**DOCUMENTATION**

**THE GAME OF “REVERSI”**

**Klim Sadov, Ilya Stubarev, Maxim Zaugolnikov**

**2025**

**Introduction**

The result of our project activity is the Reversi game. The project is globally divided into two parts - hardware and software. The entire logic of the player’s moves is implemented on the hardware level (checking the validity of the move, coloring the chips, and controlling the order). And the software part has a bot that can beat even experienced players.

**Изображение выглядит как текст, Красочность, графический дизайн, Детское искусство

Контент, сгенерированный ИИ, может содержать ошибки.**

*Figure 0 (“registers\_to\_matrix”)*

*P.S. We have a very large hardware part. Quite often, Logisim can't stand it and makes jokes (blue and red wires). So, it's better to check everything carefully in the beginning.*

**HARDWARE**

**Display the playing field**

The LED matrices in Logisim support only two states (and therefore 2 colors) - off and on (0 or 1 signal is received). But for our project this is not enough, because the Reversi game requires at least three colors: the color of the field, the color of the first player's chips, the color of the second player's chips. To solve this problem, we decided to use two overlapping LED matrices (Figure 1, blue rectangle). The lower LED matrix, with a size of 12 x 12 pixels, is responsible for displaying the green playing field and the black chips of the computer player. The upper matrix, which has a size of 13 x 12 (the thirteenth column is needed to prevent coincidence of inputs LED matrices) displays your chips, which in our implementation are pink. And to ensure better visibility, for the display of one cell of the playing field are responsible for 4 pixels (that's why the LED matrices are not the size of 6 x 6). The display is controlled through the "registers\_to\_matrix" circuit (Figure 1, blue arrow) connected to the inputs of both matrices. It performs not only the functions of video memory but also combines almost all the mechanics needed for the game to work. We will consider "registers\_to\_matrix" in detail in the following paragraphs, now it is important to understand that it contains 12 six-bit registers - 6 for each matrix, in which all the current game field is stored.

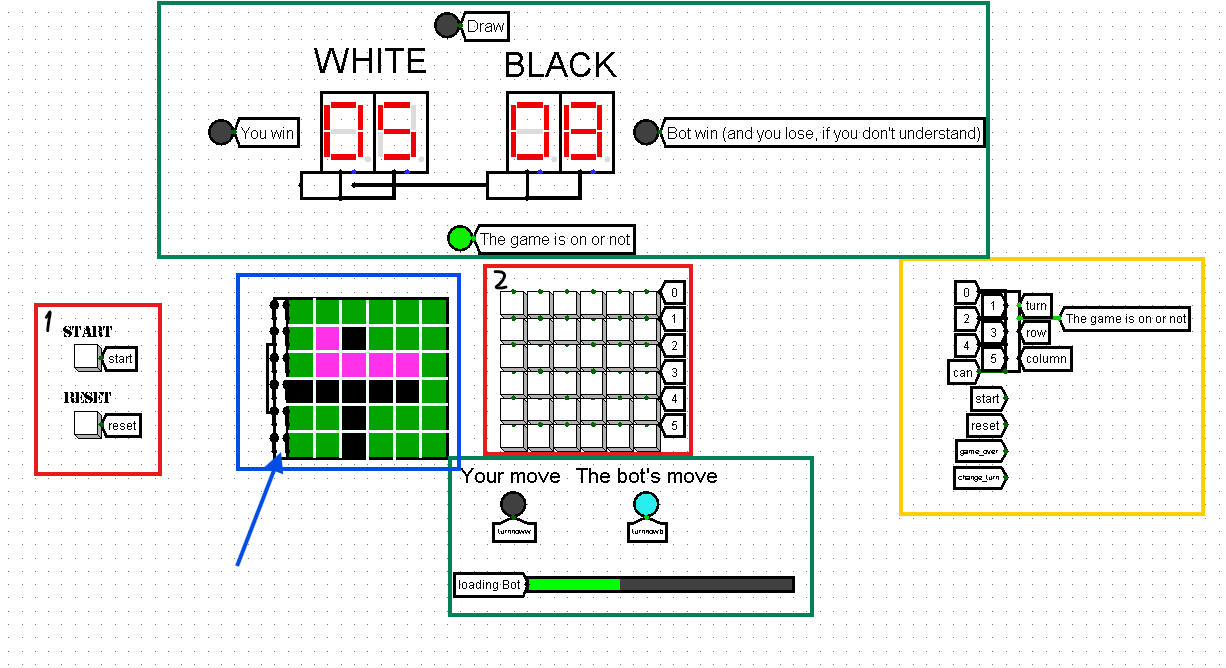


Figure 1 ("main")

In addition to matrices there are several other elements necessary for a better understanding of the game (Figure 1, green rectangles). All of them are also controlled by the "registers\_to\_matrix" scheme.

**Management and data entry**

For interaction of the player with the field 36 buttons are used, united in 6 rows, using the scheme "buttons\_row" (Figure 1, red rectangle 2). All buttons, namely, the rows in which they are located, are constantly processed by the scheme "baza" (Figure 1, yellow rectangle). It determines the current row (as a three-bit number, where the topmost row is "000" and the bottom row is "101") and column (a six-bit number, where the rightmost column is "000001" and the leftmost column is "100000") (Figure 2, red rectangle 1 and 2) of the button pressed on the playing field, if the game is in progress. The state of the game is stored in a register in the scheme "baza" (Figure 2, blue arrow), is set to 1 when the start button was pressed and is reset either by pressing the button "reset" or when the indicator "game\_over" is fed from "registers\_to\_matrix". In addition, the "baza" circuit has an output that indicates whose turn it is now (Figure 2, green arrow). All the data from this scheme goes to "registers\_to\_matrix", where it is processed (whether it is possible to put a chip in the selected place, subsequent coloring).

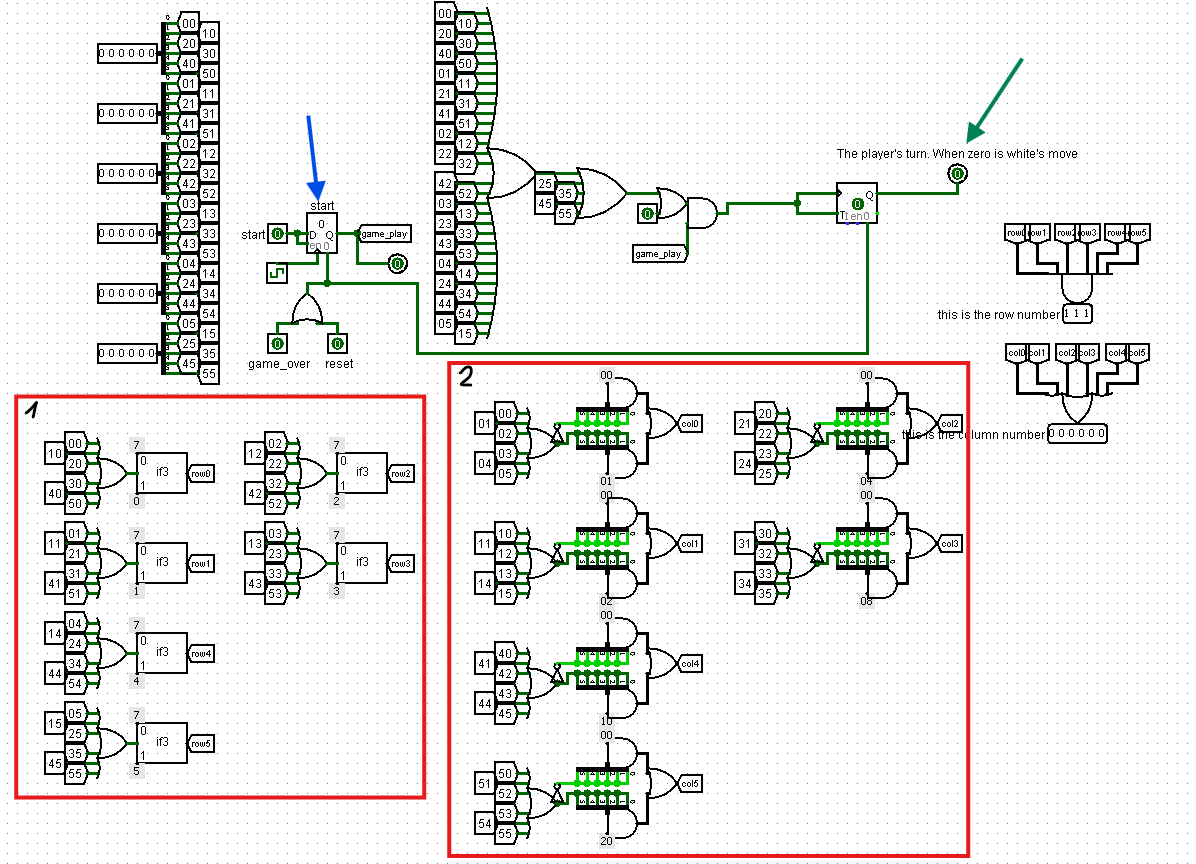


Figure 2 ("baza")

**Permissibility of a player's move**

All the circuits responsible for checking whether a chip can be placed in a selected location on the board are contained within "registers\_to\_matrix". For this check they use such data as: "turn" - whose turn it is now, "column" and "row" obtained from "baza" and the entire current field of 12 registers (Figure 3, yellow highlighting). The values on the registers are updated only if the player's turn is possible (Figure 3, blue highlighting).

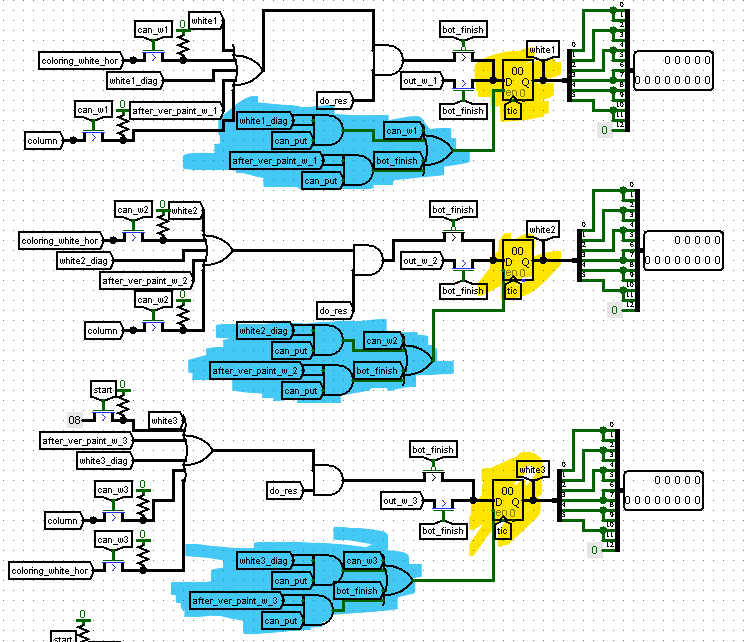


Figure 3 (example of registers from “registers\_to\_matrix”)

When a player tries to place a chip, the hardware checks if it is possible to do so horizontally (left and right from where the user is trying to place), vertically (down, up) or diagonally (down-left, down-right, up-left, up-right).

**Horizontal check**

The schemes for checking horizontally are "check\_left" and "check\_right" schemes. They have the same principle of operation, so it is enough to talk about only one of them. Let's talk about "check\_right". It receives three six-bit pins as input (Figure 4, yellow highlighting): the current white and black rows from the registers (only the row in which we are trying to put our chip) and the horizontal coordinate of the place where we are trying to put the chip (we can say the currently selected column). By bitwise shifts performed on this coordinate, a check is performed to the right of the place we are interested in. The first two shifts check only if the unit has not reached the edge and if the cell on the current shift is not black (because if the neighboring cell is not a black chip, then the coloring in that direction is impossible), then added checks for the presence of a white chip. If at any of the shifts during the check on any of the tunnels "break" was sent one, the check stops and the output pin is sent 0 or 1, depending on the possibility of a move. This implementation allows you to check the possibility of horizontal travel in a one clock cycle (Figure 4, red rectangles).

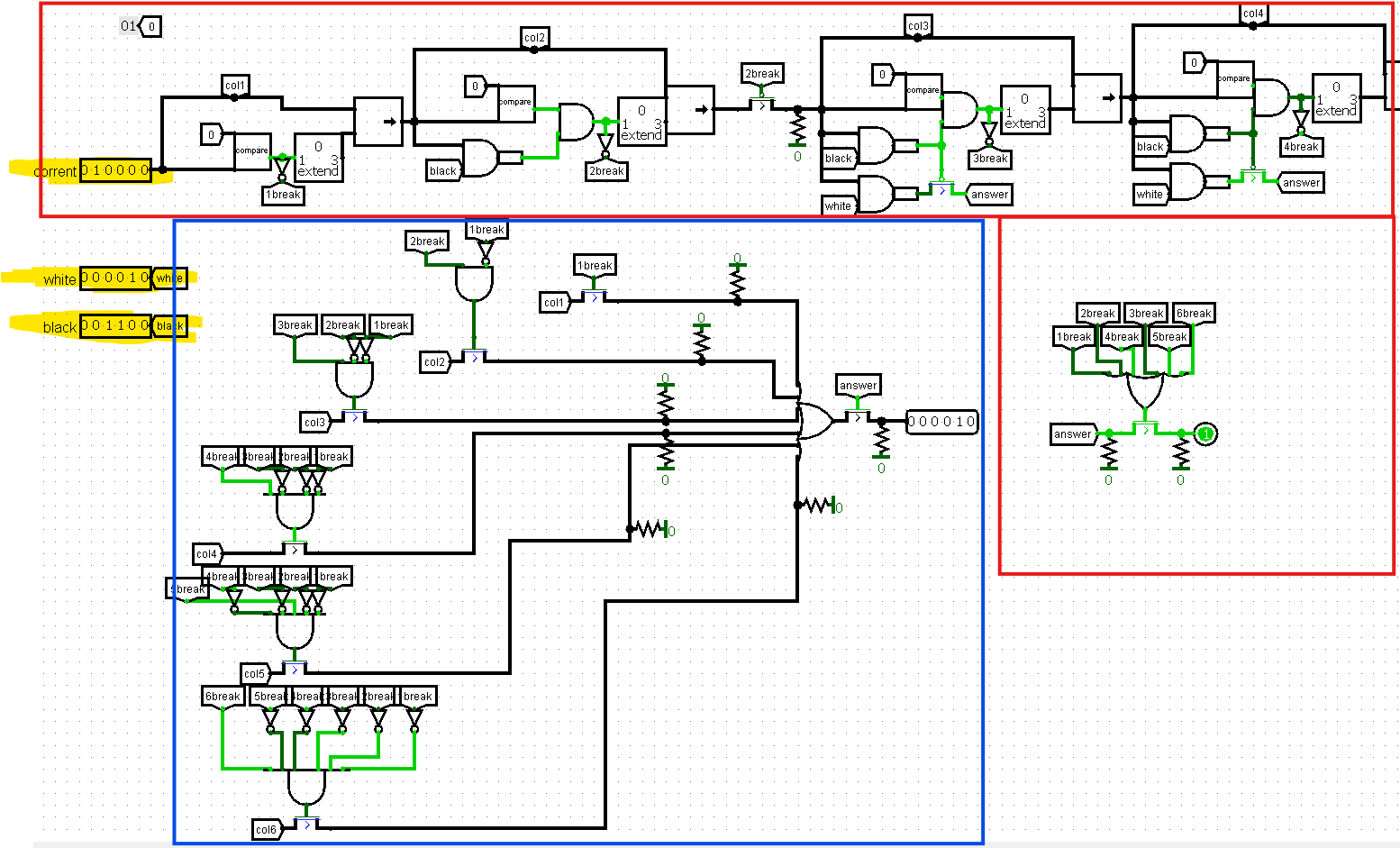


Figure 4 ("check\_right")

Also, on the "check\_right" circuit (as well as on other circuits that check if a chip can be placed), there is an output contact that serves to determine the cell to which we can paint from where we are trying to place the chip (Figure 4, blue rectangle). This pin will play a major role in circuits designed for coloring.

**Vertical check**

For the vertical check the same schemes are used as for the horizontal check, the only difference is that we need to somehow get all the black and white columns and interpret the number of the row where we want to put the chip as a six-bit number. The scheme "column\_to\_row" and the block from the scheme "registers\_to\_matrix" can handle these tasks (Figure 5 and Figure 6).

Изображение выглядит как диаграмма, снимок экрана, текст, линия

Контент, сгенерированный ИИ, может содержать ошибки. Изображение выглядит как диаграмма, План, Технический чертеж, схематичный

Контент, сгенерированный ИИ, может содержать ошибки.

Figure 5 (translation block) Figure 6 ("column\_to\_row")

**Diagonal check**

For diagonal testing we have implemented 4 schemes. They work on the same principle, but check in different directions, from the place where the player tries to put the chip. Let's see how the scheme "check\_diag\_ru" works (check on the top right of the chip). Its work is like the work of "check\_right" by the presence of bit shifts, but here the entire current matrix is transmitted, both white and black chips and shifts occur not only horizontally, but also vertically (i.e. moves not only by columns, but also parallel to this on the rows). In horizontal shifts we simply check that we have not reached the edge, and if we have, we call it "break", because we cannot go further (Figure 7, blue rectangle). In the vertical shifts we do the same as it was in the scheme "check\_right", we check for the presence of a black or white chip on the checked cell of the field, breaking in case of what (Figure 7, red rectangle). The scheme also outputs the horizontal coordinate of the cell up to which coloring will take place. All this still works in one clock, by the way.

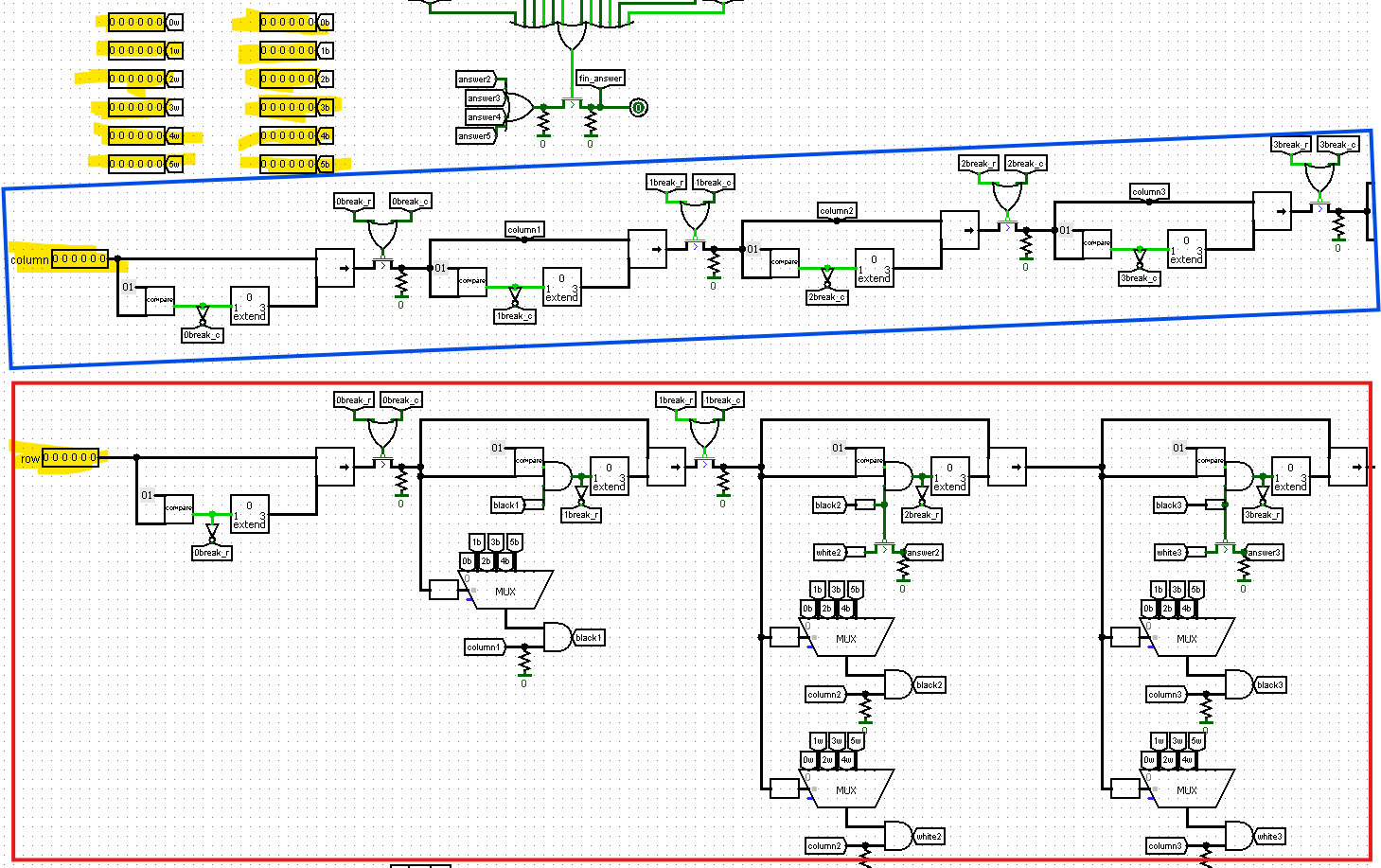


Figure 7 (“check\_diag\_ru”)

**Coloring chips**

All schemes for coloring chips are also stored inside the "registers\_to\_matrix", which we have already almost studied. There are only two schemes: one for coloring horizontally and vertically "coloring\_hor" (Figure 8.1), the other for coloring diagonally "coloring\_diag" (Figure 8.2).

Изображение выглядит как текст, диаграмма, снимок экрана, символ

Контент, сгенерированный ИИ, может содержать ошибки. Изображение выглядит как текст, снимок экрана, диаграмма, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

Figure 8.1 (appearance “coloring\_hor”) Figure 8.2 (appearance “coloring\_diag”)

**Horizontal coloring**

As already said, the scheme for coloring horizontally, only one - "coloring\_hor". Because in it the whole row is colored both to the right and to the left (if possible) of the chip. What inputs does this scheme have? First, the horizontal coordinate of the place where we are trying to put the chip, second, the current white and black row in which we want to put the chip, well, and the horizontal coordinates of the cell to which we need to paint, which we get from "check\_left" and "check\_right" (Figure 4, blue rectangle), and the coordinates both to the left of the current attempt to put, and to the right, so that you can paint both sides. The implementation of the coloring is a bit similar to the check, only here we check if our shifted coordinate has reached not the edge of the field, but the chip to be painted. Also, we don't just shift and wait until we get the result, but at each shift we do bitwise or for white row and for black row (Figure 9).

Изображение выглядит как диаграмма, План, текст, схематичный

Контент, сгенерированный ИИ, может содержать ошибки.

Figure 9 (“coloring\_hor”)

As a result, the circuit outputs the updated white and black rows, which will then be written to registers in the "registers\_to\_matrix" circuit (Figure 10).

Изображение выглядит как диаграмма, текст, План, Технический чертеж

Контент, сгенерированный ИИ, может содержать ошибки.

Figure 10 (“coloring\_hor”)

**Vertical coloring**

Vertical coloring is implemented by the same scheme as horizontal "coloring\_hor". It works the same way as with the vertical check, namely with the help of already known schemes (Figure 5 and Figure 6) the column is converted into a row, and it is treated as a horizontal one.

**Diagonal coloring**

Diagonal coloring, namely "coloring\_diag" is probably one of the most complex and extensive schemes. It takes as input all the current field (12 six-bit numbers), the coordinates of the place where we are trying to go and 4 maximum horizontal coordinates, up to which coloring will take place.

Coloring, as well as diagonal checking, is done by shifts parallel to both vertical coordinates with checks for leaving the field (Figure 11.1) and horizontal coordinates with checks for reaching the chip to which it is worth to color (Figure 11.2).

Изображение выглядит как диаграмма, линия, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.

Figure 11.1 (“coloring\_diag”)

Изображение выглядит как диаграмма, План, карта, схематичный

Контент, сгенерированный ИИ, может содержать ошибки.

Figure 11.2 (“coloring\_diag”)

As well as with horizontal shading on each shift here is bitwise OR between the current white row and the painted position to "paint" cells in white and bitwise XOR between the current black row and the painted chip to "erase" black chips. The circuit returns the entire already redesigned playing field, namely 12 six-bit numbers.

**Processor schematics**

The "CDM16" circuit is responsible for connection to the hardware of the processor and, consequently, of the computer player. It is in "registers\_to\_matrix" (Figure 12). It should be noted that we use Harvard architecture.

Изображение выглядит как текст, диаграмма, снимок экрана, План

Контент, сгенерированный ИИ, может содержать ошибки.

*Figure 12 ("CDM16" from "registers\_to\_matrix "\*)*

The input to the circuit receives the entire game field in the form we are already used to (12 six-bit numbers), the coordinates of the chip placed by the player (necessary for optimization) and a one-bit contact go, which means that the player has finished moving and the bot can start its move. In fact, now when the go contact is signaled, it starts writing values (field, coordinates, bot indicator) (Figure 13, yellow highlighting) into the processor's RAM, at the addresses we have set (Figure 13, pink highlighting).

Изображение выглядит как диаграмма, План, текст, схематичный

Контент, сгенерированный ИИ, может содержать ошибки.

*Figure 13 ("CDM16")*

As already mentioned, among other things, we write a one to RAM - a kind of indicator of bot operation, without which the bot simply will not start working. Also, when we signal go, we save on the register the information that a write to processor was made. Because the bot's operation indicator, in fact, stores information only about whether the bot is working now or not. But it is not always the case that when the bot is not running, it has gone, which is why we need this check (Figure 13, red rectangle).

When all the values are written to the processor, the bot starts to work, and how it works see further in the description of the program part. Now it is important to understand when the bot walks, in addition to updating the data in the memory locations that are responsible for the field, it also changes the memory location with the bot operation indicator to 0. This action stops the processor and gives a temporary 1 signal to the output pin (Figure 13, blue rectangle). When this happens the updated playing field is written to the registers in "registers\_to\_matrix" and the move is returned to you. There is also an output pin, whose values are supplied throughout the bot's operation. It is responsible for loading.

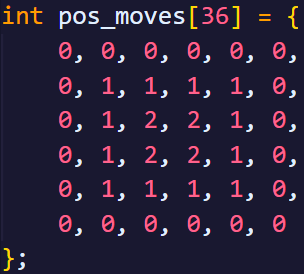
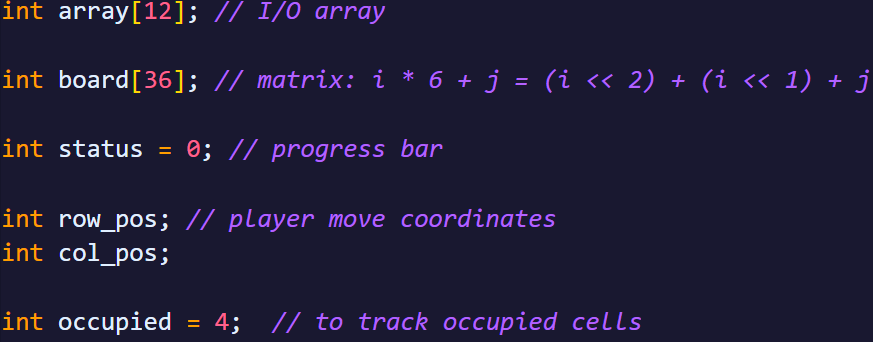
**SOFTWARE**

**Processor logic**

We used the CDM16 processor emulator on the board instead of CDM8 as it would not have been able to fill our needs. Initially it was decided to have the code written in SI, as otherwise the CDM16 process would have been longer and bulkier due to the peculiarities of the language. This code was subsequently compiled into Assembler. Note that only the implementation of functions was in SI, the whole main was written from scratch in Assembler.

**Data presentation in C**

To begin with, we need to understand that to display two-dimensional arrays we used an array named board with special indexing, namely row \* 6 + col, where row and col are rows and columns respectively. This array contains the chips of the player for pink (1) and the bot for black (2), note that for multiplication bitwise shifts are used, i.e. row \* 6 is equivalent to row << 2 + row << 1. Also to speed up the counting of bot's moves is used array pos\_moves, which stores chips of the form: 2 - any color, 1 - position for a possible move, 0 - field. The principle of operation will be outlined below. To prioritize the moves on the field we use the matrix with weights position\_weights, where each weight characterizes how advantageous the position is or vice versa. The last two matrices are initially known, but pos\_moves is changed in RAM as moves arrive, while position\_weights is not. They use the same indexing as board. The global variables row\_pos and col\_pos are used to update the pos\_moves matrix after a player's move. The number of chips on the field is counted using occupied. Remark: in all functions the flag status is used so that the user understands at what stage the calculations with the help of the progress bar take place (Figures 14).



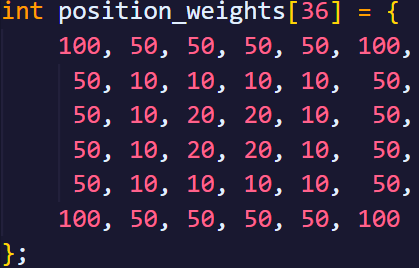
****

Figure 14 (global data)

**Functions**

**Refill\_pos\_moves function**

Working with the pos\_moves matrix requires a special approach, because the input data after the player's move is slightly different, for this purpose there is a block of conditions, which rearranges the values to the desired format. Flag is used so that the function changes the data only, when necessary, because this function is used during initialization and after the bot's move (software and hardware have different indexing). After the processing, the next block of conditions comes, which puts in all 8 sides of the new chip 1 (Figure 15).

**Изображение выглядит как текст, снимок экрана, дизайн

Контент, сгенерированный ИИ, может содержать ошибки.**

Figure 15 (refill\_pos\_moves)

**Decode function**

Let's touch upon some assembly logic for better understanding. After the player's move, 12 elements are loaded into RAM starting from address 0x80 and global\_flag = 1. The first 6 are the playing field with white chips, the next with black, each element is a 6 bit string, where 1 is a chip and 0 is the field. The decode\_to\_bit\_set function is used to represent the data in a new form. It uses bit masks and bitwise AND to write the chip to the board under its value, i.e. it runs through the whole string and finds only 1. The part with white chips is processed first, then with black chips. At the output we get an array with the playing field. The refill\_pos\_moves function is used to update the matrix with possible moves. The function is initialized once per game cycle (Figure 16).

**Изображение выглядит как текст, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.**

Figure 16 (decode\_to\_bit\_set)

**Possible moves with priority**

To calculate the “best move” (the best move is the move that flips the most chips based on the current stage of the game), we try to move to an empty square that is marked as 1 in the pos\_moves matrix. If the condition is met, we start calculating the maximum number of flipped chips, taking into account the weight of each square and the number of chips at each stage of the game. The game is divided into 3 stages: Early, Middle, Late. Initially the priority is in strategy (high priority in the position of the chips on the field), then in tactics (balance between the number of flipped chips and the position of the move on the field), then flip as many chips as possible, that is, for each stage a different priority. To count the chips to be flipped, 3 functions are used: row\_checker, col\_checker, diagonal\_checker. They take 4 values: col, row, player, opponent. The coordinates where we want to move to, which player we want to move from and his opponent. Why we use not just 2 arguments instead of 4 will be told later. The function passes through all the necessary cells and the final value is written to flip\_bonus. The total value of flips is calculated depending on the stage of the game (the stage is calculated using the variable oqupied after each move). Returns the coordinate of the best move (Figure 17).

**Изображение выглядит как текст, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.**

Figure 17 (possible\_moves)

**Row\_checker**

Let's consider only one row\_checker in detail, since all the others are identical. Initially in the condition we make a check for a "logical" move (col > = 2), this means that we can follow the Reversi rules and flip a chip, we also check if the current cell is empty and the next cell is the opponent's. In this case the check is done from the left. A loop starts, which starts immediately from the next square on the left and goes to the border of the board. If an opponent's chip, we update the counter to+ 1. If a player is encountered, then we can flip the chips along this line, increasing max\_count. If a field is encountered, then we exit the loop. Absolutely the same for the pass to the right. After all calculations we return max\_count. The player variable is necessary to view of possible moves for a player to exclude the case when the game stops because the player cannot go. A quick exit is used to avoid calculating possible moves for a player (Figure 18).

**Изображение выглядит как текст, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.Изображение выглядит как текст, снимок экрана, Шрифт, фиолетовый

Контент, сгенерированный ИИ, может содержать ошибки.**

Figure 18 (row\_checker)

**Col\_checker function**

Speaking about col\_checker everything is the same, only for borders it takes row, and checking for the opponent is done by adding or subtracting 6, i.e. a full-fledged row.

**Diagonal\_checker function**

About diagonal\_checker: row and col are used to check borders. And to check all 4 sides, addition and subtraction (-5, +5, -7, +7) are used. A loop with two conditions is used: for row and col. Thus, we pass through each line correctly.The rest of the logic is similar to previous functions (Figure 19).

**Изображение выглядит как текст, снимок экрана, фиолетовый, Фиолетовый

Контент, сгенерированный ИИ, может содержать ошибки.**

Figure 19 (diagonal\_checker)

**Bot logic**

As mentioned above, the bot always tries to find the maximum move from the point of view of the situation on the field. To do this we call possible\_moves for the bot, if it finds such a move we move on, otherwise we exit the function. Refill\_pos\_moves is called to update the pos\_moves matrix. Once possible\_moves has found the best move, the chip is placed at that location and then the coordinate for the move is converted to the row \* 6 + col representation. Next, 8 loops are used to flip the chips in all directions. Let's consider the very first one. It initializes the values to check to the left. We go through the loop until we go beyond the board boundary and until we find the player's chip, the flag changes to 1. If after this loop we have met our chip and all the conditions above are met, we start flipping the opponent's chips. The coordinate c = col\_temp - 1 is reset, we go until the moment when the opponent's chips are over, we turn over the opponent's chips. The rest of the loops are similar to each other and very much like the function loops in possible\_moves. If the cycles are completed correctly, 1 is returned (Figure 20).

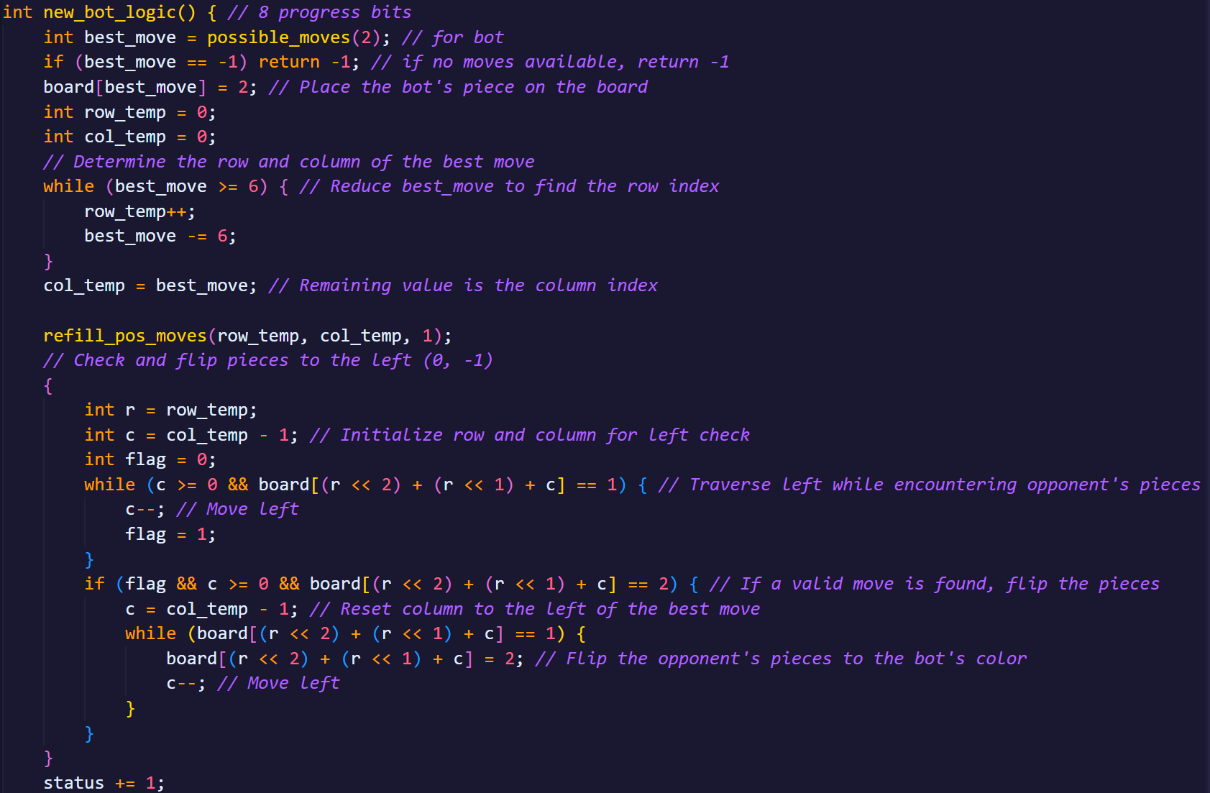
****

Figure 20 (new\_bot\_logic)

**Encode function**

After all the calculations, the new board must be encoded to be passed to the user. We write it the way these values came to us, i.e. in an array of 12 words of 6 bits each (white first, then black). Before each fill we reset the array to zero, since we use a bitmask and bitwise or (Figure 21).

**Изображение выглядит как текст, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.Изображение выглядит как текст, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.**

Figure 21 (encode\_to\_bit\_set)

**Assembler**

Declare part of the labels starting from memory address 0x80. The label for the input and output array is one, it changes only after the player's or bot's move, i.e. it interacts with the scheme directly. The next one is necessary for the matrix (array with special indexing), where further transformations will be performed. Global\_flag is needed to start the bot, because otherwise the bot will not start its work and will be in a waiting state. Flag is needed so that the processor does not stop finally, but only after the legitimate end of the game. Status is needed for progress\_bar, it is updated inside functions (Figure 22). The init\_pos\_moves tag is needed to initialize the pos\_moves array. Further starting from address 0x130 col\_pos and row\_pos are taken after the player's move, they are needed to update pos\_moves. The pos\_moves itself is defined from address 0x140, followed by position\_weights (Figure 23).

**Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.**

Figure 22 (labels)

**Изображение выглядит как текст, снимок экрана, дизайн

Контент, сгенерированный ИИ, может содержать ошибки.**

Figure 23 (labels)

**Main**

To prevent premature termination of the bot, two loops were written. The outer one is responsible for waiting for the processor, it is needed to make sure that it doesn't accidentally exit the right loop, if the flag changes to 1 then the game is over, the processor reads the stop. Global\_flag is needed to enable the bot, also when the state is 0, the hold state is enabled so that the processor does not idle idle. If global\_flag = 1, the bot's logic loop is started. First, it initializes the special indexed pos\_moves matrix in RAM, then decode\_to\_bit\_set is used to transcode the data, then new\_bot\_logic is started, from where flag\_bot is taken to check for further conditions. Now we move on to why possible\_moves takes the value of the player, this is to understand what is currently happening in the game. If the player cannot move and the bot can, the program will run until the player has a legitimate move, there is a check\_bot\_moves tag for this. If the bot and player flags match and are -1, then the game ends since no one can move, we move to the main\_exit tag. Under normal conditions we move to the else\_block tag, which encodes the data and goes to standby (Figures 25).

Изображение выглядит как текст, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки. Изображение выглядит как текст, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.Изображение выглядит как текст, снимок экрана, программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

Figures 25 (main)