**Introduction**

In this project, we are to implement the English version of the game “Checkers” in Logisim. The first mentions of Checkers date back to around 1600 B.C. in Ancient Egypt. The modern version of English draughts was created in Great Britain in the 16th–17th centuries, when it was adapted to an 8 × 8 board. The game then spread widely in North America, especially in the United States, where it became the national variant of draughts. The first official tournaments appeared in the 19th century. With the advent of software simulators, it gained popularity among both professional and amateur players. In our project we use a display to show the checkered board and piece positions, as well as digital circuits to manage moves. During development we used Logisim to create the digital circuits and the CDM-16 processor (via a Java-based emulator) to calculate the bot’s moves.

* We will use an 8 × 8 display, each square measuring 5 × 5 pixels.
* We will create a bot capable of calculating favorable move variations.
* We will implement the logic for removing a piece from the board.

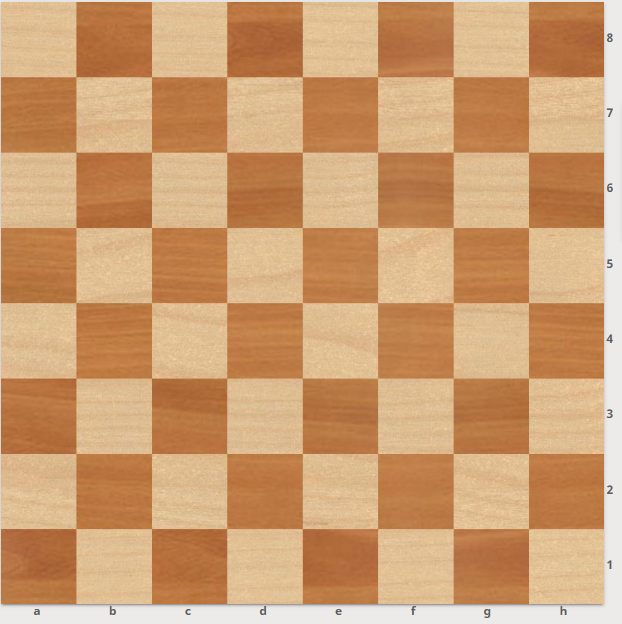
Each square includes the necessary electronics to display the piece on it. A piece is removed from the board when it is “captured” by an opponent’s piece. The goal of the game is to capture all the opponent’s pieces or to deprive them of any legal moves. Direct analogues of our game are any implementations of Checkers with rules differing from ours.**Task Statement**

The idea to implement Checkers came from our interest in creating a game with an opponent you can play against. After discussion and exploration of suitable options, we concluded that the most interesting to implement and use would be Checkers. The main reason for this choice was the requirement for a fairly high level of bot intelligence, which would allow both beginners and experienced players to play comfortably. Inspired by existing popular versions of Checkers, we decided to create two levels of bot intelligence, because:

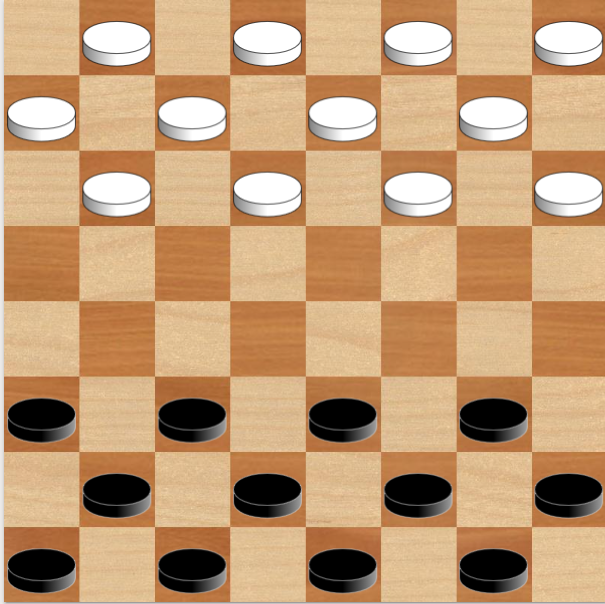
1. This will allow for two bots with different move-calculation times—so users can opt for faster games if they wish.
2. It will give experienced players the chance to play against a smarter opponent, at the cost of longer wait times per move.
3. One bot will have a very short calculation time, making it convenient for demonstrating the overall functionality of our game.

**Rules:**

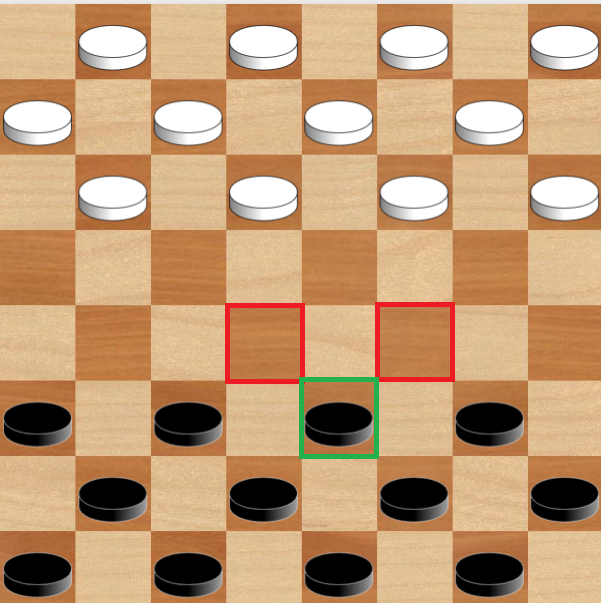
1. The playing field is an 8 × 8 board with alternating colored squares.



1. Pieces are placed on the first three rows at each end of the board.



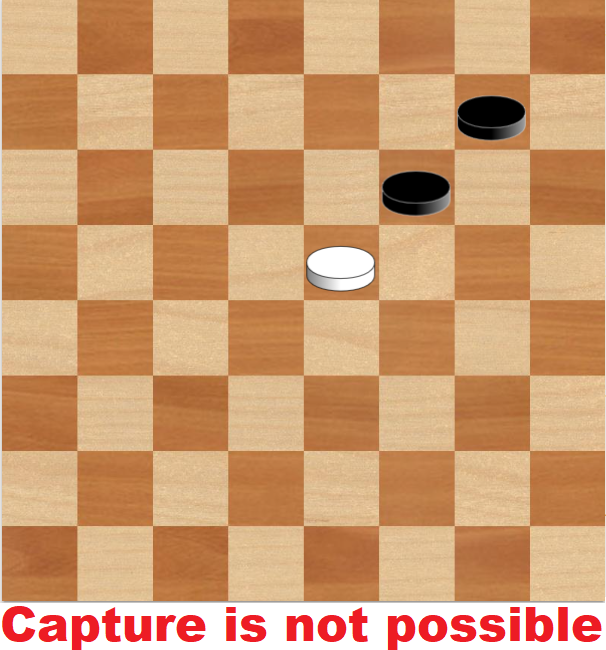
1. Players alternate moves, moving only along the diagonals; ordinary men cannot move backward.



1. Pieces move only on the dark squares.
2. To “capture,” the square immediately beyond the opponent’s piece must be empty.

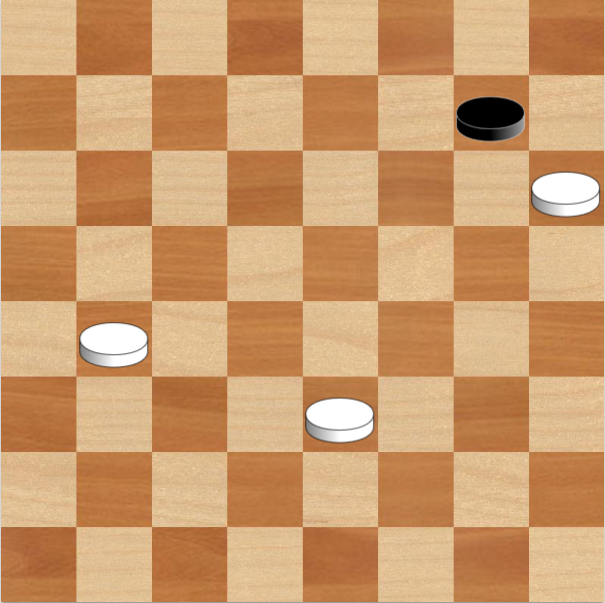
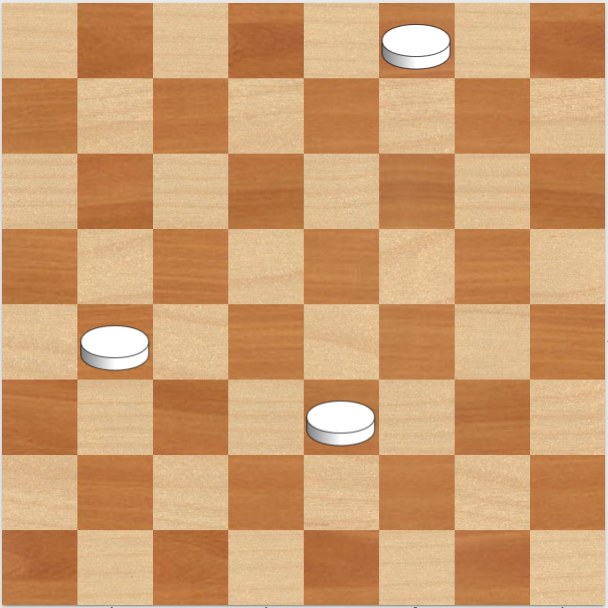
In the examples below, it is White’s turn.





1. The objective is to eliminate all the opponent’s pieces or to block them so they have no legal moves.

Elimination of all opponent pieces (White wins).

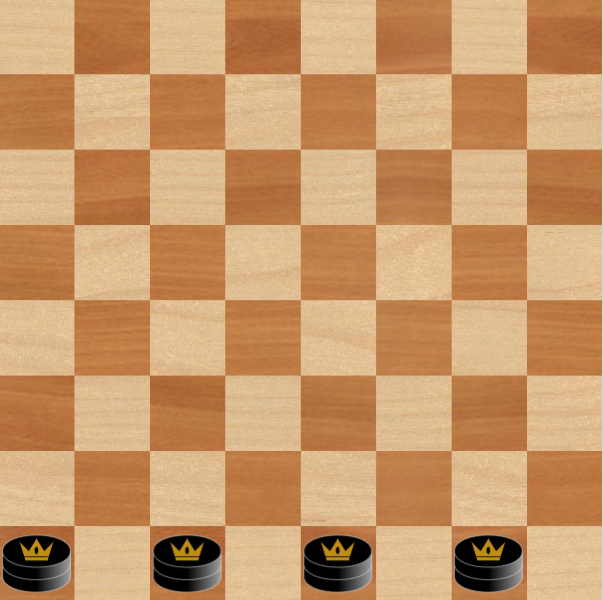
 

Blocking all opponent moves (White wins)

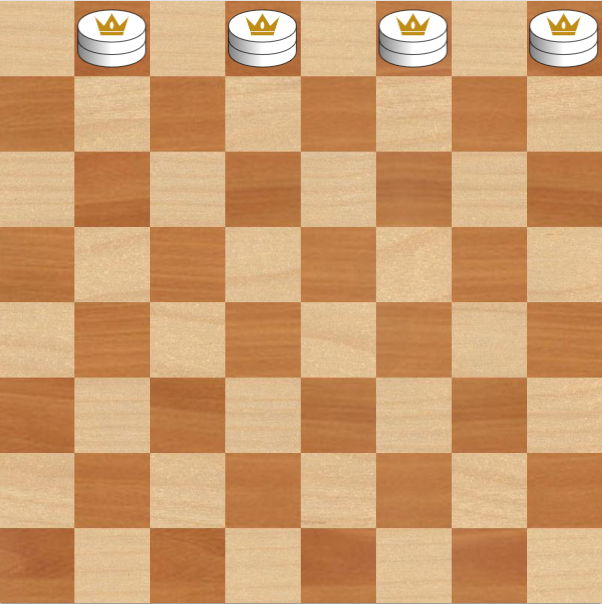


1. A man that reaches the farthest row of the board is promoted to a “King.”

Promotion squares for Black:



Similarly for White:



1. If a piece can capture an opponent’s piece on its turn, it must do so. If, after a capture, another capture is possible, the player must continue capturing, but if there are multiple capture sequences, the player may choose which to follow—not necessarily the longest one.



This case is interesting because during forced captures our piece may become a king and thereby must continue capturing.

1. Once promoted to a King, a piece may move one square in any diagonal direction.



**Work Plan:**

1. Determine which functionality will be handled by hardware and which by software
2. Implement the hardware part
3. Write the software part
4. Integrate hardware and software
5. Test functionality and make corrections
6. Optimize

**Requirements**

**Game Board**

* An 8 × 8 display, each square 5 × 5 pixels.
* Squares have the following states:

1. Empty white square.
2. Empty black square.
3. Black man.
4. White man.
5. Black king.
6. White king.

**Data Input**

Coordinates of the start and end squares of a move are entered via keys, using standard algebraic notation (e.g., B4 refers to column B, row 4). A confirmation button submits the move.

**Processor Tasks:**

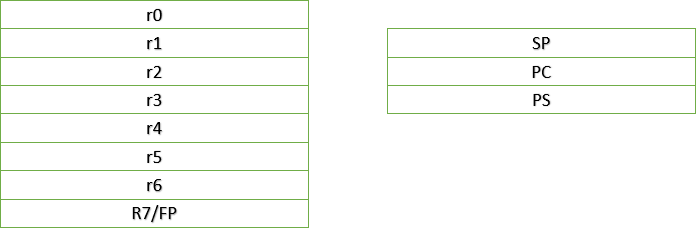
1. Calculate optimal moves for the opponent..
2. Store the current board state.
3. Process player input.
4. Update the board state
5. Determine the winner.

**Архитектура процессора:**

We use a von Neumann–architecture CDM-16 processor, chosen because Clang targets its format (single RAM) and does not require Harvard-style instructions, and because it handles memory (especially ROM access) well.

Specifications:

* 16-bit processor with a basic instruction set
* 11 accessible 16-bit registers:
  + 8 general-purpose registers r0…r6 + frame pointer FP (r7)
  + 3 special registers (SP, PC, PS)



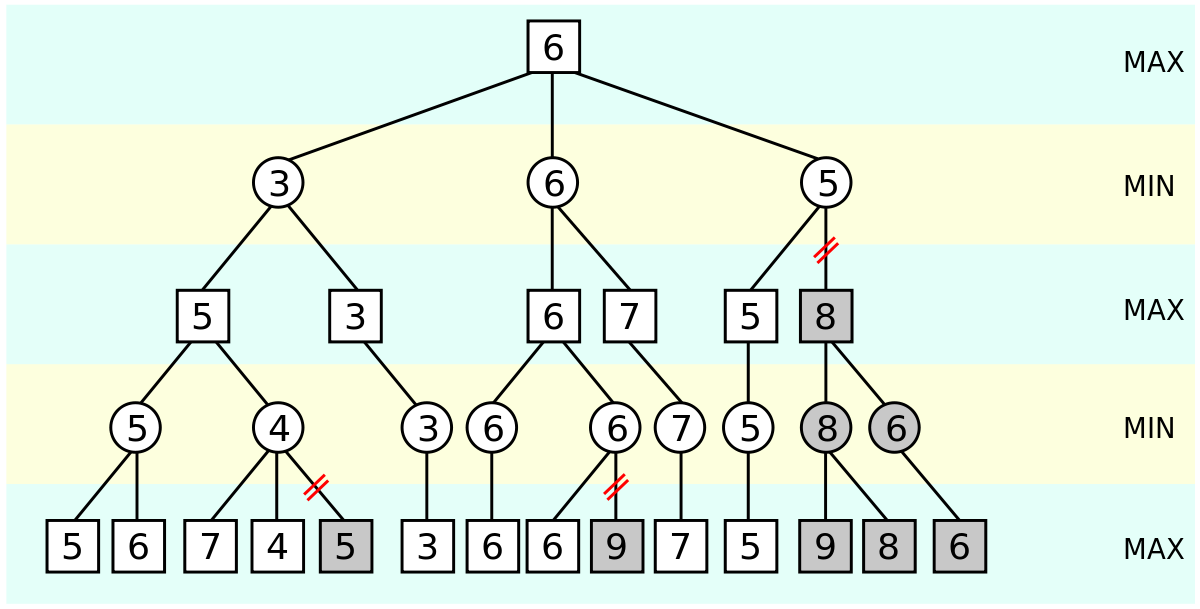
* Three-address instructions
* Hardware 16-bit arithmetic
* 64 KB memory

**Bot Intelligence**

The core algorithm of our project is Minimax. Its task is to find the most advantageous move for our bot.

To do this, it recursively explores possible moves of each piece. During the recursive descent, it evaluates the benefit of each move, producing a score: the higher the score, the more advantageous for the bot; the lower, the more advantageous for the human player. These scores are compared and the maximum (best) is chosen. Speed is crucial to ensure comfortable gameplay.

To increase the speed of our Minimax, we decided to implement alpha–beta pruning. The essence of this optimization is that, when evaluating a branch, if we discover that a previously examined situation is already better for us, there is no point in exploring the current branch further.



.