Over" screen is displayed. Depending on the number fed to the number circuit and input data, 4 different screen options can be obtained. Figure 6 shows a screenshot of the Circ circuit.

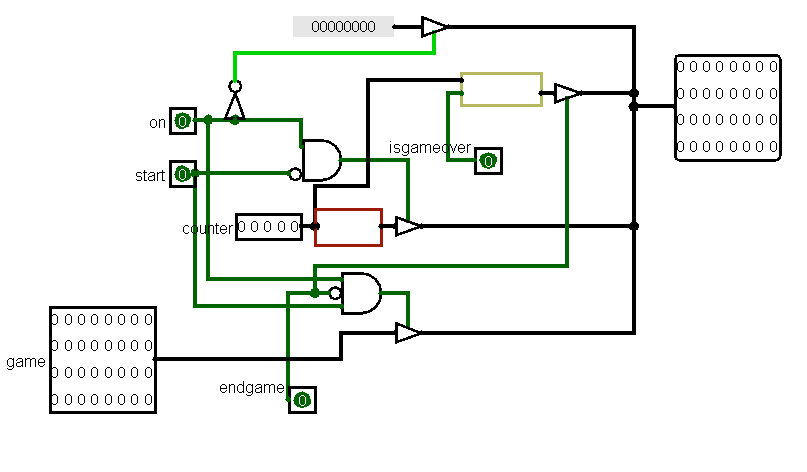


Figure 6 – screenshot of the Circ circuit

**Menumega**

This circuit uses 8 Circ circuits to display 8 rows of one of 4 screens. Screen selection depends on gameover, endgame, on, start inputs. The beg input is incremented to determine which line is the start line. The end output prints the number remaining after incrementing (i.e. multiplied by 8). Figure 7 shows a screenshot of the Menumega circuit.

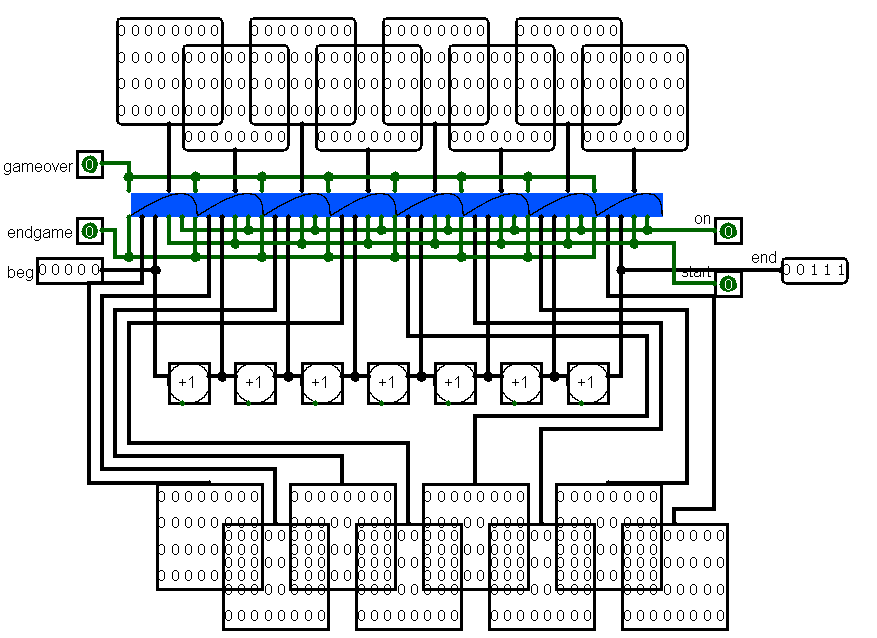


Figure 7 – screenshot of the Menumega circuit

**Pentamenu**

This circuit uses 4 Menumega circuits with 8 inputs each (32 in total). 4 inputs are also used: gameover, endgame, start, on. Null Menumega we pass zero, and sequentially receive 8 rows of each screen from each circuit. Further, depending on the inputs, we get 32 lines of the screen. If on is raised, then the “Press Start” screen is displayed, if it is raised at the same time as start, then the screen of the game itself is displayed.

The "Game over" and "You win" screens are controlled by external inputs from the Main circuit using registers and buffers. The gameover output pin from the main circuit buffers the display of the "You win" screen (the values in the go and eg tunnels rise at the same time). However, if a state occurs when the player has destroyed all opponents, then the trigger from win rises, and it blocks the losing state with the help of buffers (raises, only eg). Also in the circuit there is a five-bit control input, which is transmitted to the main circuit, and a clk output, which supplies clocks to all other circuits and the keyboard located in the Main Interface. Figure 8 shows a screenshot of the Pentamenu circuit.

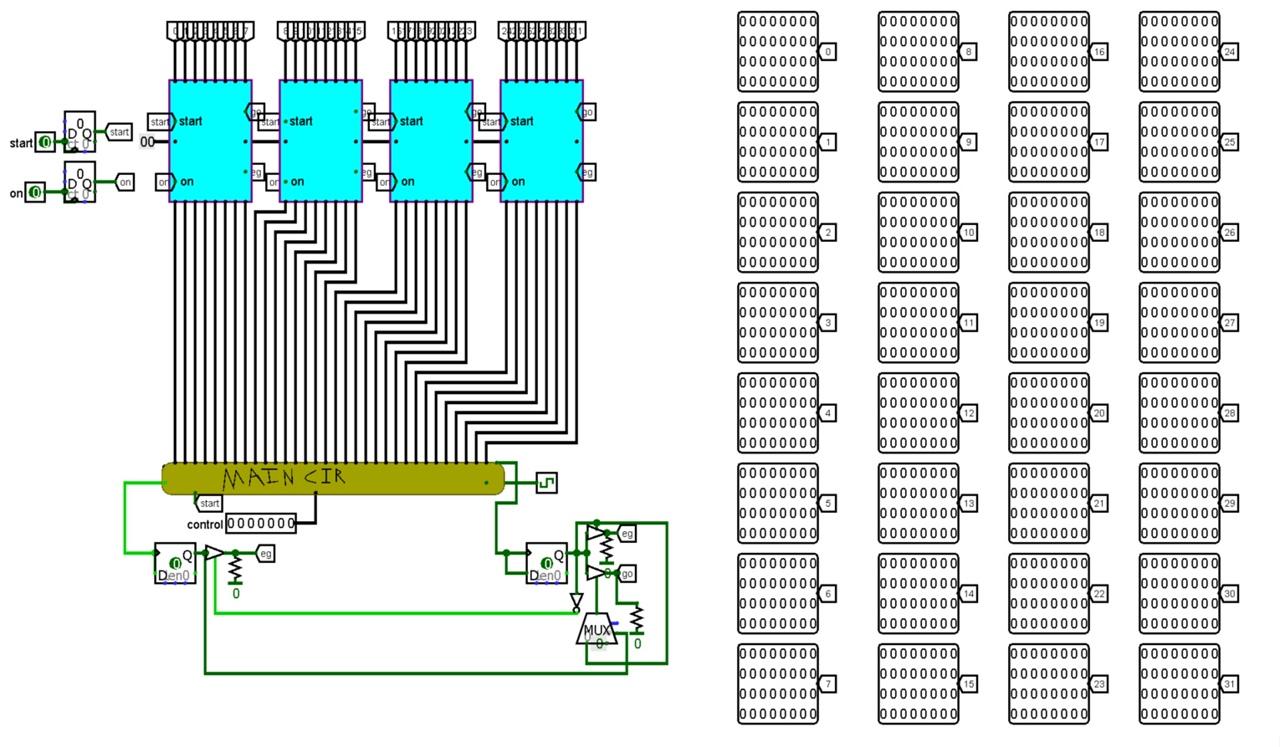


Figure 8 – screenshot of the Pentamenu circuit

**Main**

The inputs are initgame, moving and clk. Clk is served on all circuits that need clock generation, initgame is served on all circuits that require initialization. The moving input responds to three values: 61, 64 and 20. This is implemented using comparators. 61 is the key code "a" and move to the left, 64 is the key code "d" and move to the right, 20 is the key code "SPACE" and shoot. If the input value is 61 or 64, then the cannon is moved using the Player Move circuit. Main also implements interaction between all nested circuits.

32 outputs from the Shooting circuit are fed to 32 inputs to the Enemies circuit, which provides processing of hitting enemies, their shooting and movement.

There is also a boom output, which, if it hits an enemy, temporarily blocks the player from shooting to give the processor time to process the hit.

The display of the player's bullet, monsters and their bullets on the screen is connected via a logical "or" to outputs 31 to 5, which describe the lines of the screen.

4’ and 5’’ tunnels are submitted to the Walls circuit, carrying the enemy’s bullet and the player’s bullet, respectively. Two outputs come out of it, indicating a hit either from the enemy or from the player, in which case Walls blocks the operation of one of the schemes. The third OR has a triple entrance - there may still be a wall there. The second OR includes the player's bullets, or the monsters' bullets. And in the first and zero, the movement of the player and the bullets of monsters are connected.

The Enemies circuit has a win output, which means that all enemies have been destroyed, and an output from the 4th line of monster movement. if at least one bit there is equal to 1, then the gameover output in Main is immediately raised. Also, in gameover through or contains logical "and" from the bullets of monsters and the player's movement. If at least somewhere there is 1, then the player was hit, and the gameover rises. Figure 9 is a screenshot of the Main circuit.

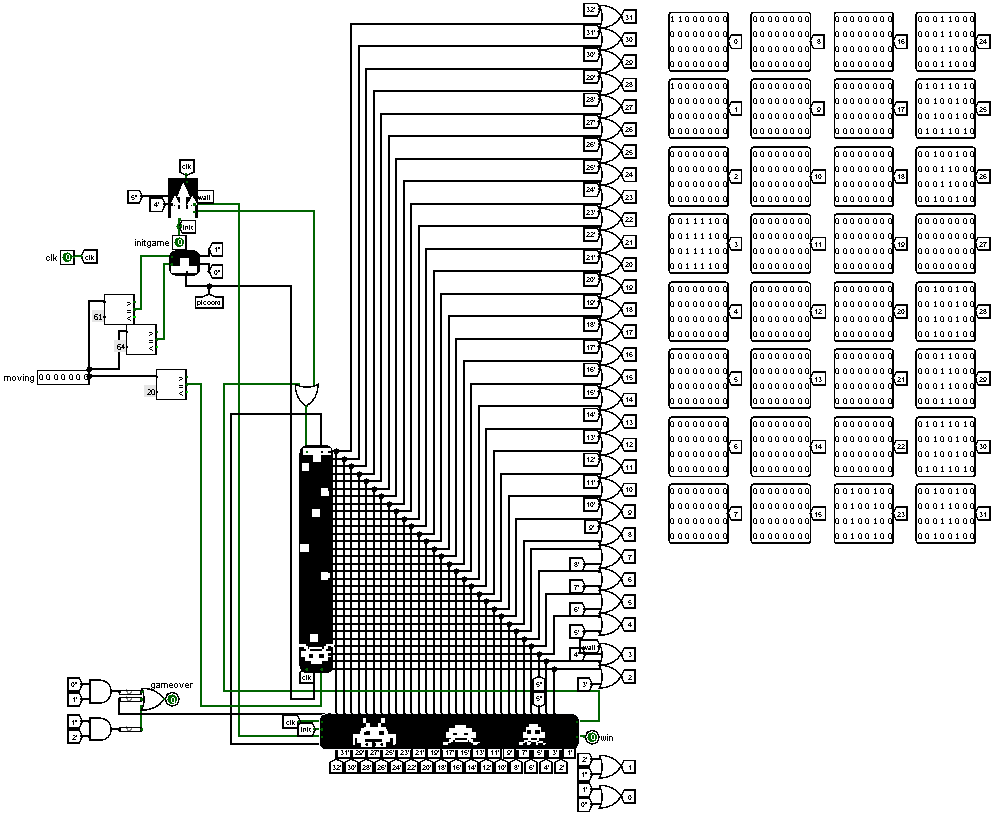


Figure 9 – screenshot of the Main circuit

**CDMEnemShoot**

This scheme takes as input a two-bit number generated by the random number generator rand and an enemy shift byte shift, with a value of -1 (shift enemies to the left) or 1 (shift enemies to the right). The clk input specifies the frequency of the processor and memory elements. Based on the input data rand and shift CdM-8 generates the number of the matrix column where the bullet will be placed (res). The output pin outputs res.

The code is looped because it must be executed several times. We need to stop the processor and start it again. To do this, a demultiplexer is used, which powers the clock generator when the PC state trigger is off, equal to 1 (the code byte of the beginning of the main subroutine). This trigger is controlled by a manual start status trigger, which is activated when the En input contact is triggered and deactivated when the program ends. The Ready output pin tells the environment if the program has finished processing. Figure 10 is a screenshot of the CDMEnemShoot circuit.

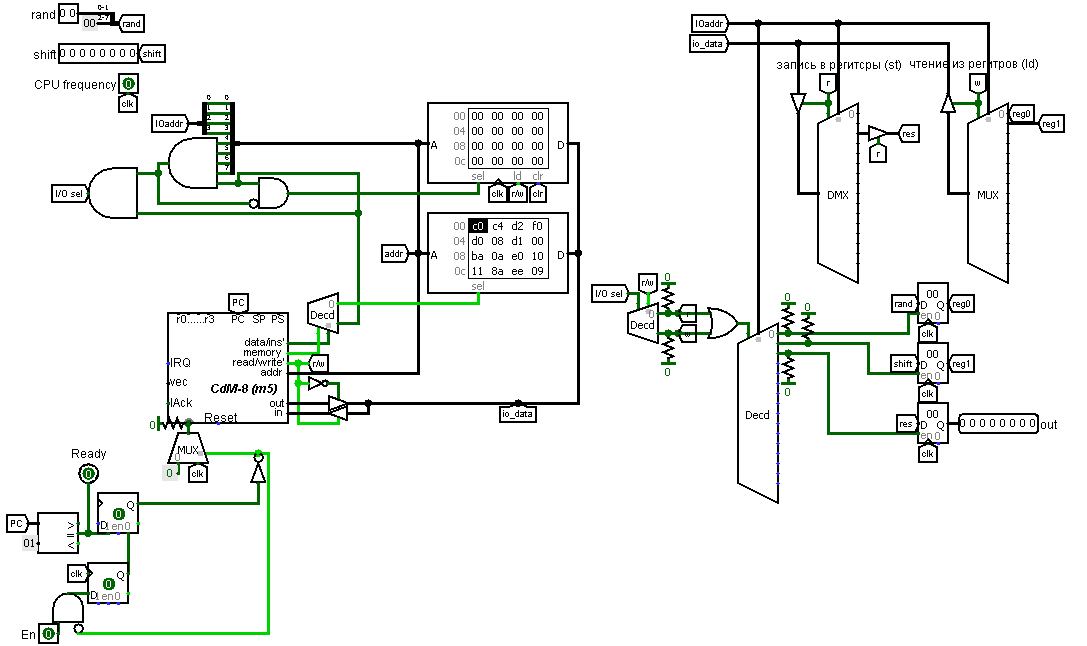


Figure 10 – screenshot of the CDMEnemShoot circuit

**Inc 5bit**

In this circuit, increment is implemented using two gates: a five-bit “and” and an exclusive “or”. The value of the five-bit input x is fed to xor and to "and". The output from the "and" gate is split by a splitter, the zero bit of the number is replaced by one, and the highest bit is transmitted to the Cout output. Figure 11 shows a screenshot of the Inc 5bit circuit.

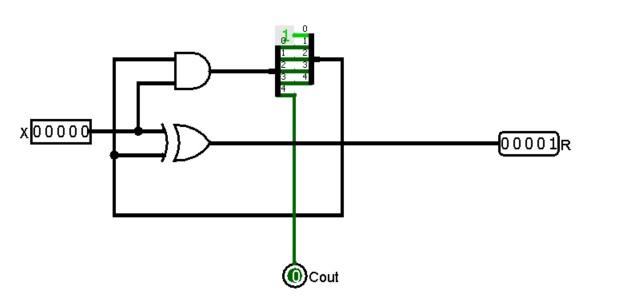


Figure 11 – screenshot of the Inc 5bit circuit

**Player Move**

The display of the player's movement is implemented using a decoder and a counter. The counter value is incremented or decremented depending on the l and r inputs. Then it is transmitted to the decoder, which feeds one to the bit number that came to the input. After that, this bit is shifted by 1 to the left and to the right from the initial position and using the “or” operation between these bits, the part of the cannon displayed on the screen is obtained. The result is sent to output 0. The value from the decoder without shifts is fed to output 1 to get a full image of the gun.

Also, in the circuit there is a counter output. The current value from the counter is fed to it. Figure 12 shows a screenshot of the Player Move circuit.

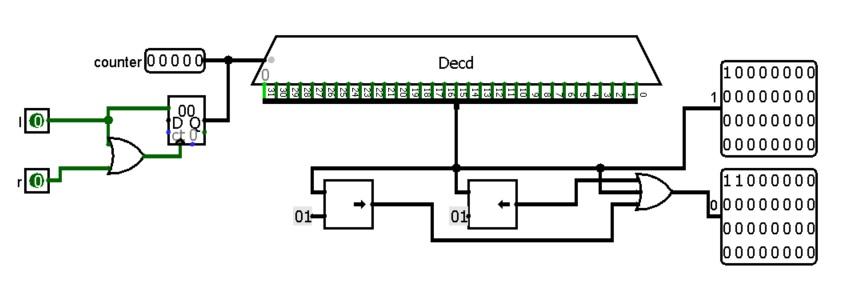


Figure 12 – screenshot of the Player Move circuit

**Transpon Field**

The scheme implements the transposition of the playing field. It has 32 32-bit inputs and 32 32-bit outputs. From the tunnels to inputs from 0 to 31, using a splitter, the zero bit is first isolated and fed to output 0. Then the first bit is selected, and the outputs from the splitters are combined and fed to output 1. Similar operations are performed with each bit. Figure 13 is a screenshot of the Transpon Field circuit.Изображение выглядит как стол

Автоматически созданное описание

Figure 13 – screenshot of the Transpon Field circuit

**Walls**

Input pins: init indicates if the game has started, clk is the clock generator, 2 bull inputs for various bullets (player and monsters) with 32 bits. Passing the bull inputs through the circuit to determine a hit will result in a number where a bit of one indicates the location of the hit. Also, on hit, a value is loaded into a 32-bit register showing the location of the hit. This only happens if the output from enemy or player bullet hit detection is non-zero. If there was a hit of the monster in the wall, then the trigger rises with an exit to BOOMmon!. If there was a hit by a player or an alien, then the trigger rises with an exit to BOOM!.

The BOOMmon! output is used only to track if there was an enemy hit, to disable the shooting of enemies for the duration of the processing. There is also a 32-bit output to the value of the walls themselves (their state is stored using a register). The processor control is similar to the CDMEnemShoot circuit, but the control of the Manual Trigger depends on the states of BOOM! and start game. BOOM State! is stored in the trigger that fires from the trigger contact of the Collision circuit and is reset to zero when the state is DONE. The DONE state is stored in a register that is set to zero when DONE is triggered and returns one when the 3rd wall hit count is 2. Figures 14 and 15 show screenshots of parts of the Walls circuit.

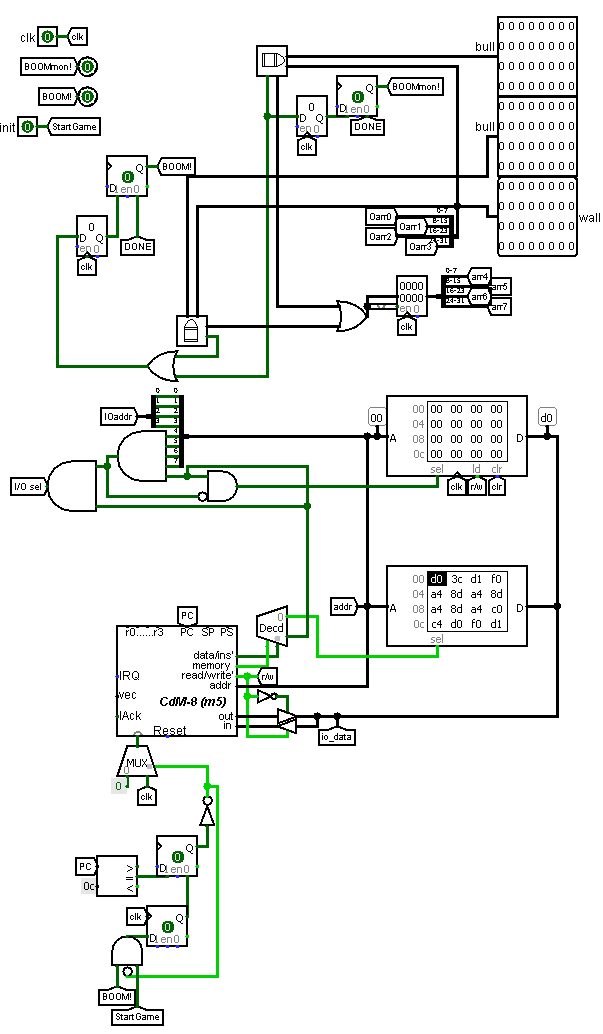


Figure 14 – screenshots of part of the Walls circuit

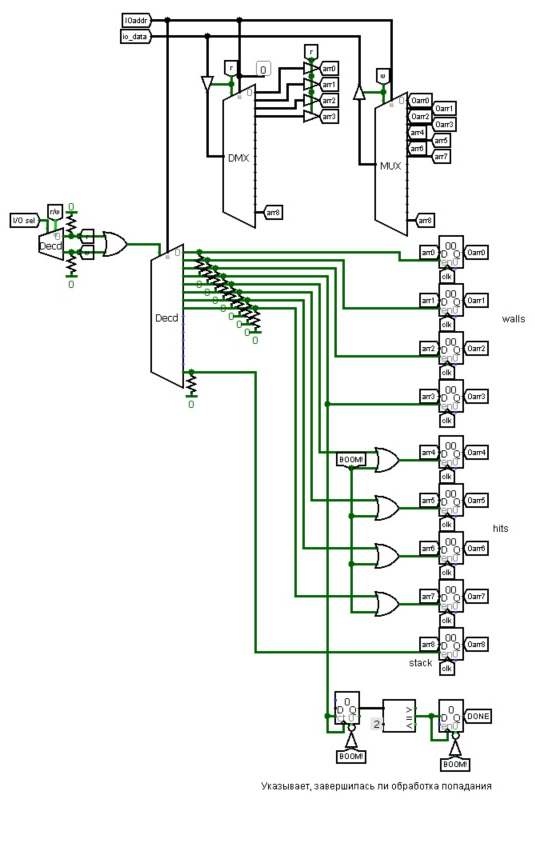


Figure 15 – screenshots of part of the Walls circuit

**Collision**

The coordinates of two objects on the screen are the same when the same bit in different lines is equal to one. This circuit implements a collision check by performing an “and” operation between two thirty-two-bit inputs A and B. The result of the operation is fed to the OUT output. Also on the circuit there is a Trigger output, to which the value from the “and” gate is passed through the Or32 circuit. Figure 16 is a screenshot of the Collision circuit.

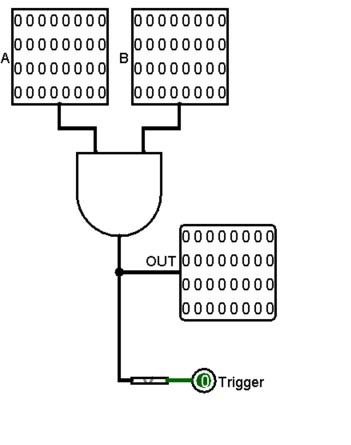


Figure 16 – screenshot of the Collision circuit

**Or 32**

This circuit is needed only to reduce the size of the 32 input OR. It contains one 32-bit value, which is split by the input splitter into 32 one-bit values that pass through the “or” gate and are fed to the output. Figure 17 is a screenshot of the Or 32 circuit.

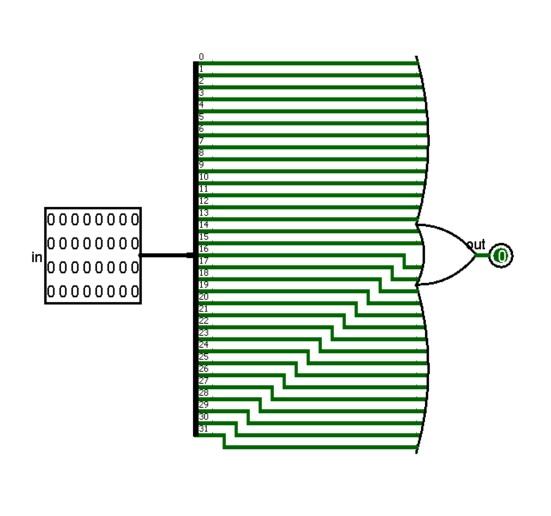


Figure 17 – a screenshot of the Or 32 circuit

**Shooting**

There are 4 inputs in the circuit: fire, reset, coord and clk. They are used to control the player's shooting. If the fire button was pressed, then the fire input is activated, which is passed to the D-trigger if reset is not equal to one. The reset input is needed to disable firing while the game object hit is being processed. The input coord is fed to the input to control the output from the demultiplexer. A column of bullets is fed into the column where the player is. The outputs from the demultiplexer are connected via the Transpon Field circuit to work with columns. After raising fire, the D-trigger rises, the output of which, through OR with an inverted input, is applied to reset the register. Thus, the register is reset to zero if the fire output is not raised, or the reset input is raised. The register increments its value each time the clk input rises. With the help of the decoder, a bit is raised, which each time is shifted by 1 due to the increase in the value of the counter. The outputs from the decoder are collected into a 32-bit number, which is transmitted to the input to the demultiplexer, which transmits it to one of the 29 outputs. Also on the circuit there is a five-bit output for use by other circuits. Figure 18 shows a screenshot of the Shooting circuit.

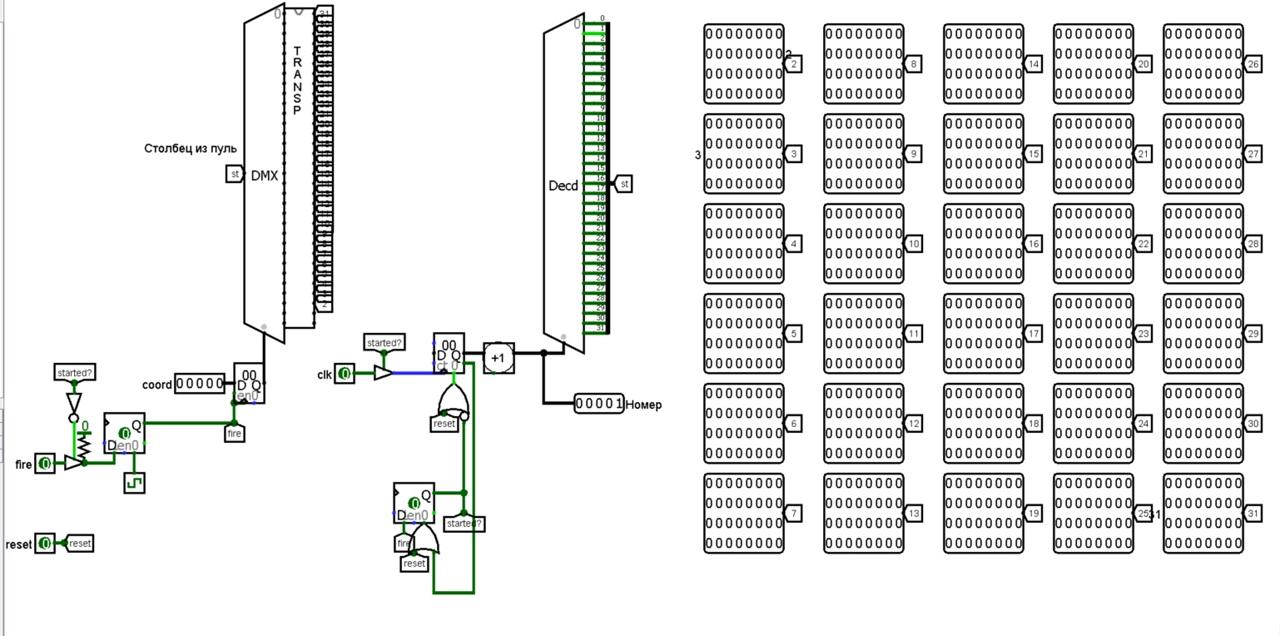


Figure 18 – screenshot of the Shooting circuit

**Shifts**

This scheme implements the motion control of the aliens. For this, a counter is used that takes values from 0 to 5: 1 - shift left, 2 and 5 - shift down, 3 - zero shift left, 4 - shift right, 6 - zero shift right.

Shifts are calculated by passing 9 input lines through ShifterLeft or ShifterRight, controlled by counters, which store the amount of shifts, and ShifterDown, which shifts the result down by the value of its counter.

The output is the new layer of shifted enemies (2e - 31e), the shift value shift (left (-1) right (1) or none (0)) and the 5-bit down shift value height. Figure 19 is a screenshot of the Shifts circuit.

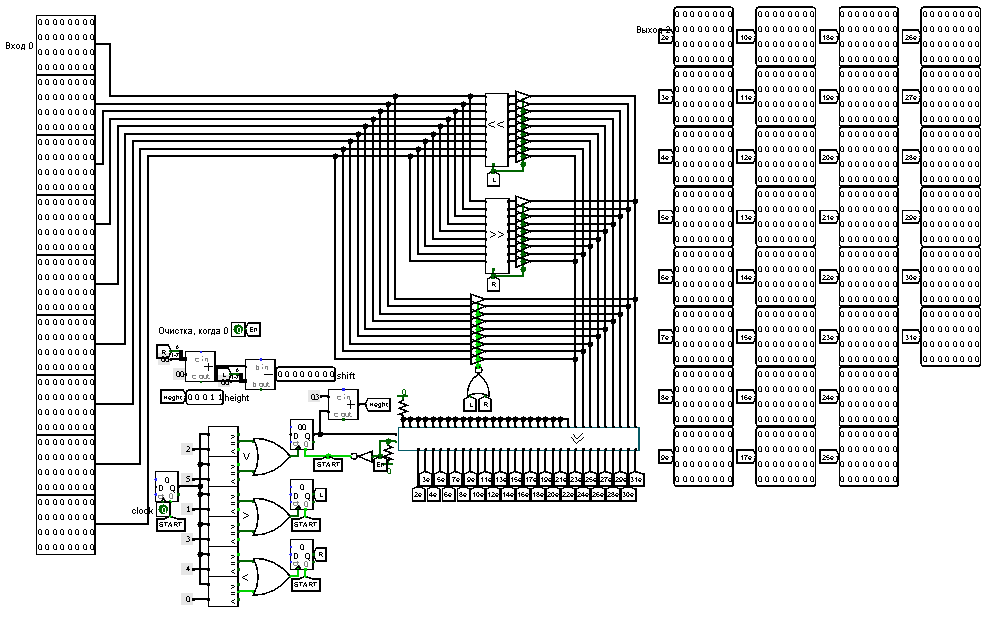


Figure 19 – screenshot of the Shifts circuit

**ShifterLeft and ShifterRight**

8 32-bit inputs are used to implement the movement of opponents left and right. These inputs are passed through left and right shifters only if the 1-bit input was high. This is achieved using controlled buffers - if this input is low, the values will not pass through the shifters. The input shifter receives one of the 32-bit numbers, and shifts it only by 1, this is implemented using a splitter, to the 4 highest inputs of which the ground is connected. Thus, by turning them on and off in turn, the effect of moving left and right was achieved. Figures 20 and 21 show screenshots of the ShifterLeft and ShifterRight circuits.

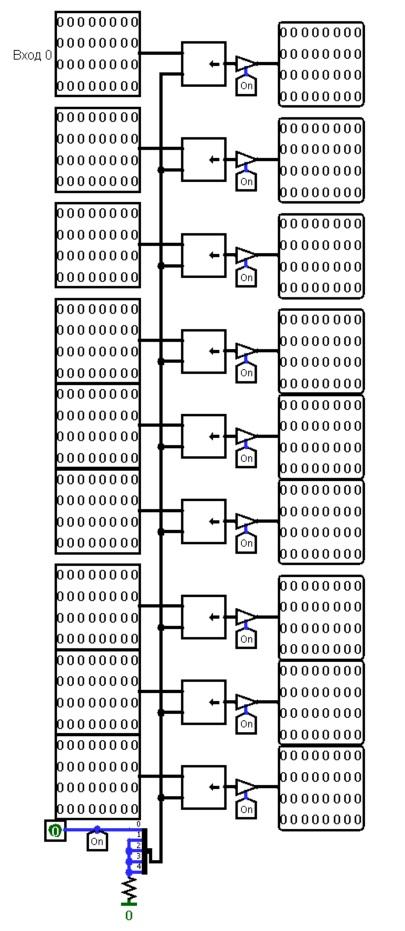


Figure 20 – screenshot of the ShifterLeft circuit

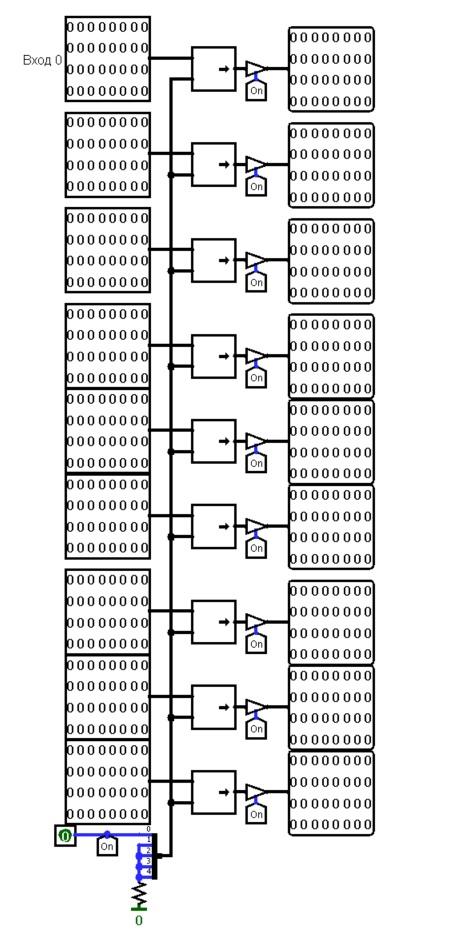


Figure 21 – screenshot of the ShifterRight circuit

**ShifterDown**

This circuit shifts the layer (pins 0-31) down by the value of the height pin, provided the En pin is on.

Layer shifting is implemented through 30 multiplexers (for each input pin). Each has its own unique value of the select input equal to the addition of the unit shift value n times for the n-th multiplexer. The disabled output of the multiplexers has a floating value, which allows multiple values ​​to be applied to one pin without the use of control buffers.

The result of the shift is applied to pins 0' - 29' (30' and 31' are not used, but can be added). Figure 22 and 23 are screenshots of parts of the ShifterDown circuit.

Изображение выглядит как текст, устройство

Автоматически созданное описание

Figure 22 – screenshot of part of the ShifterDown circuit

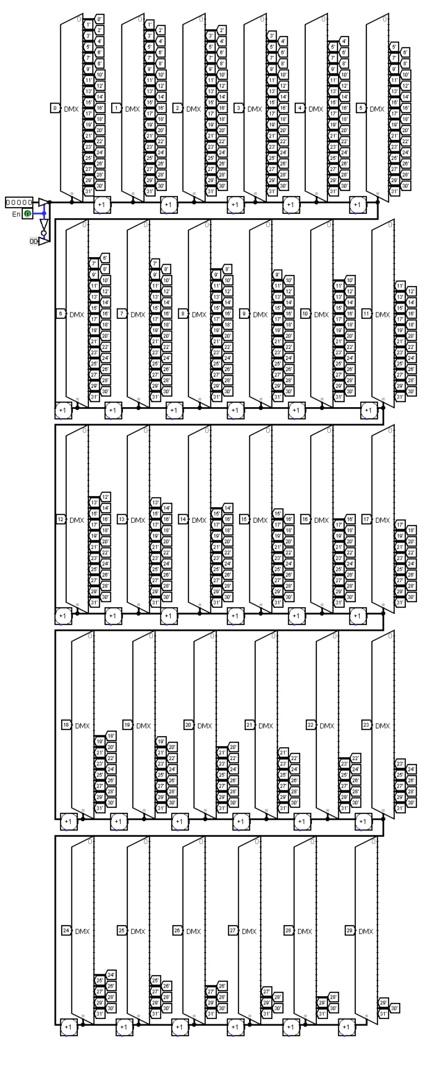


Figure 23 – screenshot of part of the ShifterDown circuit

**Enemies**

This circuit is responsible for the behavior of enemies, their appearance, destruction, movement, shifts and shooting. There are input contacts: initgame says if the game has started, clk is the clock generator, wall says if the walls are being processed now (this is necessary to disable the shooting of enemies for the duration of processing); Num is coordinate of the bullet line and the 30 values of the player's bullet layer (2bul, 3bul,.. 31bul) - for the Collision check. Figure 24 shows a part of the Enemies circuit.

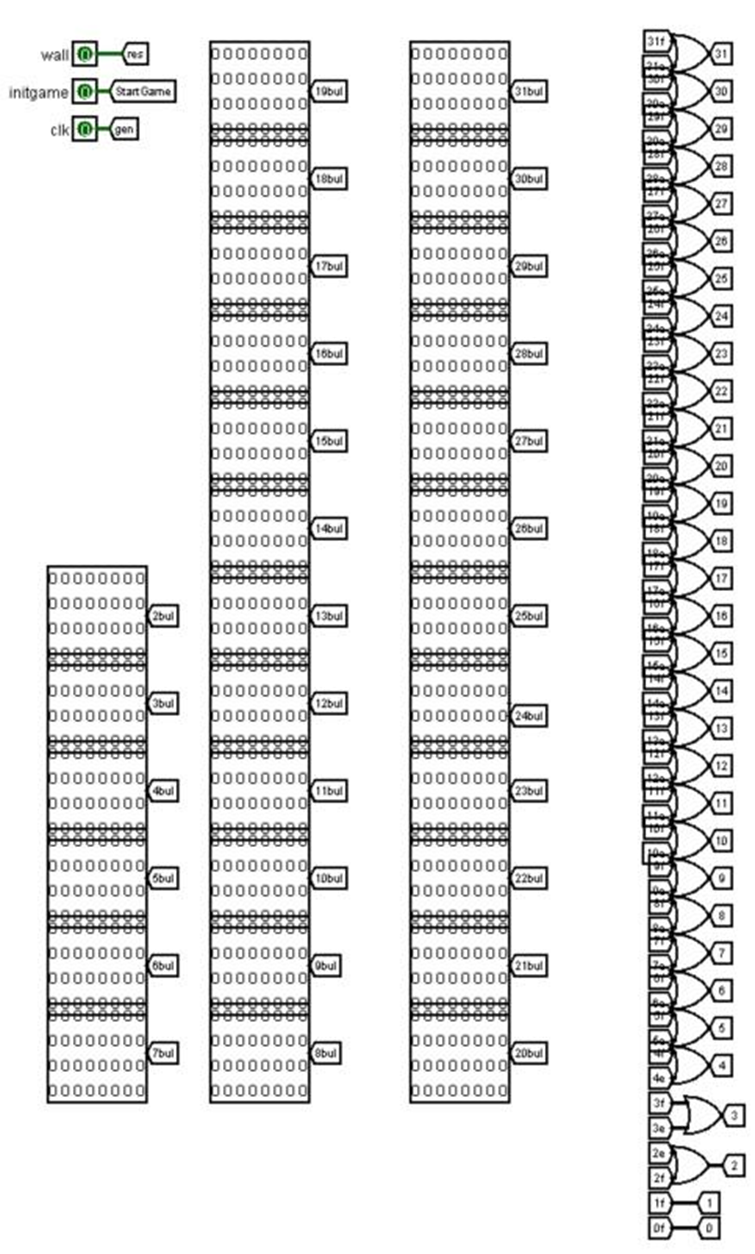


Figure 24 – screenshot of a part of the Enemies circuit

The processor in the diagram is responsible for initializing and turning off enemies when hit. Processor control exactly follows the Walls circuit. Figure 25 shows the Processor control part of the Enemies circuit.

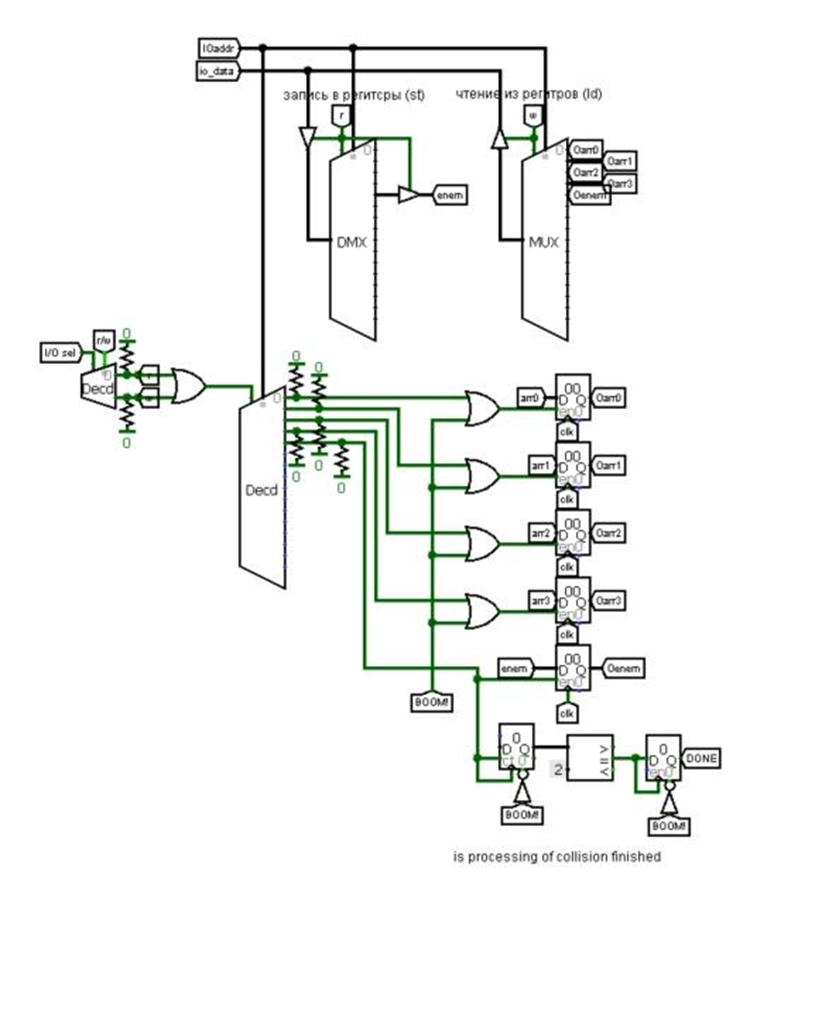


Figure 25 – screenshot of the Processor control part of the Enemies circuit

The state of the enemies is stored in a 8-bit register (from which the Oenem tunnel departs). 1 - there is an enemy, 0 – there is no.

The display of enemies is set as constants. There are 2 constants for 2 types of enemies. These constants are added to registers controlled by the bits of the enemy status byte and are arranged through splitters on 8 lines of 4 enemies each. These rows are passed under the control of the Shifts circuit. The Shifts circuit gives a matrix layer with enemies; the height at which the enemies dropped and the parameter for shifting them to the left or right info. Figure 26 is a screenshot of a part of the Enemies circuit.

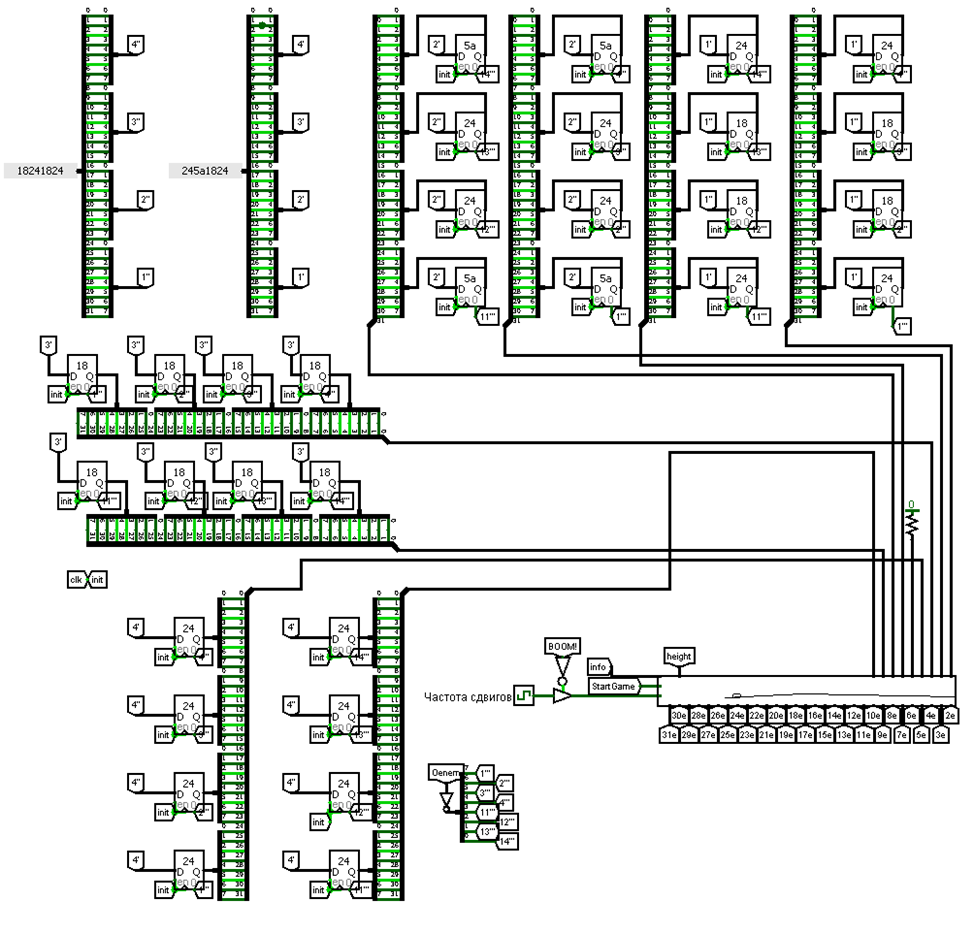


Figure 26 – screenshot of a part of the Enemies circuit

The Collision check part compares 2 layers: the player's bullet layer and the enemy layer. The Num row is taken from the enemy layer, and the Num + 1 row is taken from the player's bullet layer. These rows are sent to the Collision circuit, which will return a 32-bit string of zeros with a single one - where the hit was, and the hit trigger. The 32-bit string will be split into 4 8-bit strings and sent into registers for processing by processor. Figure 27 is the Collision check part of the Enemies circuit.

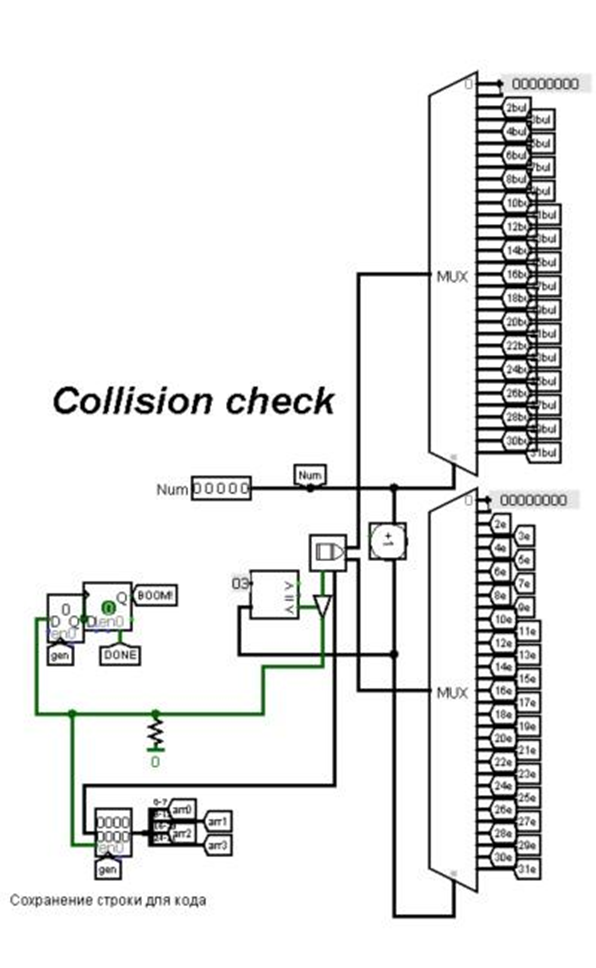


Figure 27 – screenshot of the Collision check part of the Enemies circuit

The Enemy shooting part controls the shooting of the top row of enemies. The random generator generates a number from 0 to 3, this value is added to the register and will lie there until the enemy's shot ends. If the enemy with the generated number exists (the multiplexer checks the enemy existence byte) the value from the register is sent to the CDMEnemShoot circuit, along with information about shifts info from the Shifts circuit. CDMEnemShoot generates a coordinate where 1 should be placed to shoot. 1 is subtracted from the same coordinate or not subtracted depending on the random generator (enemies have 2 cannons nearby). The result is passed to the demultiplexer, which generates a 32-bit string of zeros and adds a one to it, provided that the previous bullet has ceased to be. This string is placed in the register which will be reset to zero when the bullet is gone. Meanwhile, the circuit is already preparing a new coordinate for the next bullet. The value from the register is given to the 32-bit demultiplexer. Its Select input was given a counter value, to which was added the height value (that was puted in the register when the counter value was 0) which the enemies have wented down. Bullet shutdown occurs when the counter returns to 0 or when the bullet hits a wall. Figure 28 is the enemy shooting part of the Enemies circuit.

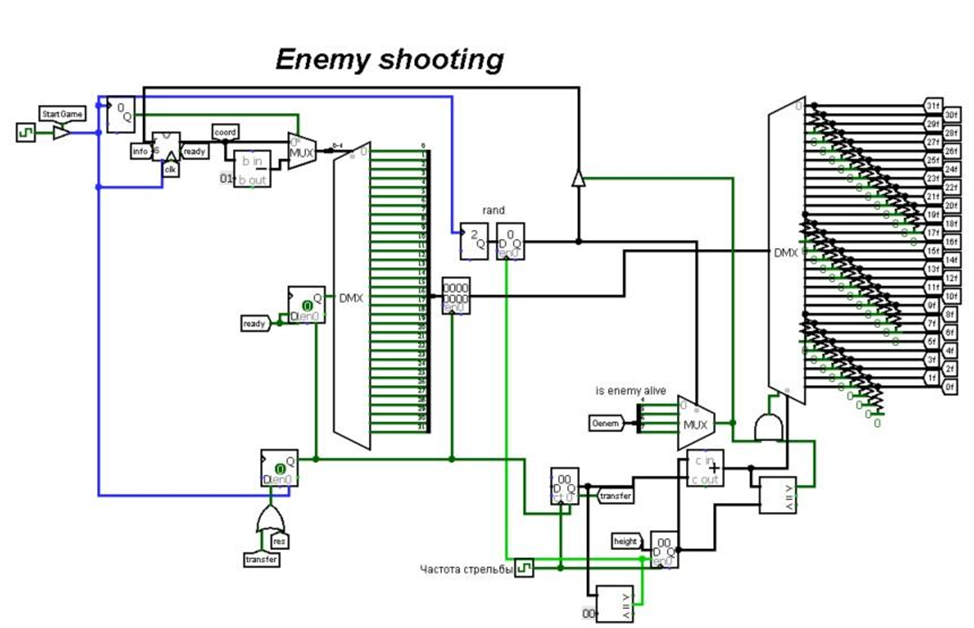


Figure 28 – screenshot of the enemy shooting part of the Enemies circuit

As soon as there is no any enemy left, the output contact win lights up. The enemy layer and the enemy bullet layer pass through the OR gate and the resulting layer sets to the output pins (0 - 31). Another output pin 4e give the 5th line of the layer of enemies from below. This need by the Main circuit for the Gameover trigger. Figure 29 shows a screenshot of a path of the Enemies circuit.

Изображение выглядит как стол

Автоматически созданное описание

Figure 29 – screenshot of path of the Enemies circuit

4 SOFTWARE

The software part of our project was made using the assembly language of the CdM-8 processor in the CocoIDE integrated development environment, created specifically for developing the code executed by this processor. Let’s consider the algorithms.

**Wall destruction**

This is an algorithm that implements the initialization and destruction of protective walls when monster bullets hit them. The input data is two four-byte arrays: a string with the current state of the walls and a null string with one hit bit equal to 1. The output data is a string with the state of the walls after a bullet hit or miss.

The first step initializes the registers located at addresses F0-F3 with the value 3C (00111100 in binary). This constant is the four-pixel wall displayed on the screen. At the second step, the input data is loaded into the general-purpose registers, the counter is initialized. Then, in a loop, while the counter is greater than zero, an exclusive "or" operation is performed between the corresponding bytes of the input arrays, and the result is stored in the register into which the walls were loaded. The result is a bit string with states after hitting or missing bullets. At the end of the loop, there is a transition to the second step of the algorithm. Figure 30 shows a block diagram of the algorithm for initializing and destroying walls.

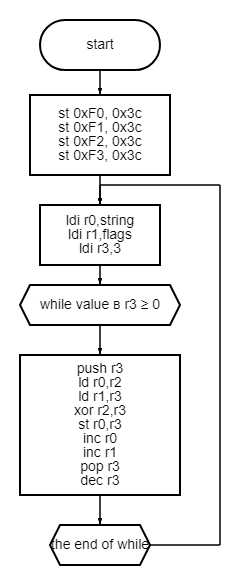


Figure 30 – a block diagram of the algorithm for initializing and destroying walls

**Monsters shooting**

This is an algorithm that implements the calculation of the coordinate (column number) in which the monster will shoot. The input data is two one-byte numbers: the number of the shooting alien, (from 0 to 3) and the offset value of the monster (-1 or 1). The output data is a one-byte column number.

The first step is to load the constants 8 and 0 and the alien number to be used as a counter into the general-purpose registers. Then the loop will calculate a value equal to the number of the shooting monster, multiplied by 8. After that, the offset value and the constant 4 are added to it, and the result (the resulting column number) is stored in the register. At the end of the loop, there is a transition to the first step of the algorithm. Figure 31 shows a block diagram of the algorithm for calculating the number of the column in which the shot will be fired.

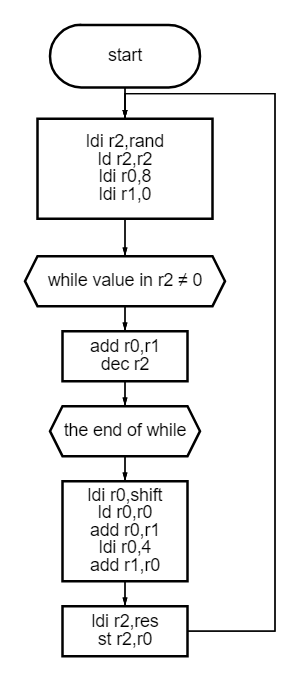


Figure 31 – a block diagram of the algorithm for calculating the number of the column in which the shot will be fired

**Hit**

this is an algorithm that implements the initialization of the state of the aliens and the processing of the player's bullet hitting them. The input data is a byte that stores the state of the aliens (if the bit is 1, then the monster is alive, if 0, then not) and a null string with one hit bit equal to 1. The output data is a byte that stores the new states of the enemies.

At the first step, the monster status byte is initialized with the value FF (11111111 in binary). The second step loads the zero string with one hit bit equal to 1 and the constant 0 into the general-purpose registers, the counter is initialized. Then the loop looks for a non-zero byte in the string and calculates the number of the alien that was hit. After that, by bitwise shifts to the left, a one-byte mask is calculated, where the only bit equal to 1 has an index equal to the number of the alien that is in the bottom row. The next step is to compare this mask and the byte that stores the states of the enemies by performing a bitwise "and" operation. If the result is zero, then there is no monster in the bottom row, so 4 more bitwise shifts to the left are performed to check the top row. Next, an exclusive “or” operation is performed between the new mask and the status byte, and the result is stored in a register. There is a transition to the second step of the algorithm. Figures 32 and 33 show parts of the block diagram of the algorithm for initializing the state of aliens and processing hitting them.

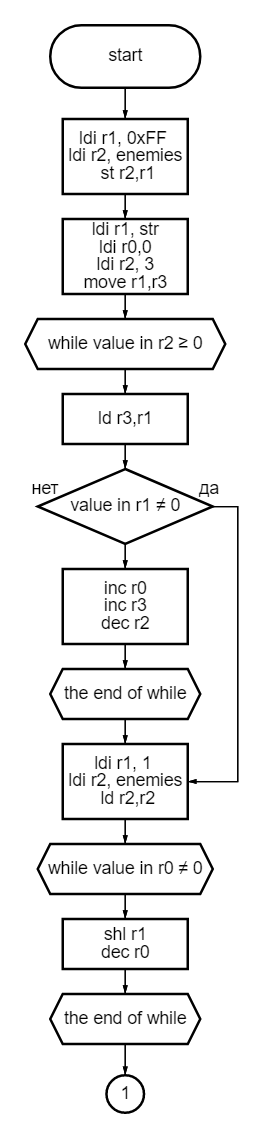


Figure 32 – the first part of the block diagram of the algorithm for initializing the state of aliens and processing hitting them

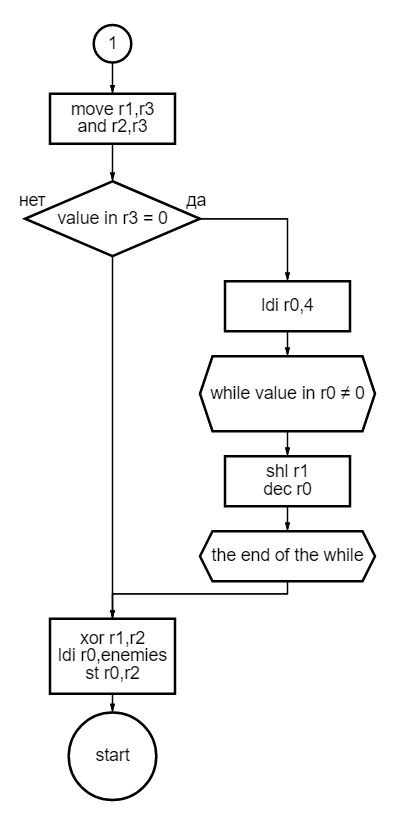


Figure 33 – the second part of the block diagram of the algorithm for initializing the state of aliens and processing hitting them

Thus, the developed game is the result of the interconnected work of the hardware and software parts of the project described earlier.

CONCLUSION

As a result of the work done, we managed to create the game "Rush space" in the Shoot 'em up genre (see Appendix A for the user manual). It is implemented using electronic circuits in which we have used a CdM-8 processor that executes the codes we have written. The problems set before us at the beginning of the project were successfully completed. We were able to implement all the indicated functional requirements: movement and shooting of the player and opponents, control from the keyboard buttons, the possibility of winning and losing in the game. Creating the project, we got knowledge in the field of creating electrical circuits, working with a processor built on the Harvard architecture and its programming, writing project documentation and experience in working in a team.

REFERENCES

1. Computing platforms / A.Shafarenko, S.P.Hunt. – 2015.

APPENDIX

APPENDIX A

**User manual**

When the game is opened, the user sees an electrical circuit stylized as an arcade machine. It has two buttons and a keyboard with which the player can control the game. Figure A.1 shows a screenshot of the initial screen of the game.

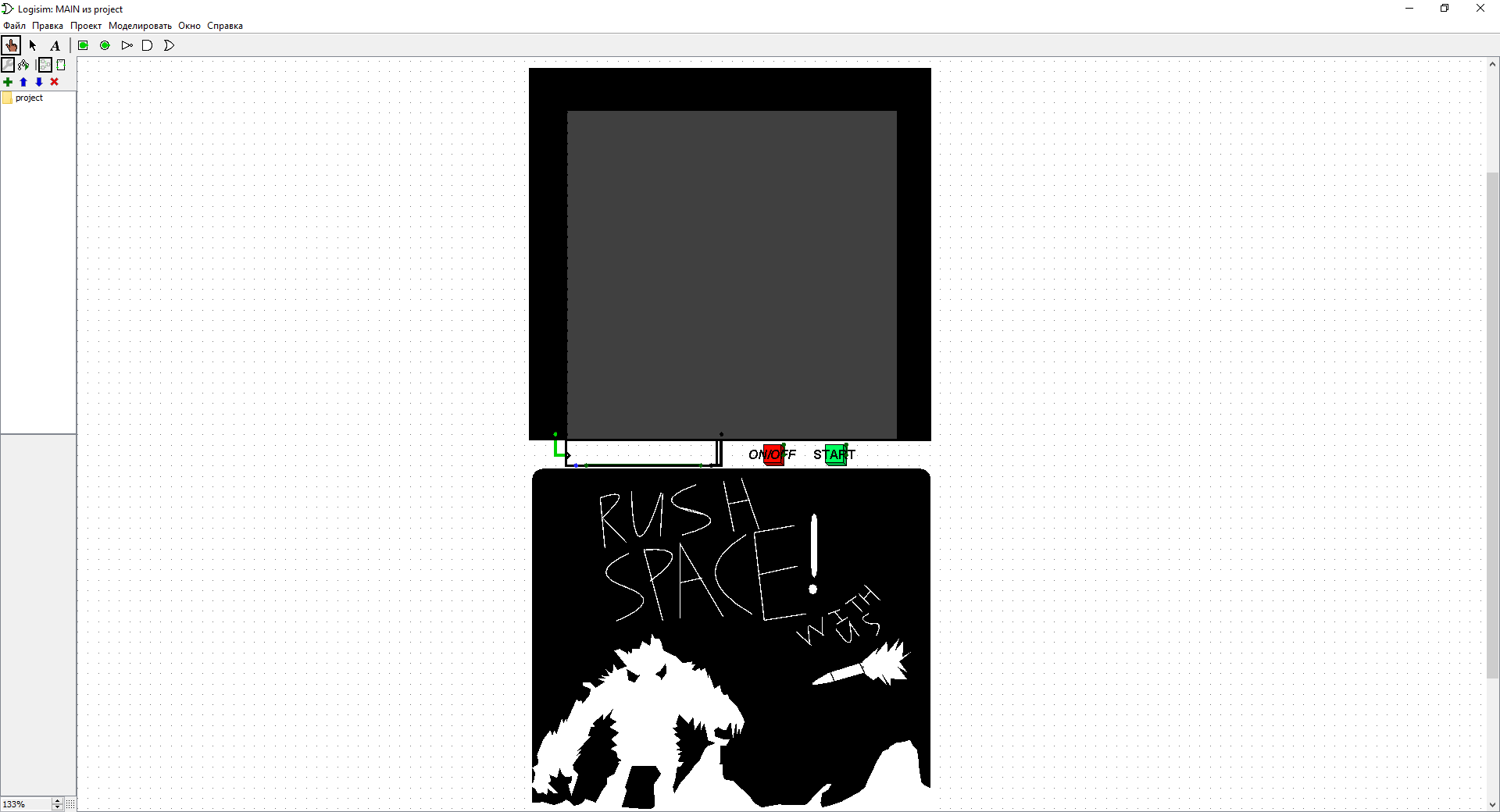


Figure A.1 – screenshot of the game's initial screen

To turn on the game, the user needs to press the "ON / OFF" button. After that, the player will see a message on the screen that he needs to press the “START” button to start the gameplay. Figure A.2 is a screenshot of the screen with this message.

Изображение выглядит как текст, часы

Автоматически созданное описание

Figure A.2 – screenshot of the screen asking user to press the "START" button

When user click on the button that starts the game, the field is displayed on the screen and the game process begins. The player controls the cannon, located at the bottom of the screen, in a horizontal plane using the keyboard, namely the keys “w”, “a”, “s”, “d”. Also on the field, the user sees 4 static walls, and 8 monsters moving uncontrollably horizontally and vertically. Figure A.3 shows a screenshot of the playing field at the start of the game.

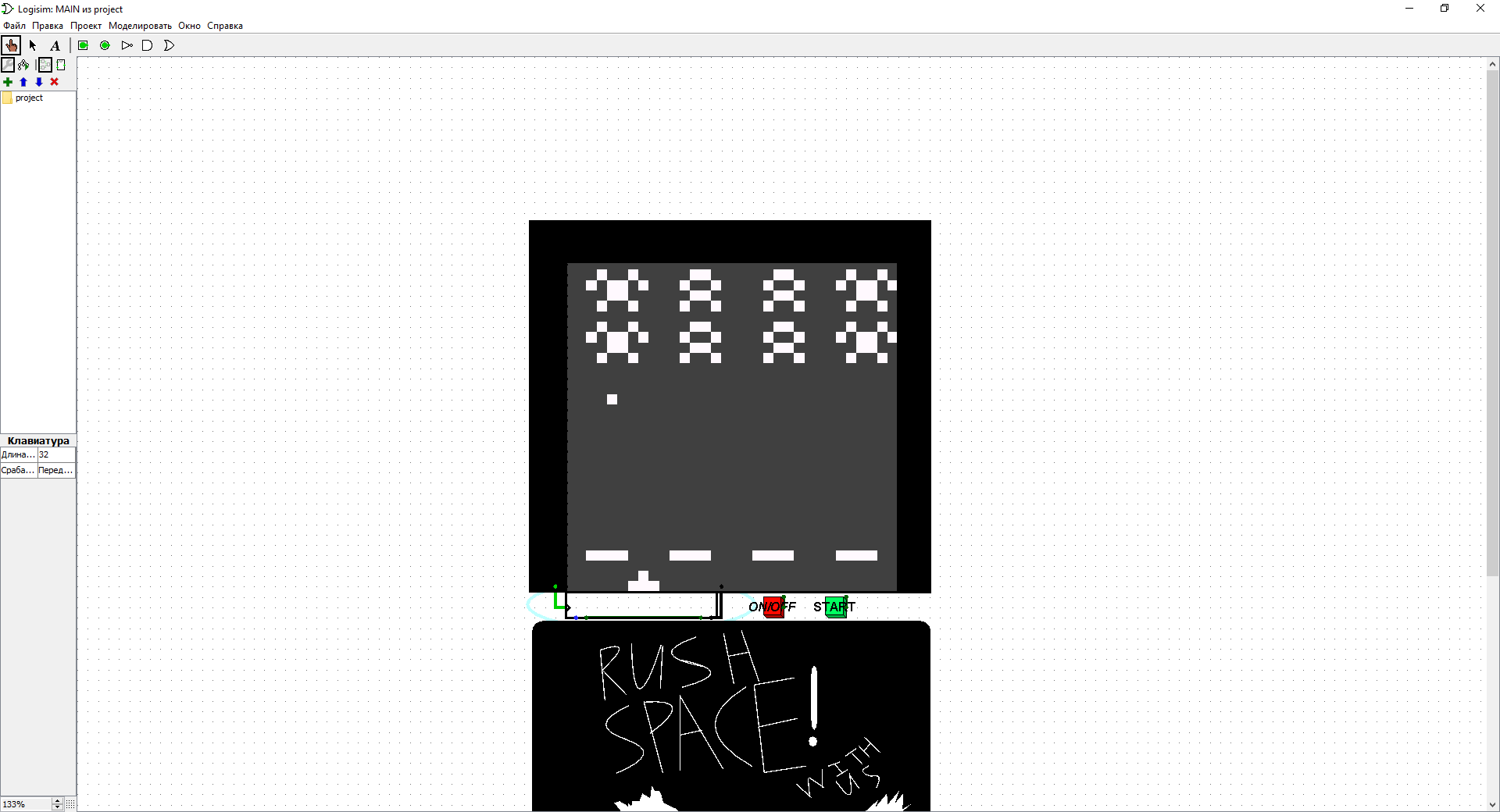


Figure A.3 – screenshot of the playing field at the start of the game

The user can also shoot using the "SPACE" key on the keyboard. When pressed, a bullet flies out of the cannon, flying vertically upwards until it hits an alien, or before it goes out of the playing field. Figure A.4 shows a screenshot of the playing field at the time of the shot.

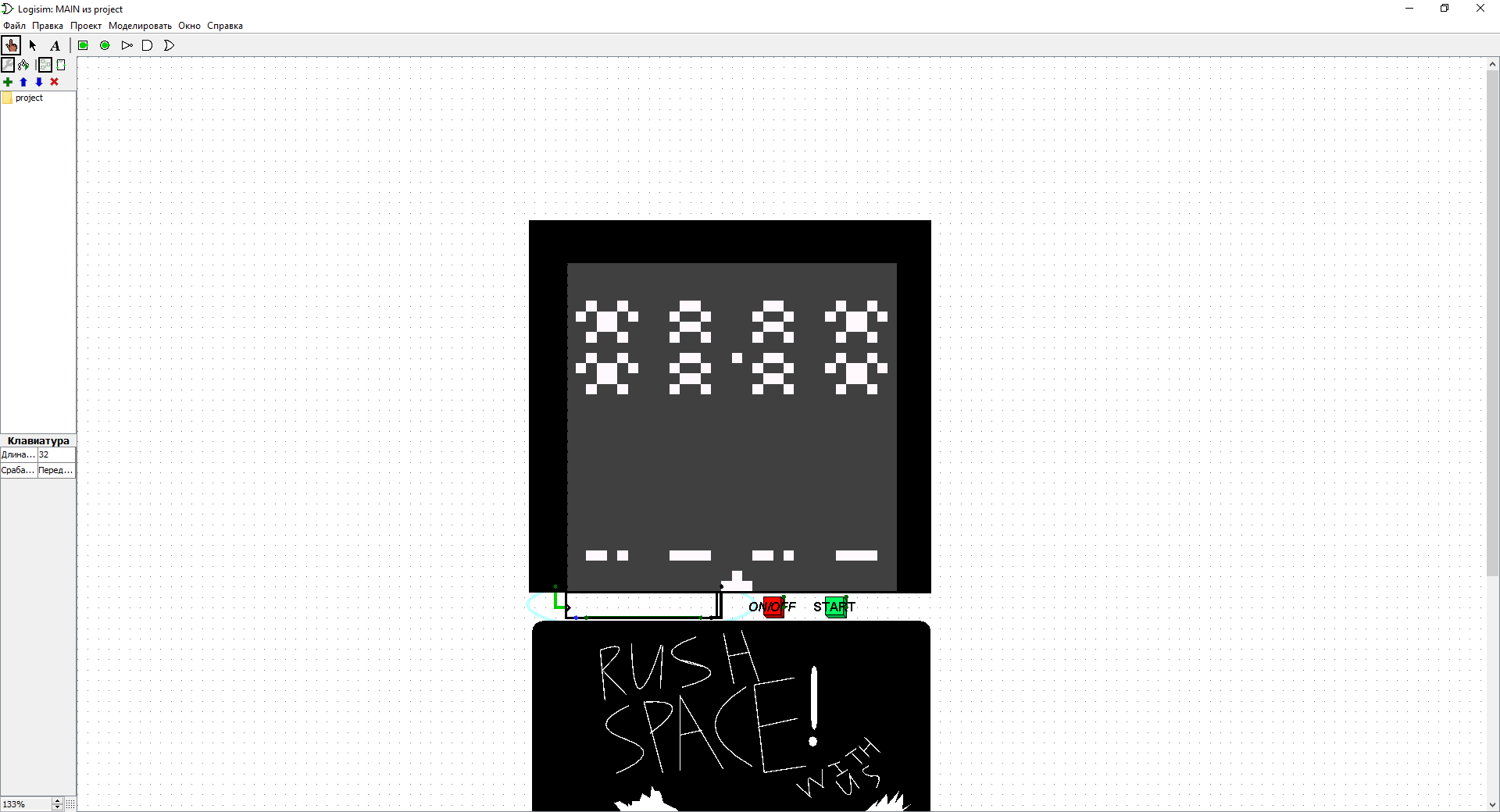


Figure A.4 – screenshot of the playing field at the time of the player’s shot

Monsters also shoot. Every few seconds, a random alien fires a bullet that travels vertically downwards until it hits the player, either against the wall or before it goes out of bounds of the playing field. The wall protecting the player is partially destroyed when a bullet hits it. Each "protection" can withstand 4 hits before it completely breaks down. It is important to note that the wall can be attacked not only by aliens, but also by the player himself. Figure A.5 is a screenshot of the playing field with partially destroyed walls.

Изображение выглядит как текст, часы, снимок экрана

Автоматически созданное описание

Figure A.5 – screenshot of the playing field with partially destroyed walls

When an alien bullet hits the cannon, the user loses and a game over message shows on the screen. Figure A.6 is a screenshot of the game over message.

Изображение выглядит как текст, часы, снимок экрана

Автоматически созданное описание

Figure A.6– screenshot of the screen with a message about the game over

The player can lose under another condition. Every few seconds, all the monsters move closer to the cannon. If at least one alien can survive and get the walls, then the game will be end and an endgame message will show on the screen. Figure A.7 is a screenshot of the playing field in a state close to losing.

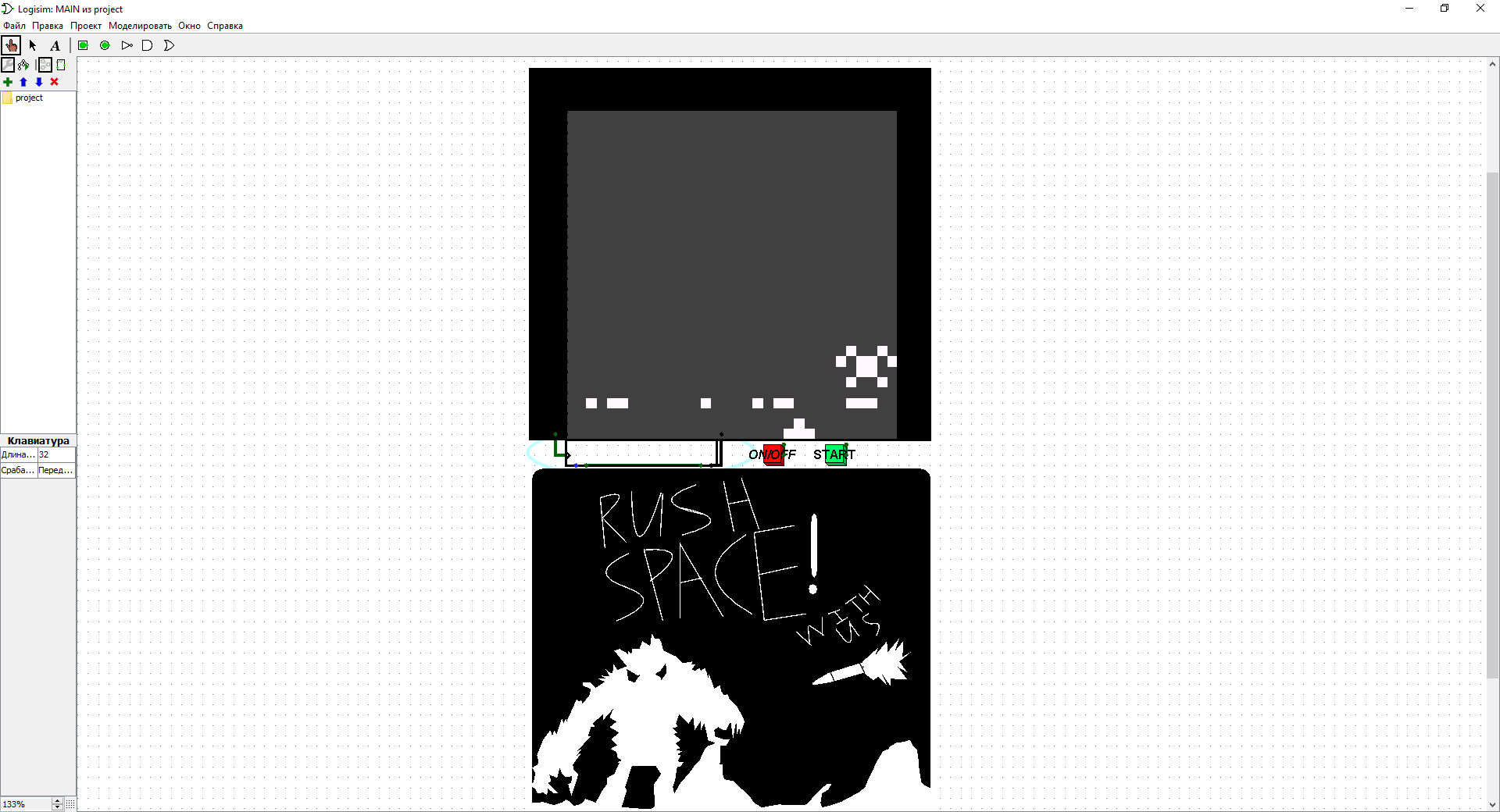


Figure A.7 – screenshot of the playing field in a state close to losing.

The player must destroy all the monsters to win. Figure A.8 shows a screenshot of the playing field after hitting most aliens.

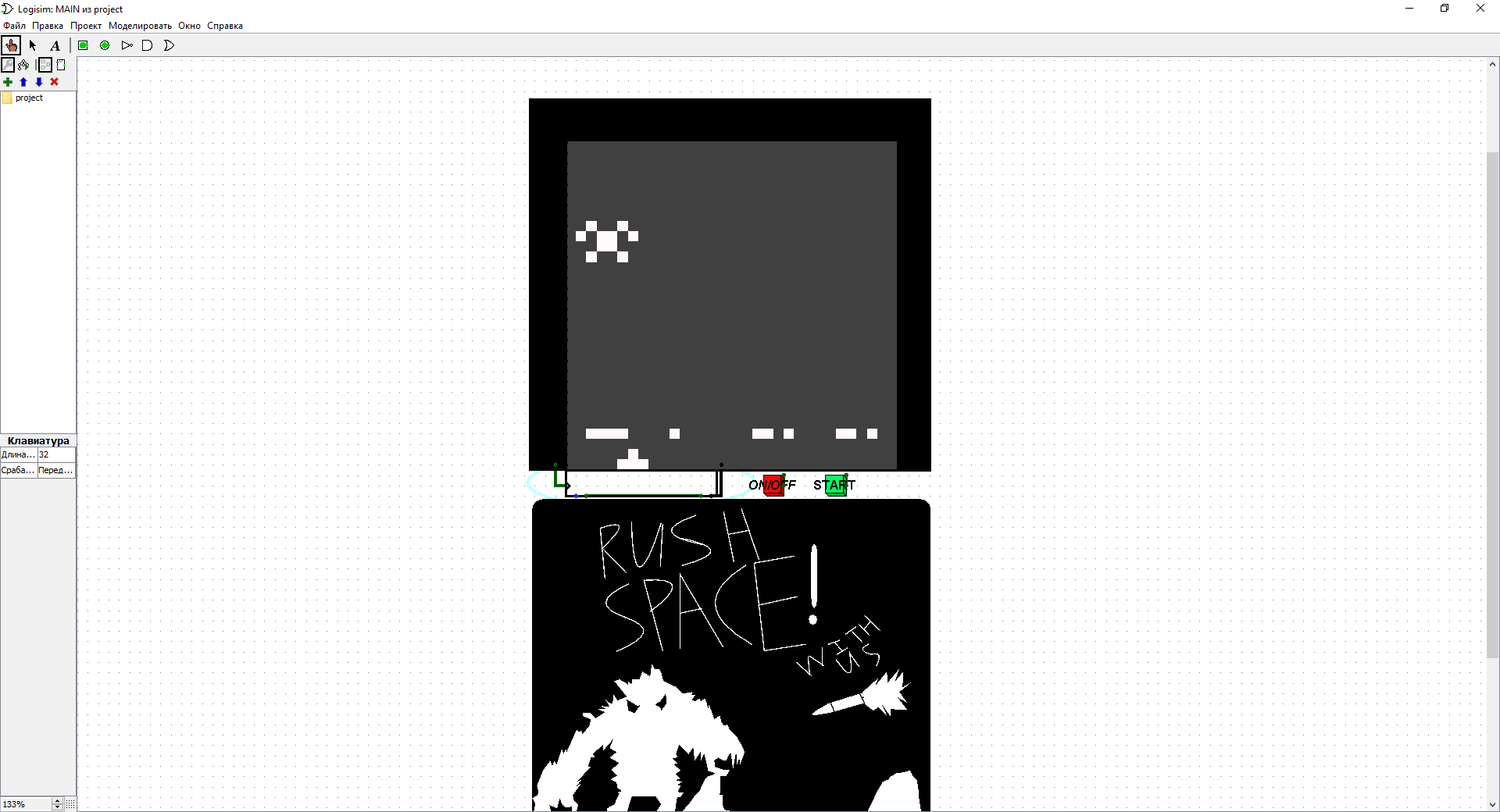


Figure A.8 – screenshot of the playing field after hitting most aliens

After the destruction of all monsters, a message about the victory of the player will show on the screen. Figure A.9 is a screenshot of the win message.

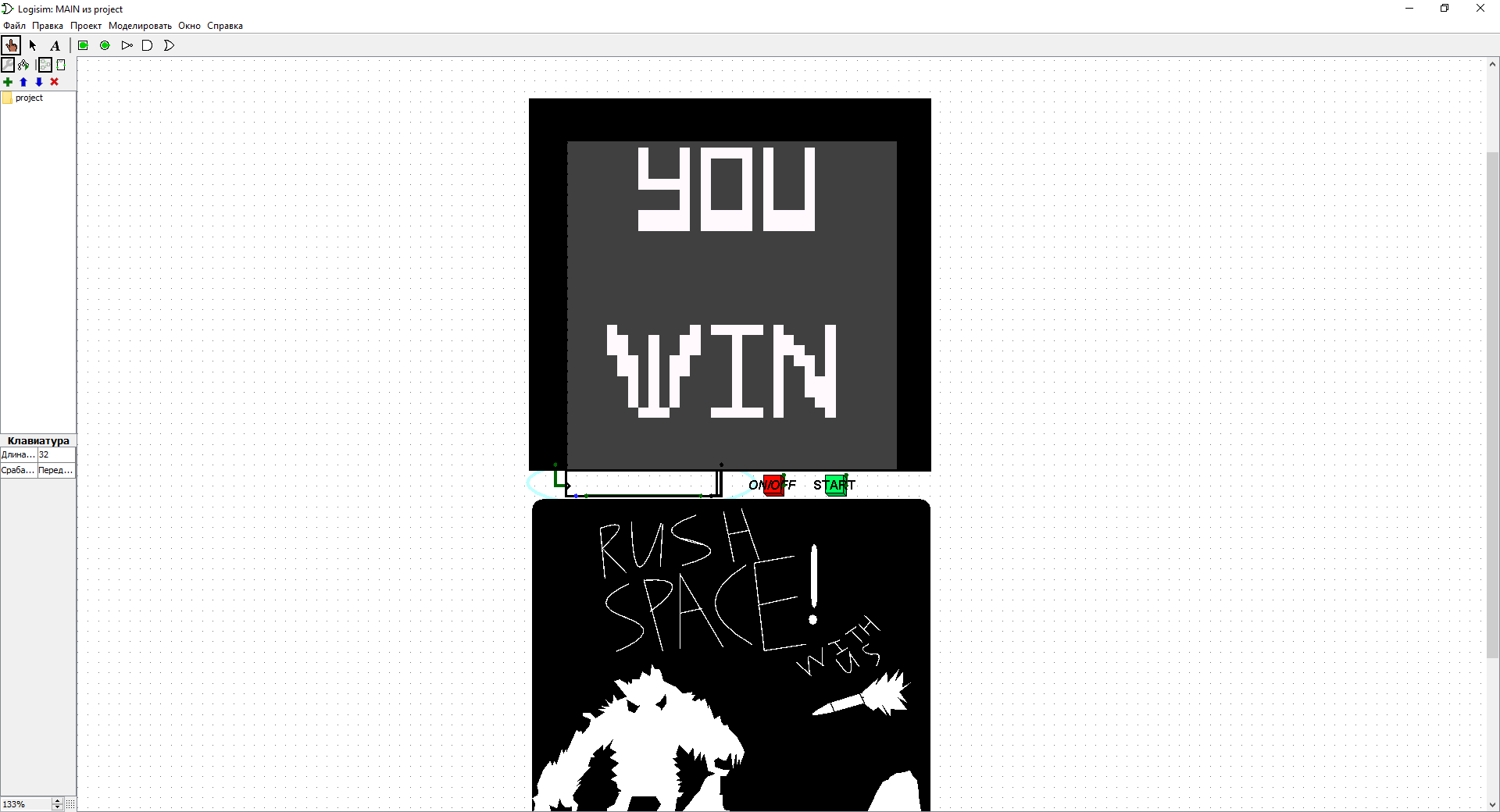


Figure A.9 – screenshot of the win message