**MINISTRY OF SCIENCE AND HIGHER EDUCATION OF THE RUSSIAN FEDERATION**

**FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION OF HIGHER EDUCATION**

**"NOVOSIBIRSK NATIONAL RESEARCH UNIVERSITY**

**STATE UNIVERSITY"**

**(NOVOSIBIRSK STATE UNIVERSITY, NSU)**

**09.03.01 - Informatics and Computer Engineering**

**Focus (profile): Computer science and system technique**

**TERM PAPER**

**Authors**

**Gaev Vladislav Denisovich**

**Masolygin Maxim Olegovich**

**Efimov Stepan Vladimirovich**

Job Topic: **"Checkers"**

Novosibirsk, 2024

**Table of contents**

**Hardware** 4

**Main** 4

**button\_**decoder 9

**draw\_board** 10

**coord\_convert** 11

**find\_between** 11

**variable\_25 ,variable\_32, variable\_16** 12

**white\_black\_log** 12

**checkers\_piece** 13

**Map** 13

**check\_move** 14

**signa\_extension** 17

**signa\_extension\_rev** 18

**CDM** 18

**cdm\_processing** 19

**parsing\_data** 20

**transform\_color** 21

**display** 21

**lvl\_bot** 22

**Software** 22

**Data storage** 22

**Packing and unpacking** 24

**Hashtable.** 25

**Checking the possibility of capturing a shape.** 26

**Movement** 27

**Going back** 28

**Search for possible moves** 29

**Assessment and MINIMAX** 31

**Output from the processor** 34

**Checking the situation on the field when moves are deadlocked** 35

**Bot progress** 36

**Player's Turn** 39

**Game cycle** 40

**Input to the processor** 41

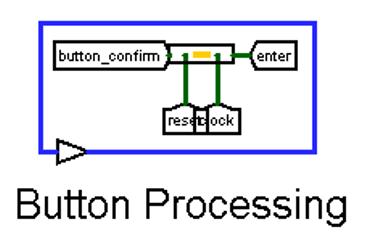
**Conclusion** 42

**Hardware**

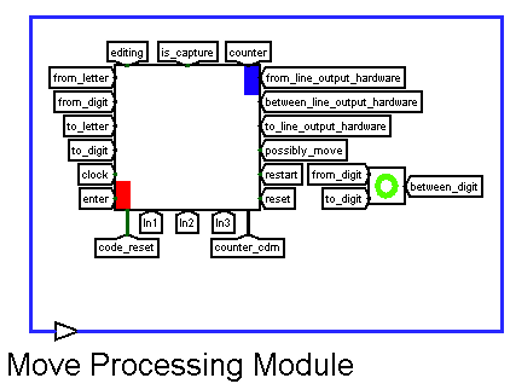
Our project is divided into 20 schemes:

1. main
2. button\_decoder
3. draw\_board
4. coord\_convert
5. find\_between
6. variable\_25
7. variable\_32
8. variable\_16
9. white\_black\_log
10. checkers\_piece
11. map
12. check\_move
13. signa\_extension
14. signa\_extension\_rev
15. CDM
16. cdm\_processing
17. parsing\_data
18. transform\_color
19. display
20. lvl\_bot

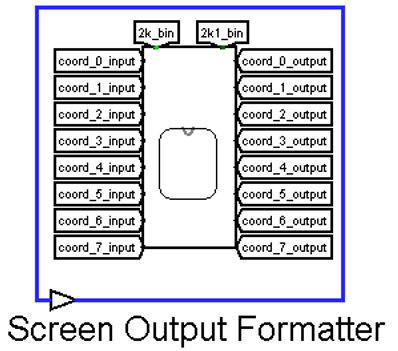
## **Main**



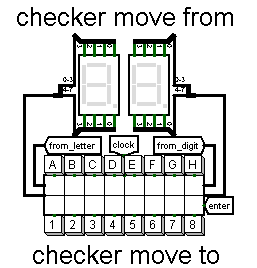
1. The Button Processing module is responsible for correctly pressing the progress confirmation. It is needed so that the program does not depend on the duration of pressing the button.



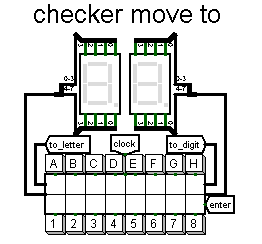
1. The Move Processing Module is used to pass information about the position of the piece that we want to move and the overall progress to check\_move.



1. Screen Output Formatter uses information about the shape type to render it onto the board.



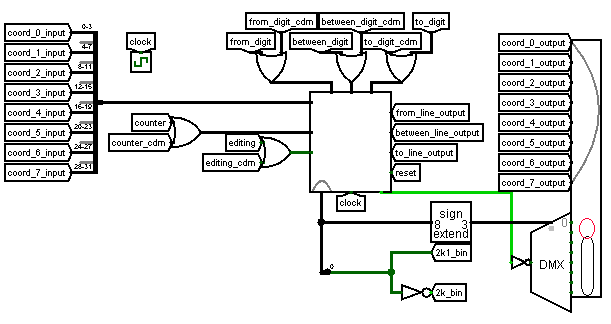
1. checker move from – used to enter the initial position of the piece that the player wants to match. To enter coordinates, use the standard checkers Algebraic notation.



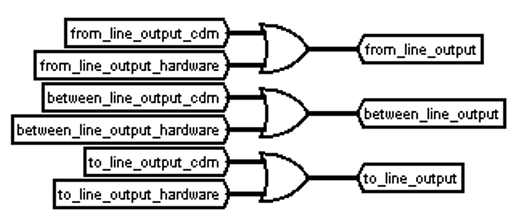
1. checker move to - remembers the coordinates where the player wants to move their piece.



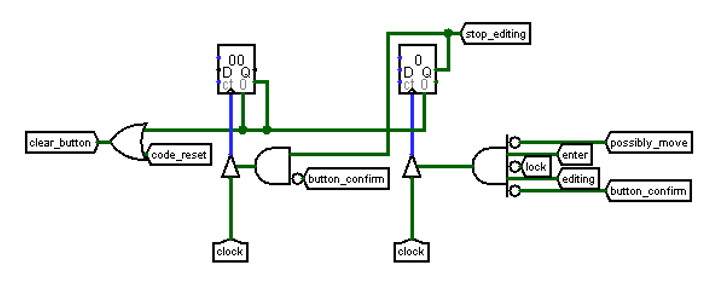
1. confirmation of the move - button to confirm the move.



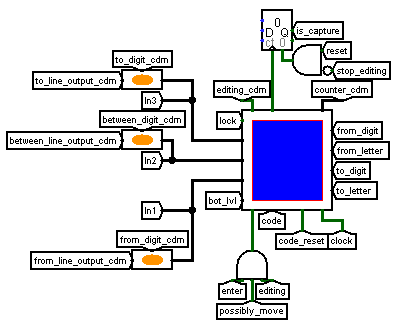
1. This part of the scheme is responsible for processing the correctness of the move, working with memory, it is responsible for allowing changes to the board



1. This scheme is used to overwrite the coordinates of the player's and bot's moves in hardware memory



1. The logic of this scheme is to identify situations when the coordinates that the player entered are either not needed or are not correct.

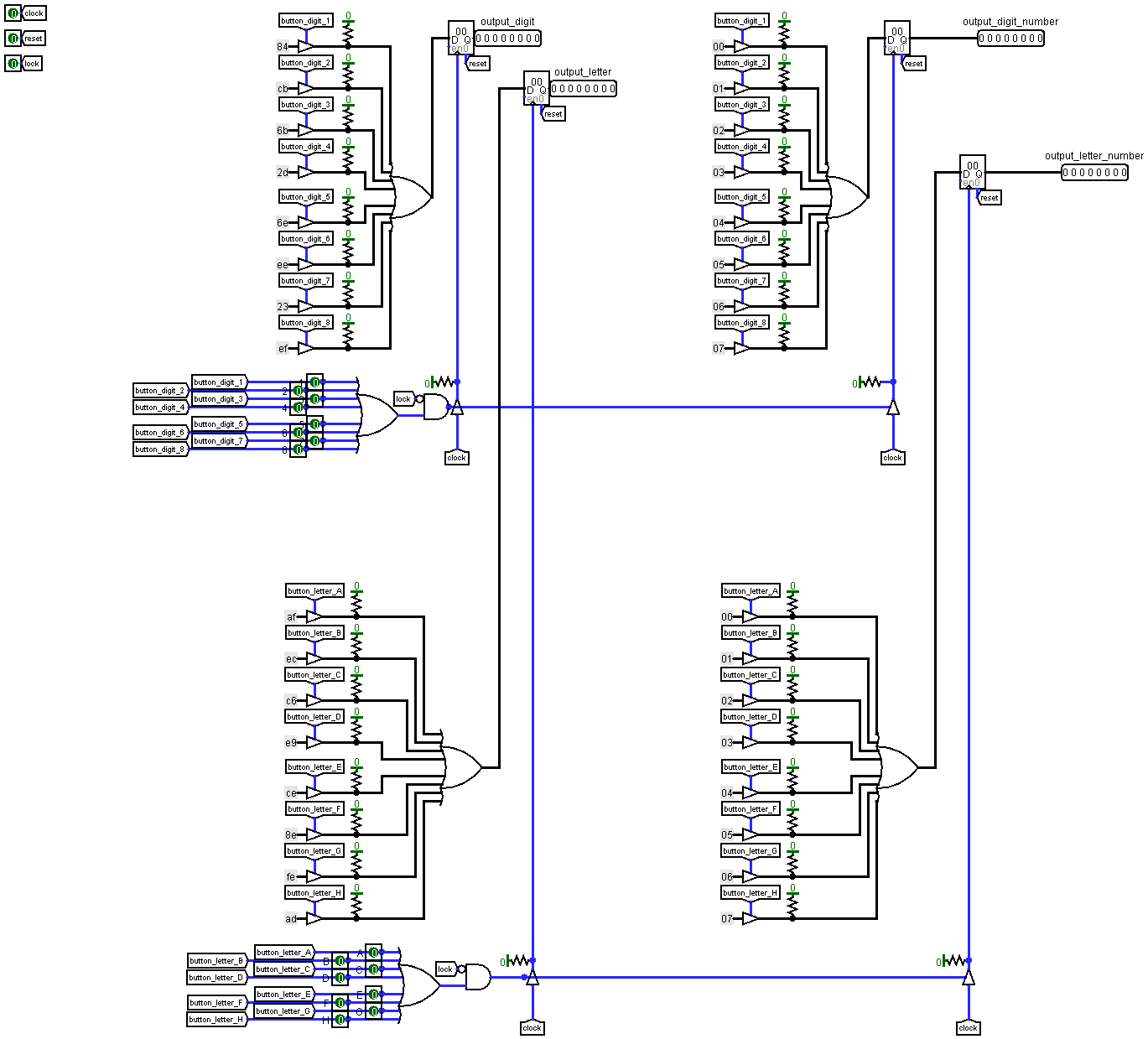


1. This scheme is used for working with the processor, processing character input, managing editing, and interacting with external modules. The main task is to receive, transform, and transmit coordinate information.

## **button\_decoder**

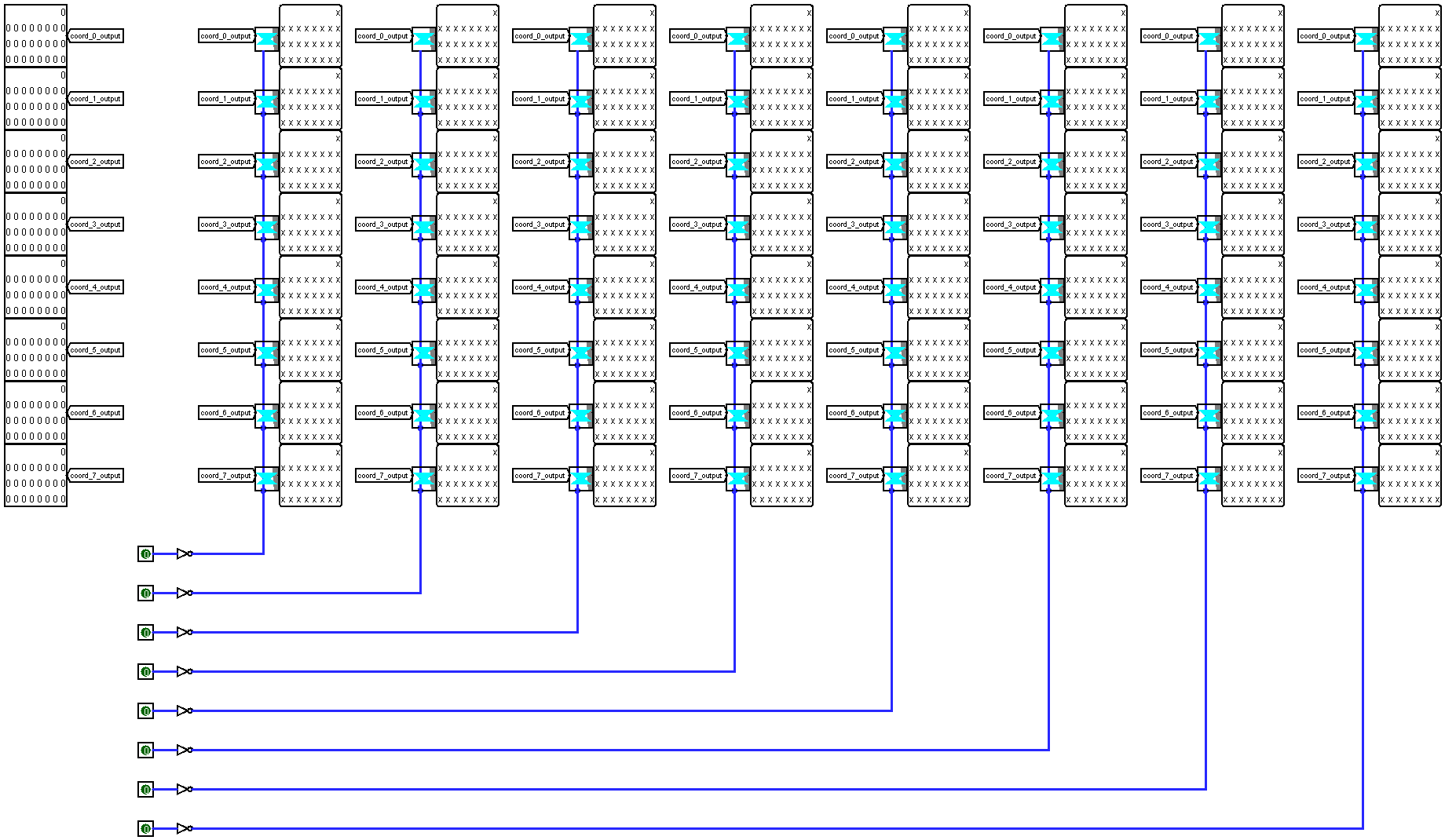
It consists of four identical circuits, in which it differs only in input/output. Top left is a graphical representation for the 7-segment indicator. Upper-right indexes of numbers. The same applies to letters.

An enter signal is also sent to the input to clear the registers when the move is processed.



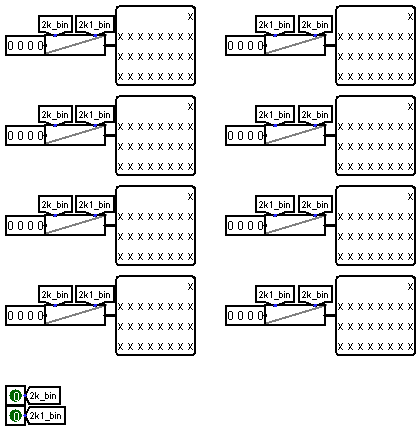
## **draw\_board**

This is the pictogram required for drawing the board correctly. Inside it there are 8 identical schemes, which are the lines of the board (from bottom to top on the board, from left to right in the scheme). Each of the parts represents an input, a register save, and an output. Correct operation is provided through the parallel inputs of each of the lines. When a single signal arrives at the demultiplexer (in the main circuit) and a line is selected only on this circuit, a clock cycle (0-1 (trailing edge in variable\_25)) occurs and registers are overwritten



## **coord\_convert**

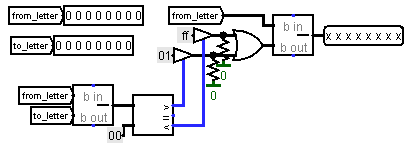
Converts the 4-bit code for each position in the current line to the 25-bit code required for drawing.



## **find\_between**

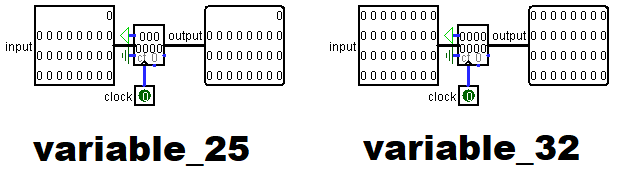
Finds the letter index for the coordinate between from and to.

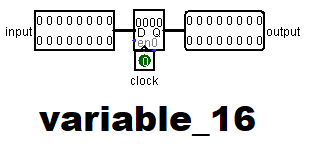
from\_letter and to\_letter - start and end cells of the move.



**variable\_25 ,variable\_32, variable\_16**

25 or 32, or 16 bit register to store the specified bit depth (back edge)

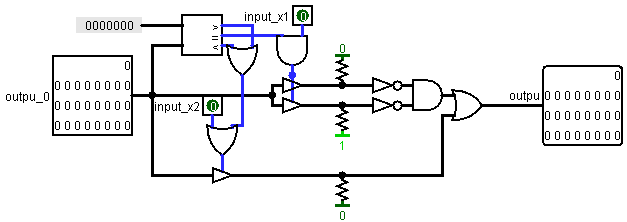




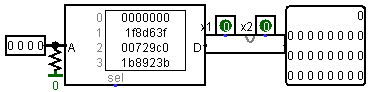
## **white\_black\_log**

The input is a graphical representation of the shape, the parity of the line and line. Initially, it checks whether there is a shape at the input at all or if there is nothing there. If it is even, then we pass it without changes (either the shape itself or 0 will come out,which suits us. Otherwise, if the cell is odd and empty, then not(x) and not(x) = not(x) = not will occur(0...0) = 1...1 (the cell has turned black), if one of the two fails, then matching registers will result in this expression 1..1 and 0..0 = 0..0 (the cell is white).

Since 1 input and 2 input are always not equal, only one of the outputs will work, we set logical or

**checkers\_piece**

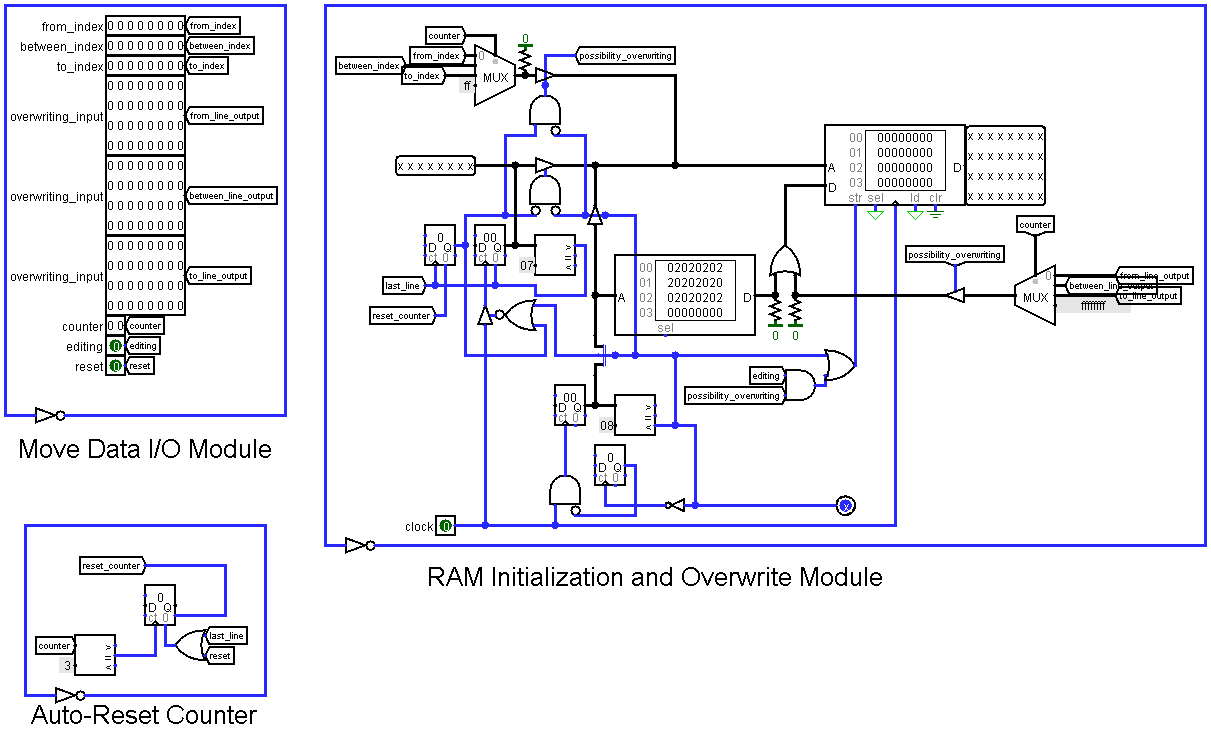
Takes from the ROM that stores which indexes correspond to the graphical representation of the shape.



## **Map**

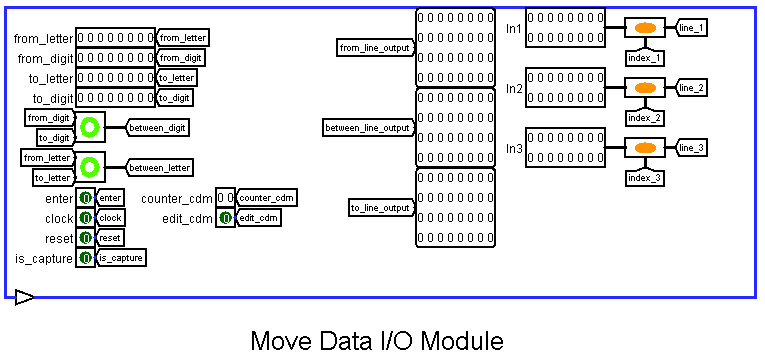
For the first 8 clock cycles, the RAM takes values from the ROM (loading the card), then the ROM is closed and the next 8 clock cycles data is unloaded from the RAM to the card.

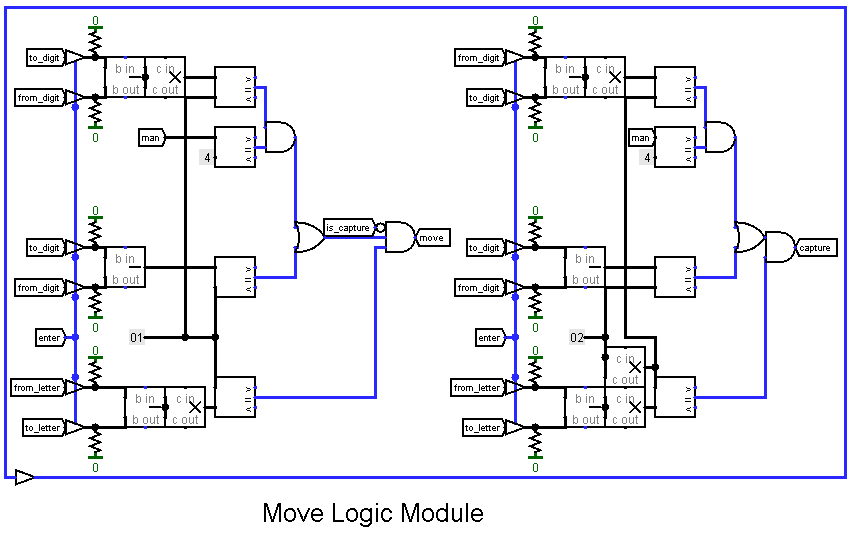
Rewriting of values (check\_move) is also implemented here



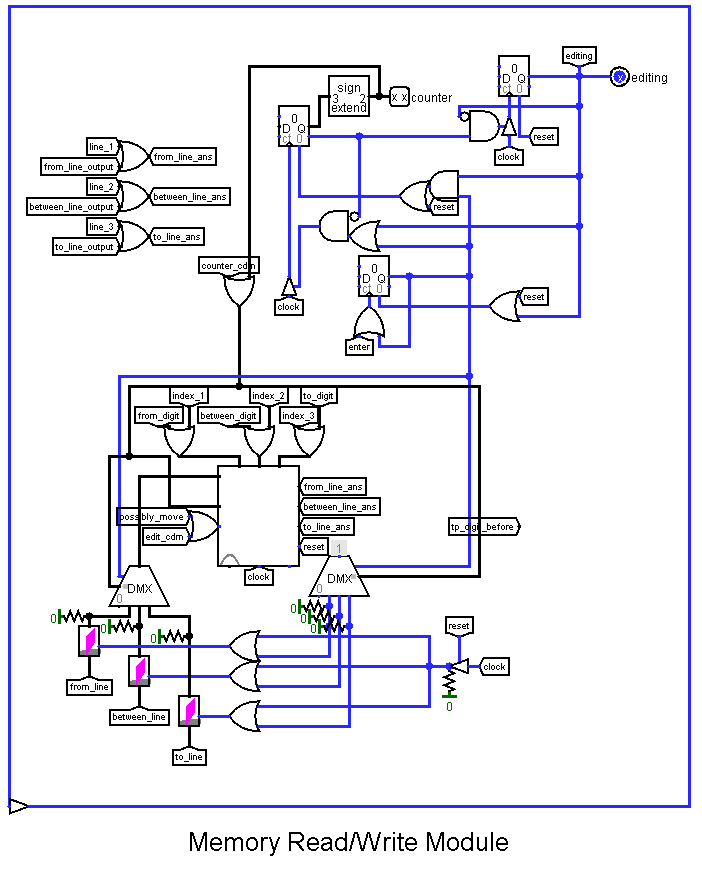
## **check\_move**

Move Logic Module-checks the possibility of moving or taking through coordinate checking (initial check for correctness of input data.)

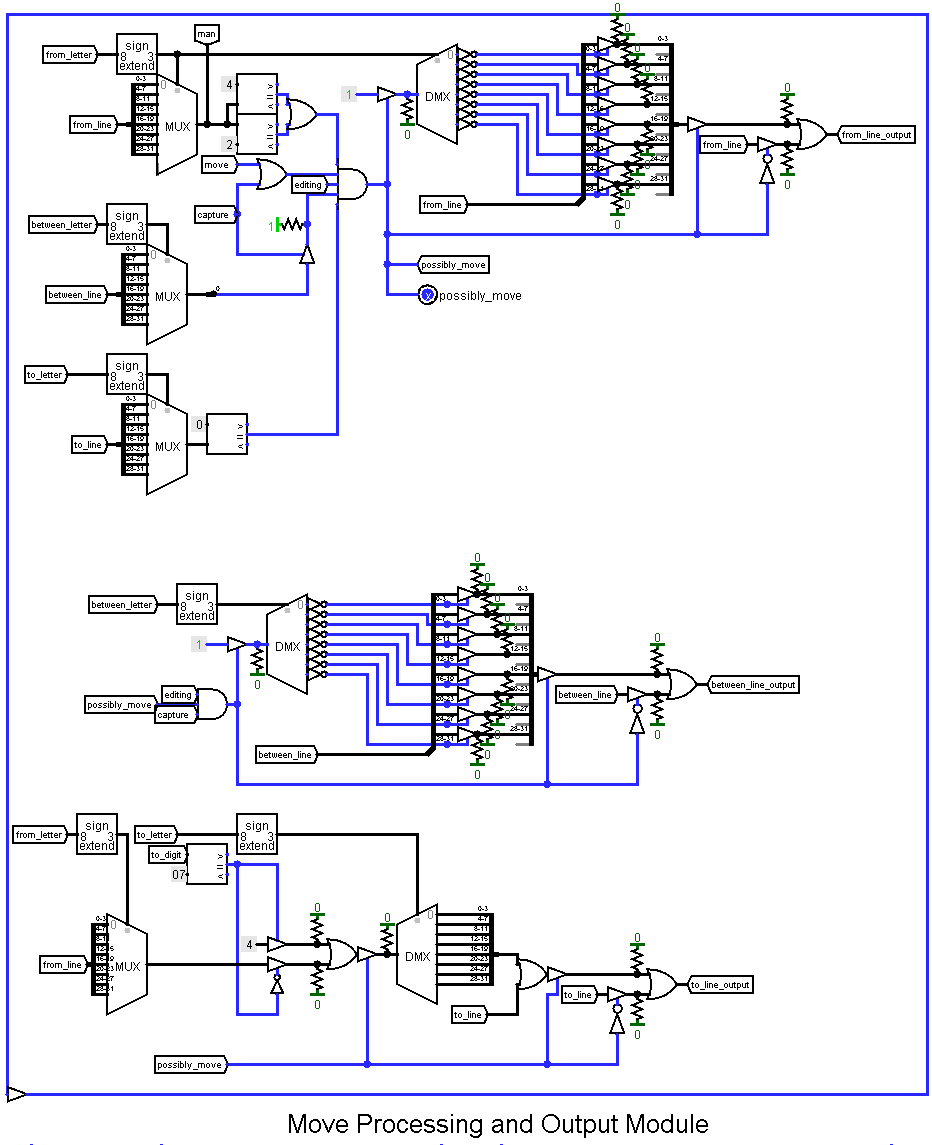




Memory Read/Write Module-takes the necessary lines from the internal memory of the module, and if the move is successful, then overwrites them to RAM. This is done sequentially by selecting coordinates and values through a counter and multiplexer.

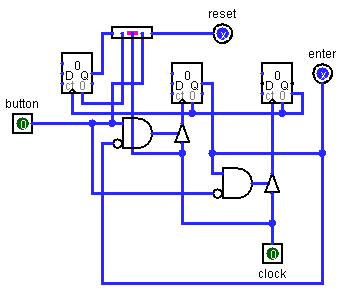


Move Processing and Output Module – final check and data modification if the check is successful. Takes three values: from where, through which (if the player has set the capture of the piece) and where the piece goes. The from part checks whether there is a piece at the from position and whether there is no piece in the to, and if the move is successful, it removes the piece from the from line. Between similarly checks whether an enemy piece is in it, and if there is free space in to, a capture will occur and the piece will be removed from From and Between. Part to takes from from the shape that looked like and makes a mask that is superimposed on to.



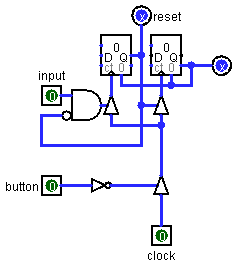
## **signa\_extension**

This scheme extends the input signal until the end of data processing.



## **signa\_extension\_rev**

This scheme outputs a signal after the end of data processing.



## **CDM**

This is the circuit on which our processor is based.

At the output, the processor sends a flag to our hardware in order to report events planned or occurring on the field and the coordinates of the bot's progress(from where to where and between).

0 - nothing

1-the player's turn, you need to submit the coordinates

2-empty (there will be a mandatory log cabin by the player)

3-victory

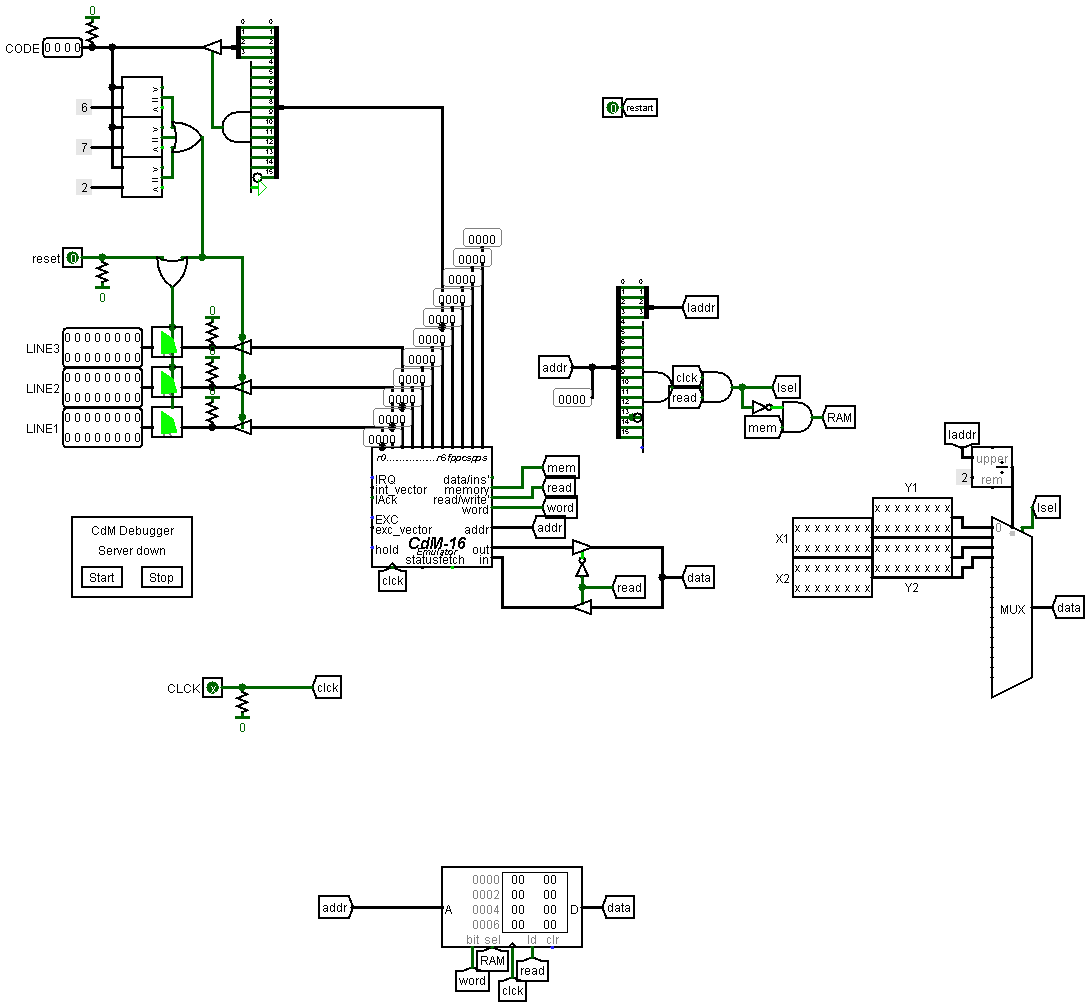
4-defeat

5-continuation of the player's turn

6-continuation of the bot's progress

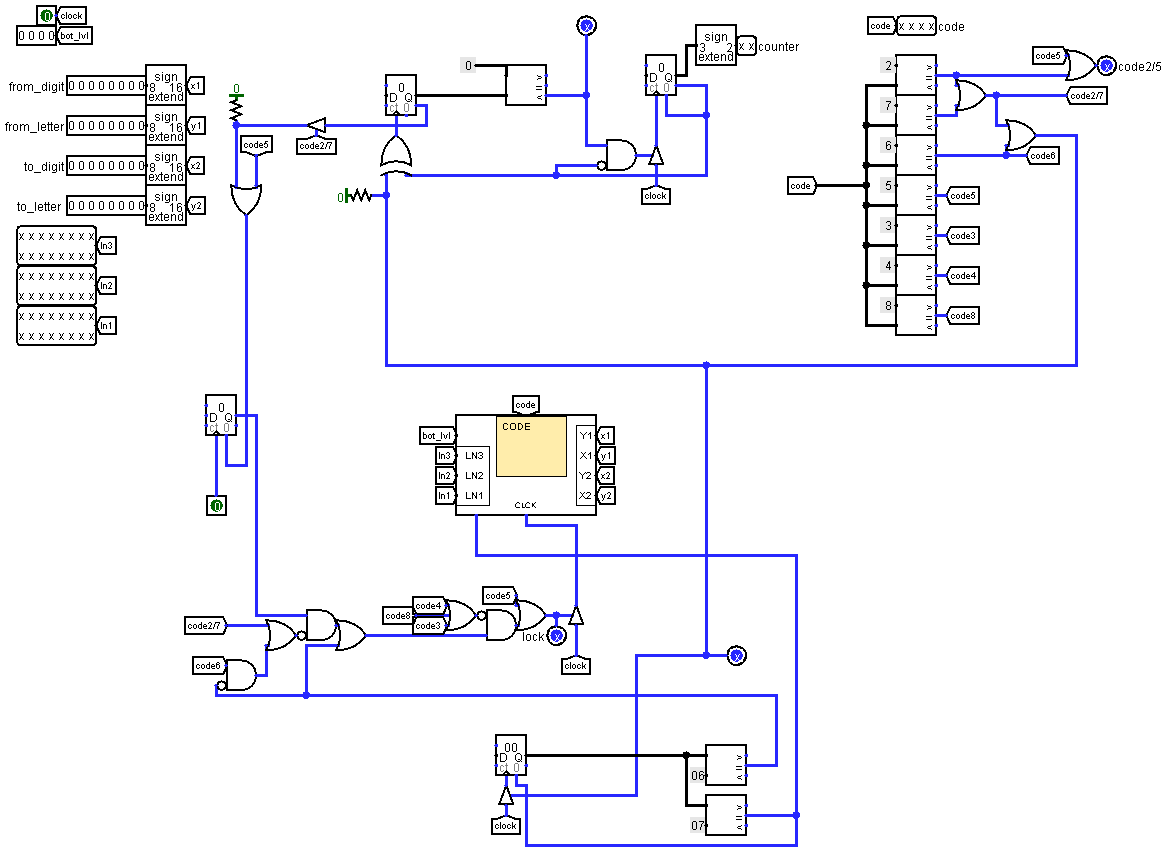
7 - the bot's turn is over (no continuation)

8-draw



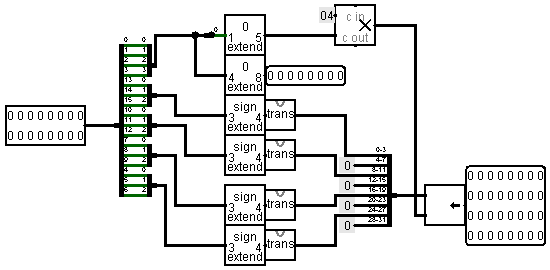
## **cdm\_processing**

It is needed for processing data from the processor. We get information about the lines using three lines that indicate the numbers of the types of pieces whose position corresponds to their position on the board cells.



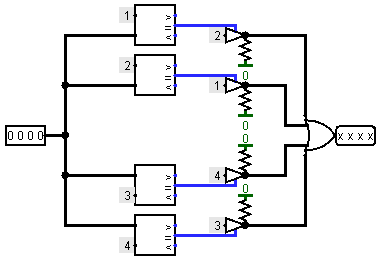
## **parsing\_data**

This scheme is used for parsing data from the processor(32-bit lines+ 8битный -bit line number)



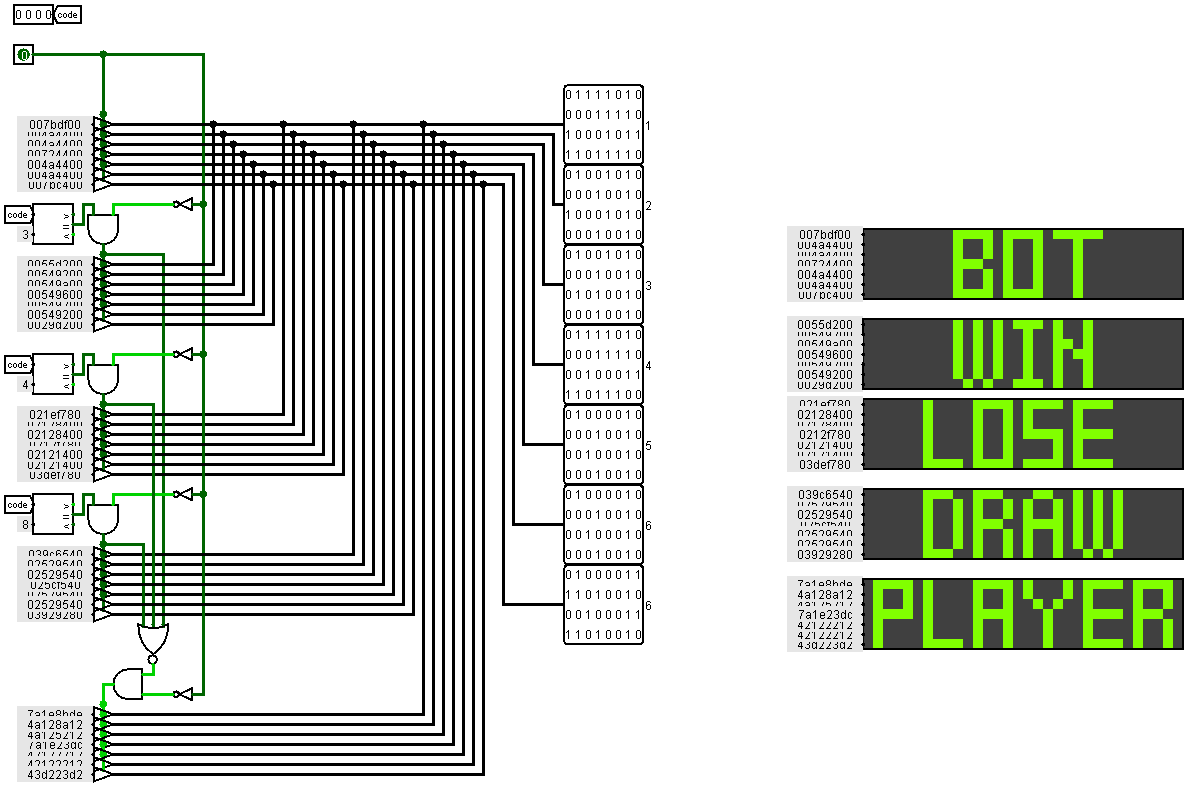
## **transform\_color**

This scheme is necessary for the correct collaboration of software and hardware, matching the color of shapes.



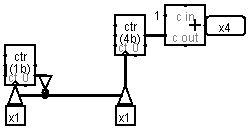
## **display**

This scheme is necessary for displaying the current stage of the game on the display: the turn queue, the game result, and the drawing process. Also the essor procессор checks the multiplicity of the player's log house so a false announcement of the bot's turn queue appears between the player's moves

****

## **lvl\_bot**

This scheme is used to select the bot's difficulty level.



# **Software**

We decided to use the CDM-16 processor, with the Harvard architecture, to create the logic of the opponent's moves created for our player, tracking the moment when the piece should become a queen.

As input, the processor receives:

* Signal from the clock generator
* Coordinates of the position from which the player moved his piece
* Coordinates of the position to which the player moved his piece

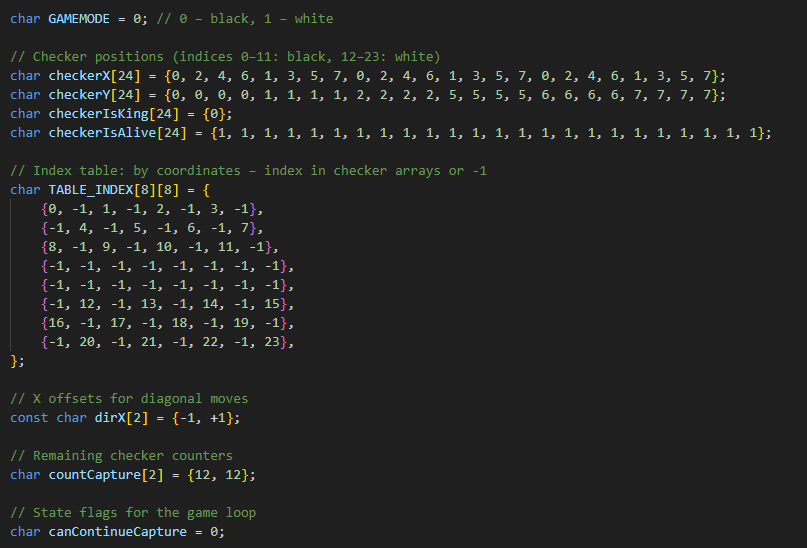
Transmitting the clock signal of the generator is necessary for us to prevent clock stacking.

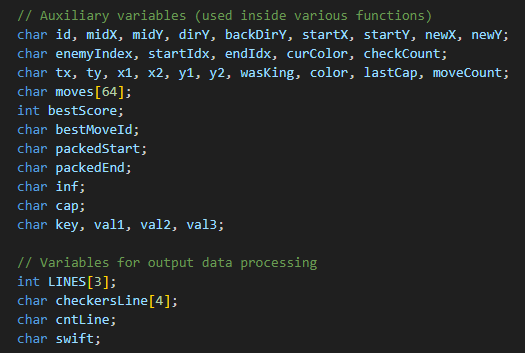
We have implemented two game modes, a bot without MiniMax, which works on primitive conditions, and a bot with the MiniMax algorithm. An implementation without an algorithm is faster.

To determine the best move, we use the MiniMax algorithm. Its essence is to calculate the coefficients of the "profitability" of the move. For this, the processor recursively calculates possible plots for each piece on the field.

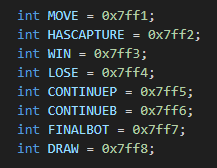
## **Data storage**

We store most of the data globally to speed up access to it.

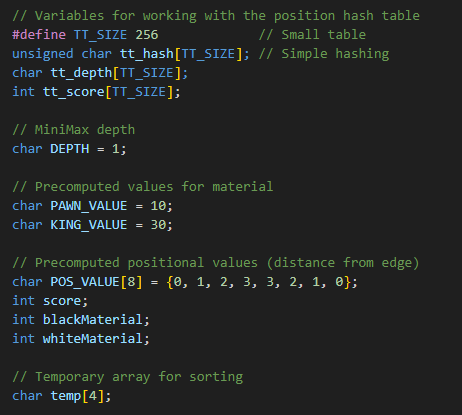


****

For the hardware part to work, we output some codes that help you combine the hardware and software parts

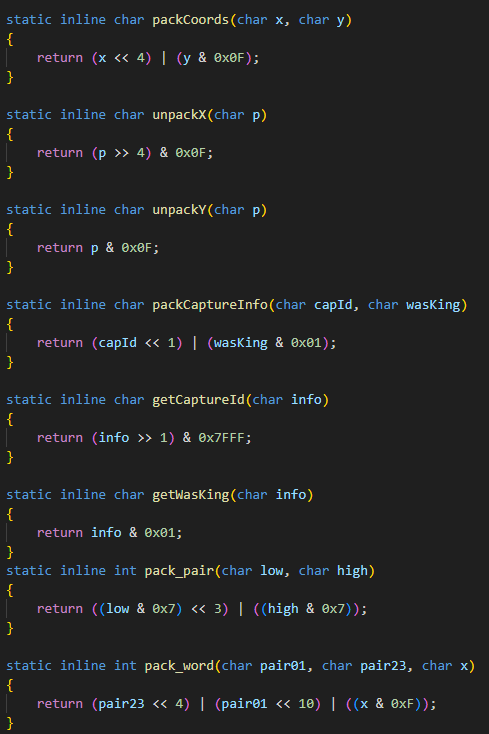
****

There are also variables that are only used in the MiniMax implementation



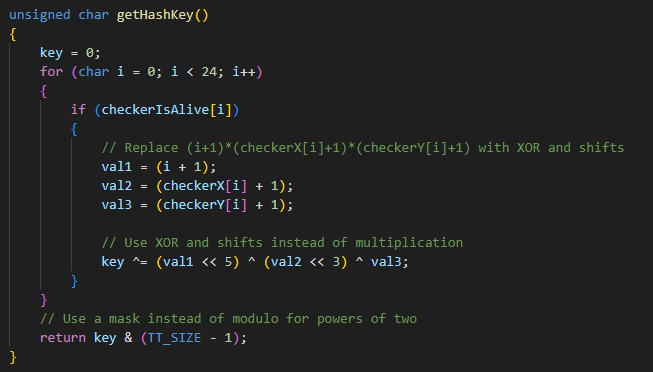
## **Packing and unpacking**

These functions of packing and unpacking data are necessary in order to pass several arguments in one, this has to be done to make it easier to compile through Clang.

****

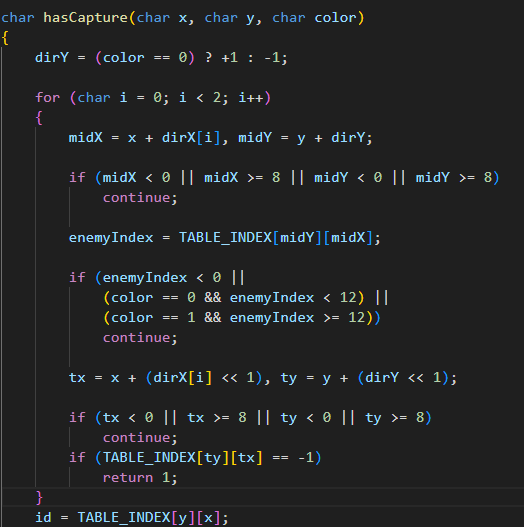
## **Hashtable.**

To save calculated positions in the implementation of the MiniMax bot, we use hash tables. The getHashKey function returns a key that we use to save the calculated options

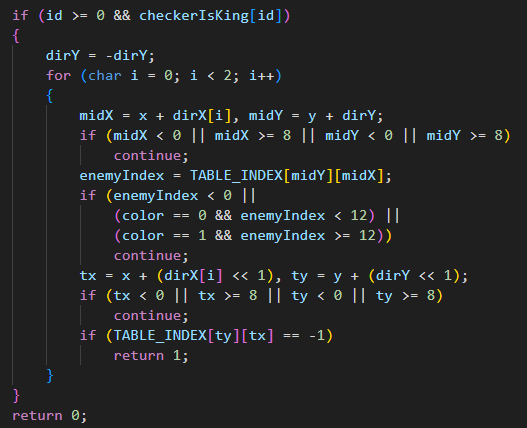


## **Checking the possibility of capturing a shape.**

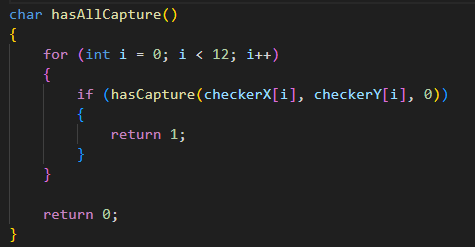
Here we consider the case for an ordinary checker, the diagonal cells are checked for the color of the figure that is in them and for the free cell after them.

****

In the continuation of this function, it checks whether the piece is a queen and in this case also considers the possibility of cutting down going back. The result of the function is the number 1 if taking is possible and 0 otherwise.

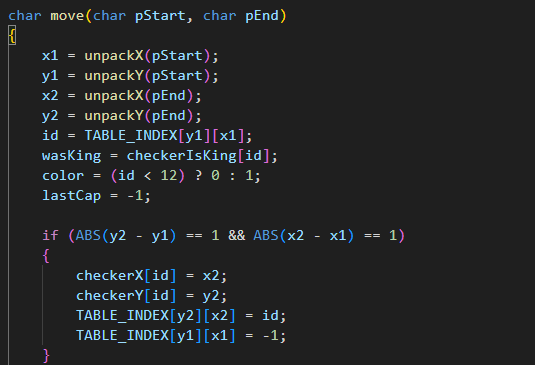


Also, for the game cycle, we will need to check the presence of a log house with the player's figures

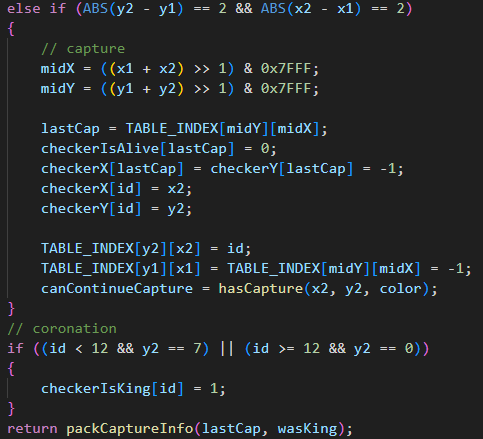


## **Movement**

Функция The move function unpacks the input data and uses it to get information about whether the shape is a die and its color. It also shows the usual move for 1 square.

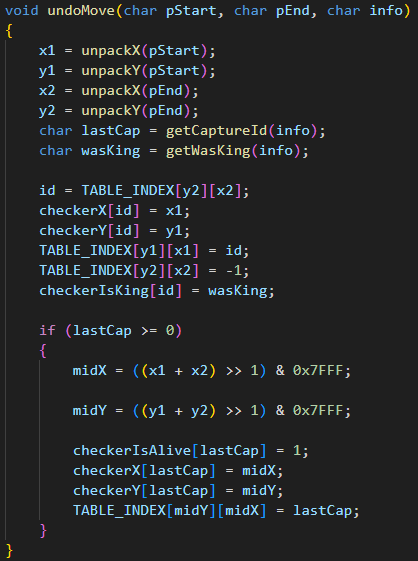
****

The second part of the function deals with the situation of capturing a piece and checking whether the piece has become a queen as a result of its movements. The function results in packed information.



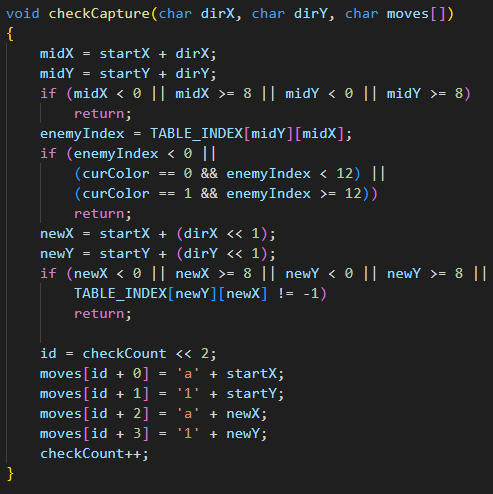
## **Going back**

Changes the previous move and returns the piece to its previous state.  
It is used to roll back the changes made.

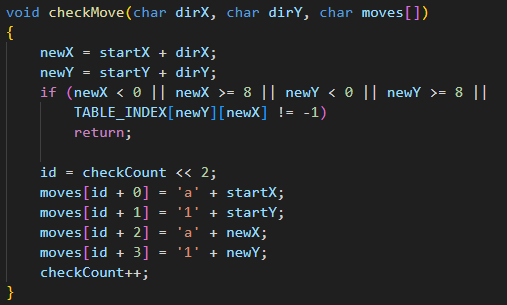
****

## **Search for possible moves**

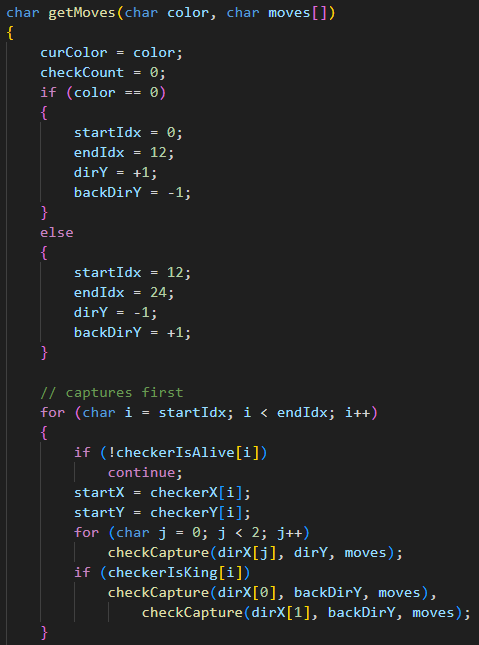
**checkCapture** looks for opportunities to capture an opponent's piece and adds them to the moves array. By gluing together the coordinates of the beginning + the coordinates of the end of the turn.



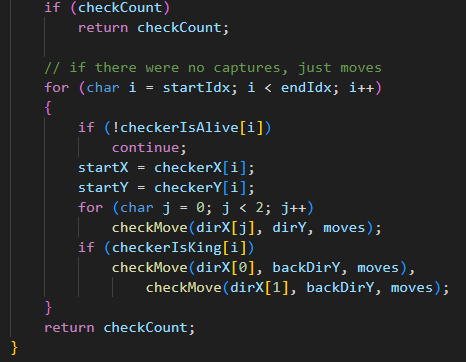
**checkMove** searches for the normal moves of a piece and adds them to the moves array

****

**getMoves** returns all possible moves for the player. Separately, there is a check for whether the piece is a queen and in this case adds additional moves.

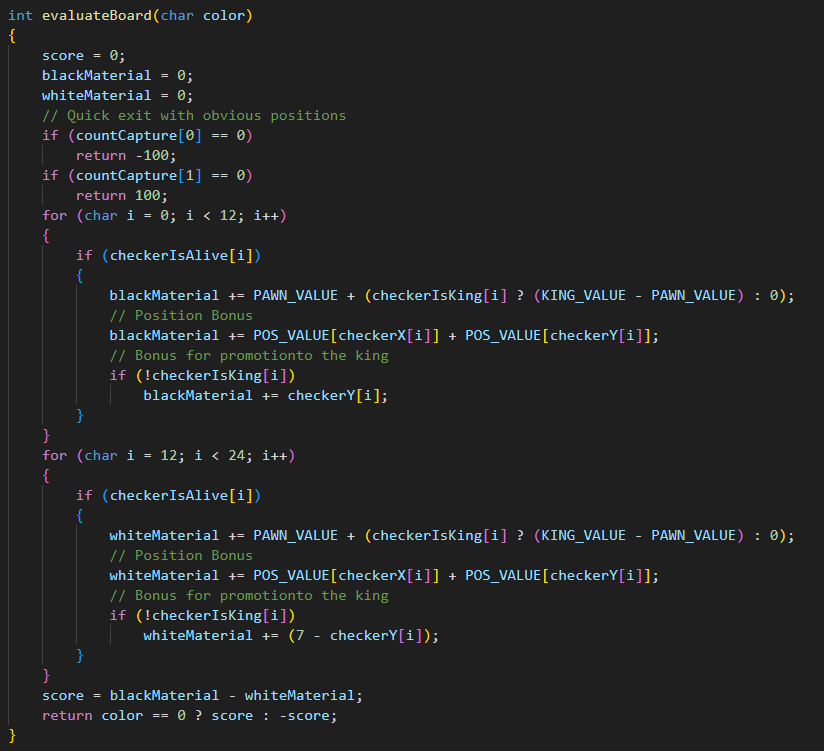
****

Checks if log cabins were found, then returns the number of possible moves, otherwise just moves are performed and then the number of moves is also returned.

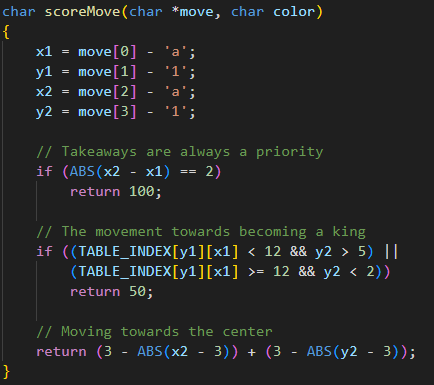
****

## **Assessment and MINIMAX**

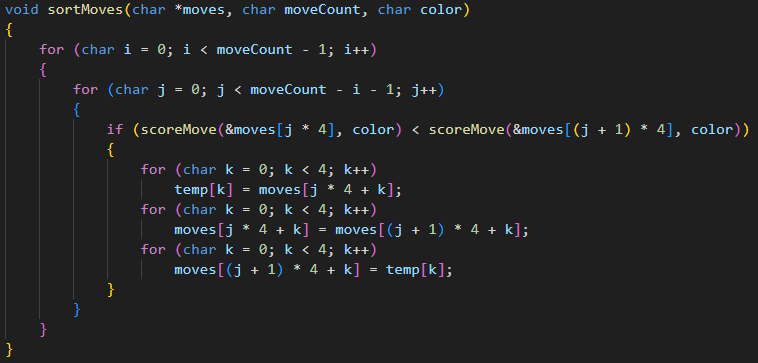
evaluateBoard is used to calculate the current position on the board. There is also a quick exit if one of the players wins

****

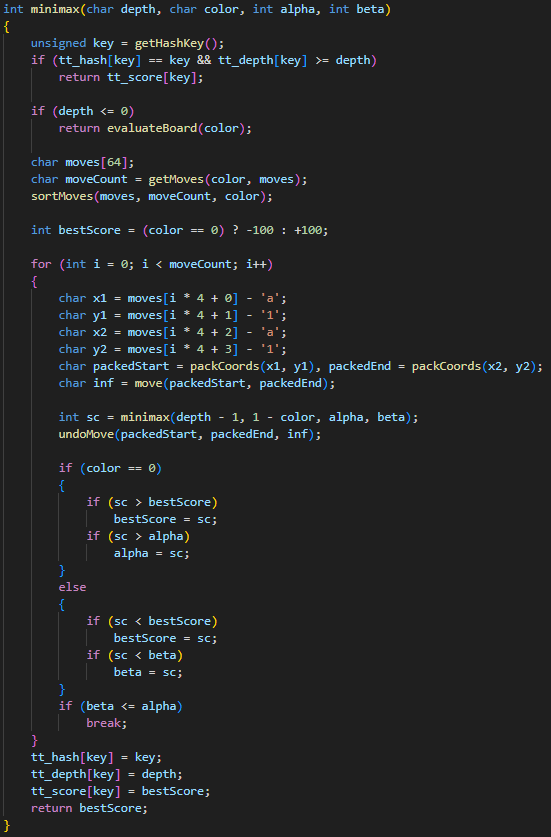
You need scoreMove to evaluate a certain move



sortMoves sorts our moves in descending order of their profitability for our bot. This is necessary for our algorithm to view the most profitable moves from the beginning. For sorting, use bubble sorting.

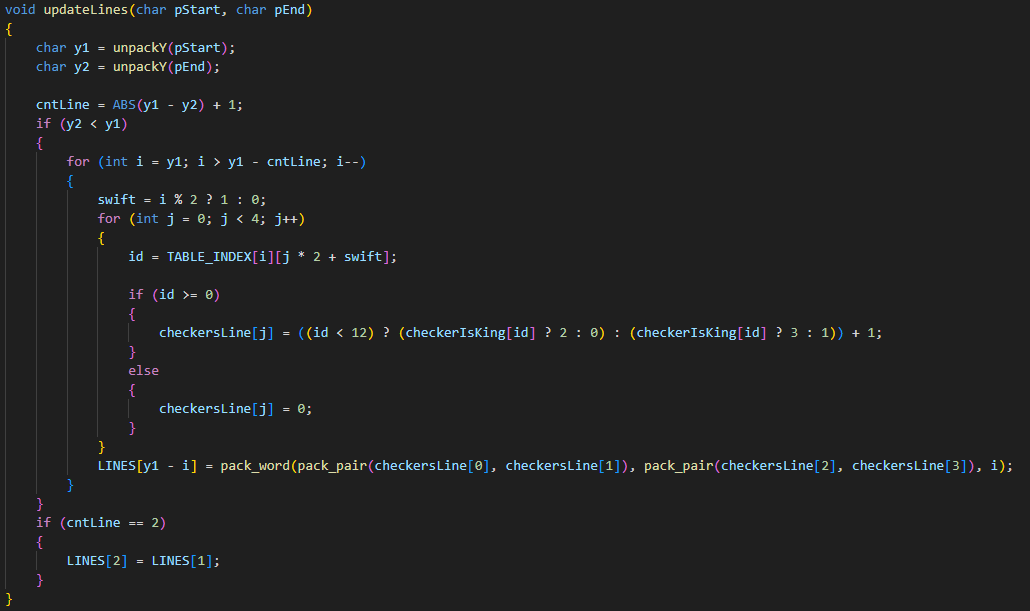


Our main function minimax рrecursively examines moves up to the depth we set. To do this, it пselects possible moves by making a move and then undoMove. We used alpha-beta cutoff to in crease the rate of miscalculation. The result is also stored in the Hash table to avoid repeated miscalculation.



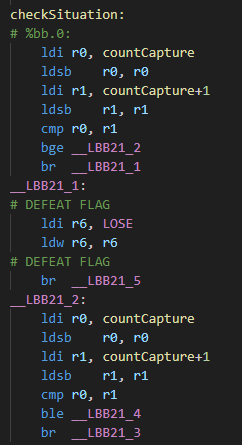
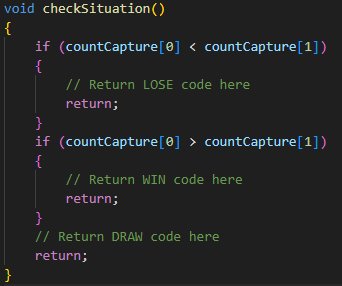
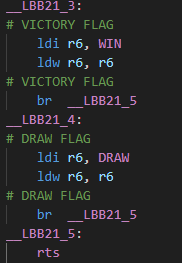
## **Output from the processor**

To update the playing field, we output data from the processor, which is packed with functions that we wrote about above. The updateLines function updateLines analyzes the entire playing field in the processor and collects the necessary data for output.



## **Checking the situation on the field when moves are deadlocked**

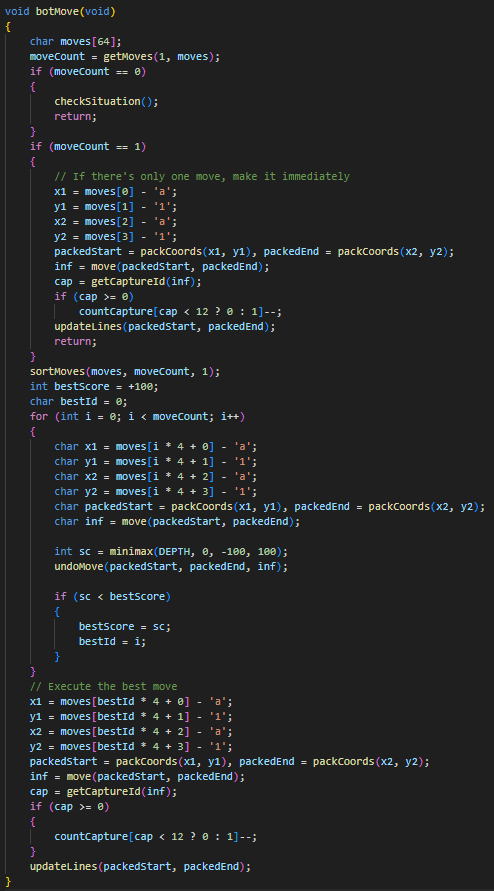
In a situation where the bot cannot descend, the checkSituation function is called, which C is not added in the C code, since we write the rest of the code, and that is, we call the flags for the hardware part directly in assembly

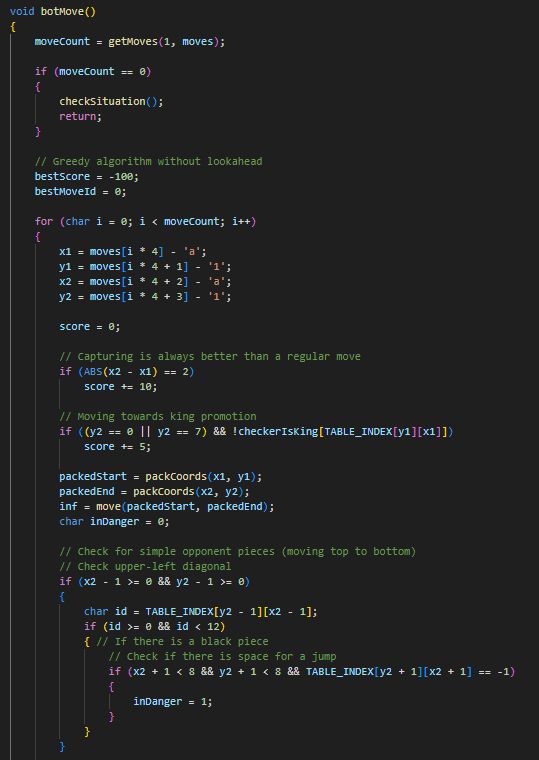
## **Bot progress**

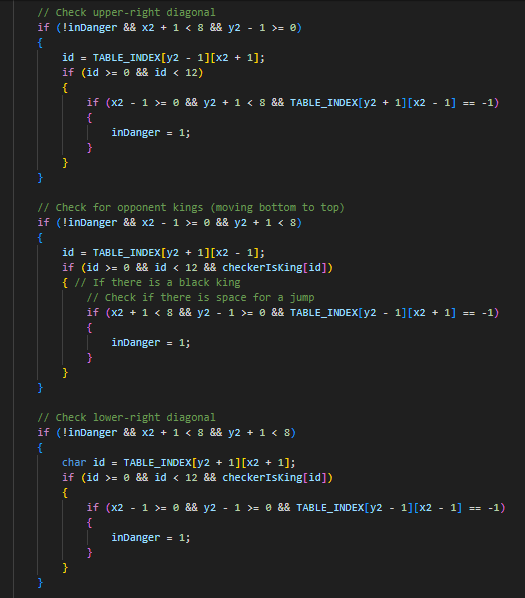
Function for making the most profitable move by the bot. It differs in two implementations, but both use previously implemented functions.

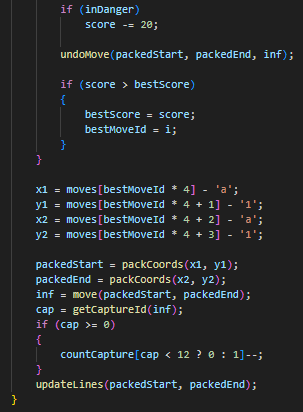
In the implementation with MiniMax:



In the implementation without MiniMax:

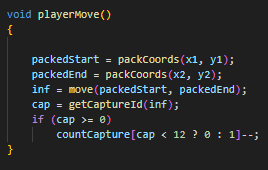






## **Player's Turn**

Packs data and runs the move function and updates global variables by the number of checkers.

****

## **Game cycle**

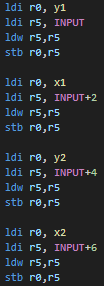
****

## **Input to the processor**

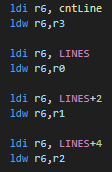
Implemented via the INPUT address

****

Input

****

Output to the hardware



Example of raising a flag



# **Conclusion**

The result of our project is a game of Checkers in Logisim using CDM-16. During the production process, we used the knowledge gained during the course "Digital Platforms", as well as we studied the history of the game and the differences between the English version and the rest, implemented an algorithm for calculating the most profitable move for the bot. Thanks to this project, we have developed our teamwork skills and shared team responsibilities, as well as gained experience in circuit design and game design.