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): Software Engineering and Computer Science

TERM PAPER

Authors

Anton Tsoy, Michael Kozoliy, Vyacheslav Vetrov

Job topic:

"Mancala Game on CDM-8 assembly and Logisim**"**

Table of contents

Introduction

Problem statement

Our goal for this university project was to create a Mancala game using CDM-8 assembly and Logisim. We had to implement three basic circuits and a primitive strategy for an AI that would confront the player. The detailed description of this game is in the document “Co-design Group Project C - The Game of Mancala” [1].

Mancala Rules

First, from 4 to 8 stones are placed in each pit. In total, there are 48-96 stones. Players take turns to make a move.

A move consists of taking all the stones from one of the player's non-empty pits and distributing them counter clockwise. One stone is dropped into each pit or the player's own store along the way, but the opponent's store is bypassed.

[Capture rule] If the last stone falls into the player's own empty pit, and the opposite pit is not empty, then that stone and all the stones from the opposite pit are captured and transferred to the player's store.

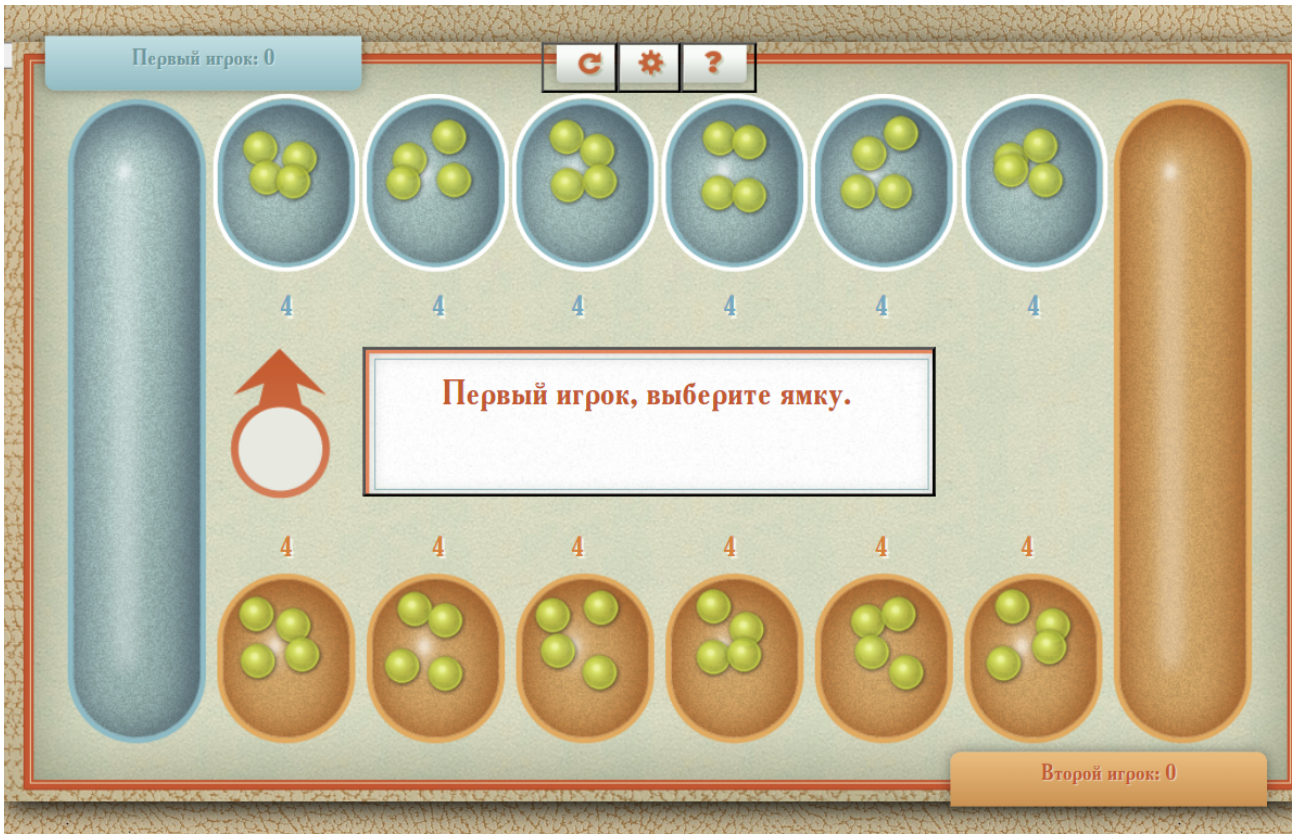
[Repeat rule] If the last stone at the end of a turn falls into the player's store, the player gets an extra turn.

The game stops when one of the players has no seeds left or is unable to make a move. If the number of seeds in their stores is equal, it is a draw; more seeds indicate a victory, and fewer seeds indicate a loss.

Analogues

Mancala is an ancient game with a rich history, dating back to ancient times, possibly before the 3rd century. The game was created in Ancient Egypt, making it one of the oldest known games that is still widely played today. Nowadays there are numerous board and online versions of this game.

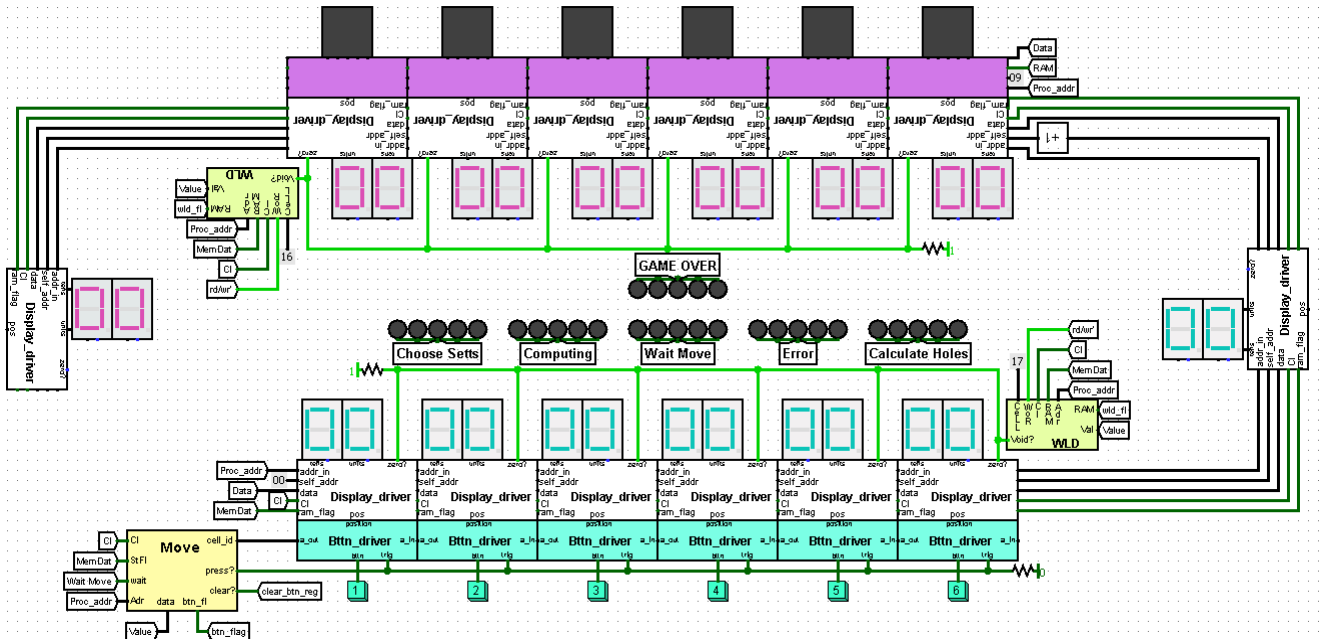
Here is an example of online mancala game [2]:



Picture A: Online mancala game example

Hardware

Game field description

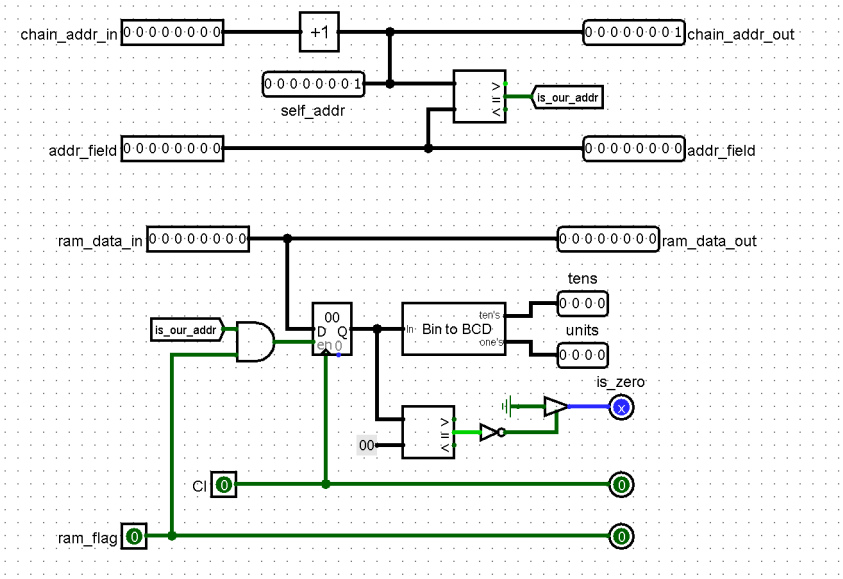


Picture 2: Game field

This is the appearance of our game field. The user controls the game by pressing buttons on the gamepad. As a result, circuits respond to them and change the state of pits in the game / send data about the current stage to the Cdm-8.

Logisim components

Display-Driver

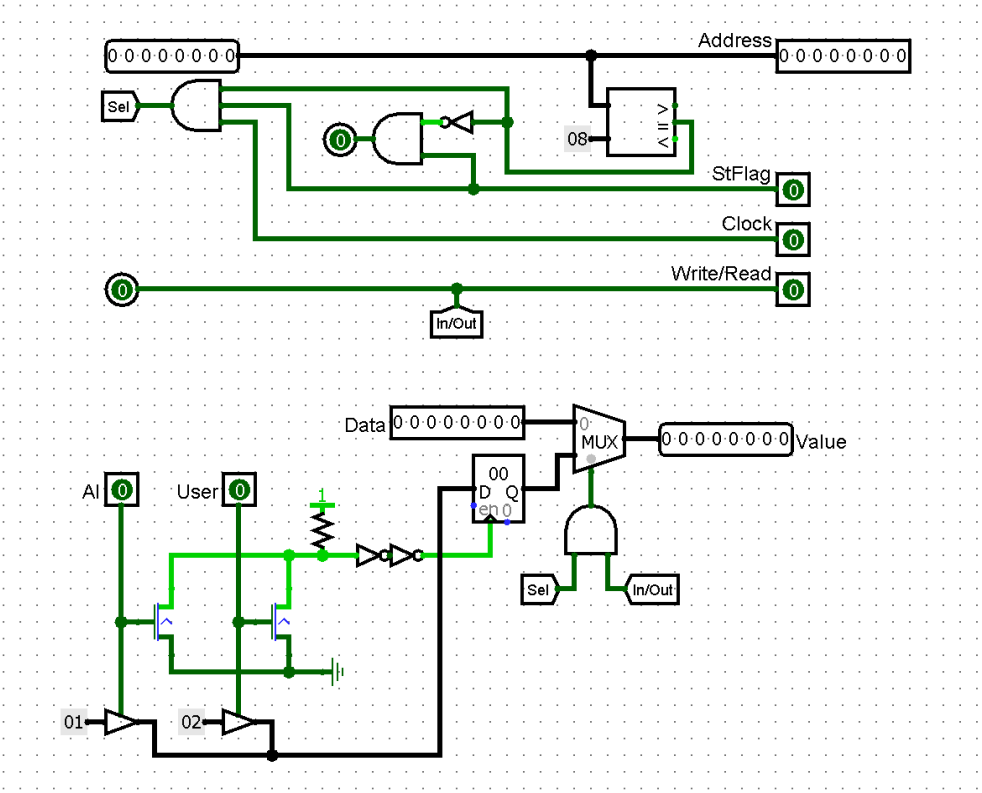


Picture 3: Display driver

One of the main circuits in our project is the display driver. This circuit reads a value from the corresponding RAM cell, which is responsible for storing information about how many stones are in a given hole, and outputs this value to the indicator in BCD format. Internally it contains a register with the value of the stones for each specific hole it displays. On the left or west side, for the driver, all the necessary parameters, some of which it processes and transmits to the right or east side, except for the value of stones in the hole and a Boolean value about if the value of stones is zero, the next chip takes these values and processes them. The values are then passed down the chain across the entire chip. More precisely, each chip inputs the value of the previous display driver, which is incremented and passed on. The number of stones is also received and output to the north side using the Bin_To_Bcd converter, which divides the value into tens and ones, its operation will be described below. Also, a zero stone check is performed internally and the result is also output to the north side. The zero stone check is necessary to understand the completion of the game because a move from this hole cannot be made later on.

Name	Description
addr_field	Memory address, 8 bit. Input and Output, west east sides.
chain_addr_in/chain_addr_out	Position in the chain, 8 bit
ram_data_in/ram_data_out	Data on the number of stones, 8 bit
Cl	Clock, trigger for writing into display register
ram_flag	Boolean value of RAM/ROM selection, 1 bit. Input, west side.
is_zero	Result of zero stone check, 1 bit. Output, north side
tens	Data on the number of tens of stones in a hole. Output, north side
units	Data on the number of units of in a hole, Output, north side

First Move



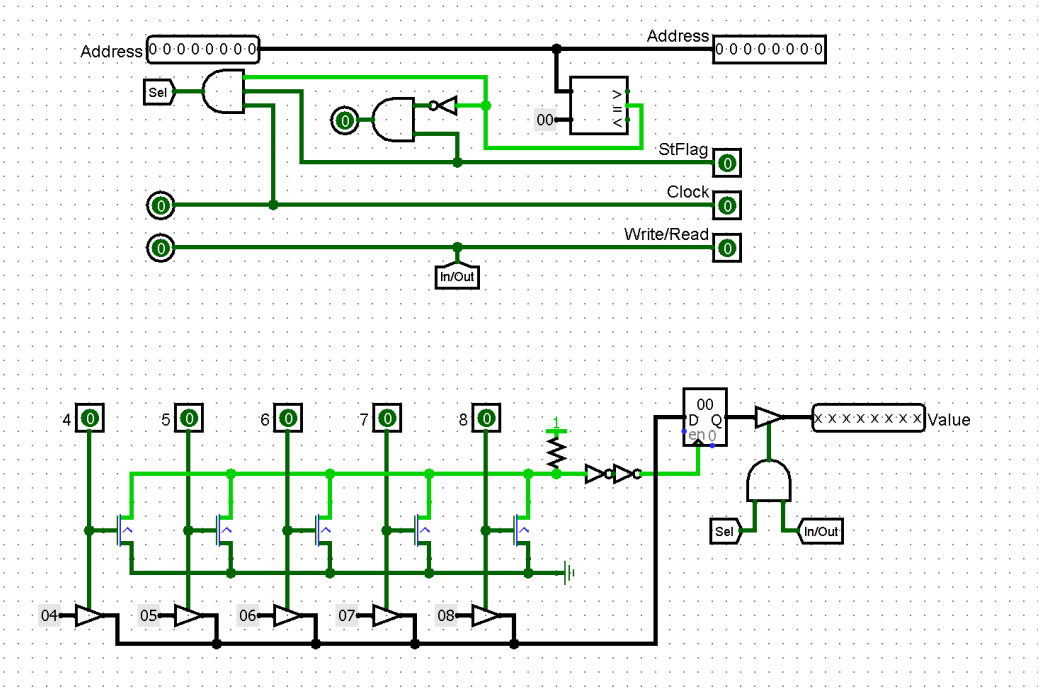
Picture 4: First_move

This circuit, which is not in Alex Shafarenko's documentation because it is one of those chips that our team developed the idea and implementation of. It is a system for entering values into memory, with a choice of who makes the move first. We felt that for Mancala, as for any other game, the choice of who makes the first move is obvious; this chip implements it. It takes a value from the player using the buttons,

stores it in a register and then transfers it to memory, if the memory address if the memory address value is 0x8.

Name	Description
Address	Memory address, 8 bit. Input and Output, west east sides.
Data	Memory data, 8 bit. Input, west side.
Value	Value of who makes the move first, 8 bit. Output, east side.
In/Out . Write/Read	Boolean value of read/write selection, 1 bit. Input and Output, west and east side.
Clock	Trigger for writing into the display register, 1 bit. Input and Output, west and east side.
StFlag	Boolean value of RAM/ROM selection, 1 bit. Input, west side
AI/User	Who starts the game player or robot. Input from buttons, south side

Start Seeds



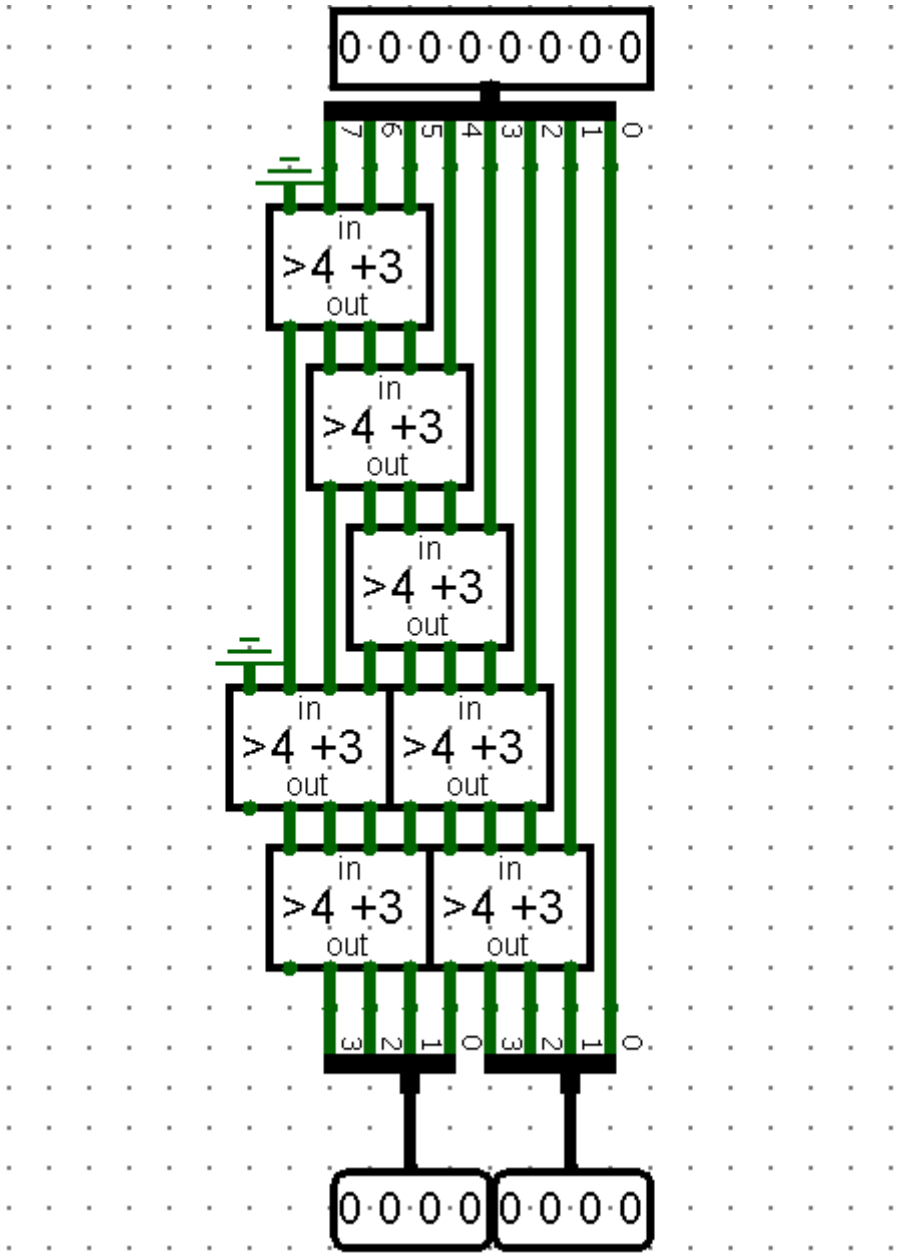
Picture 5: Start_Seeds

Also one of the schemes that was introduced into the game by our team, as the rules of the game do not say how many stones are in the holes at the start, it can vary. The idea behind this scheme is that the player can choose the number of starting stones, which allows you to change the duration of the game, where the fewer stones,

the shorter the duration of the game. The scheme itself allows the player to choose between 5 values from 4 to 8 stones, the player selects a value using the button, the value is stored in the register and if the memory address is 0x00 the number of stones is written to memory.

Name	Description
Address	Memory address, 8 bit. Input and Output, west east sides.
Value	Value of number of starting stones, 8 bit. Output, east side.
In/Out . Write/Read	Boolean value of read/write selection, 1 bit. Input and Output, west and east side.
Clock	Trigger for writing into the display register, 1 bit. Input and Output, west and east side.
StFlag	Boolean value of RAM/ROM selection, 1 bit. Input, west side
4, 5, 6, 7, 8	Buttons for number of starting stones. Input from buttons, south side

Bin To BCD

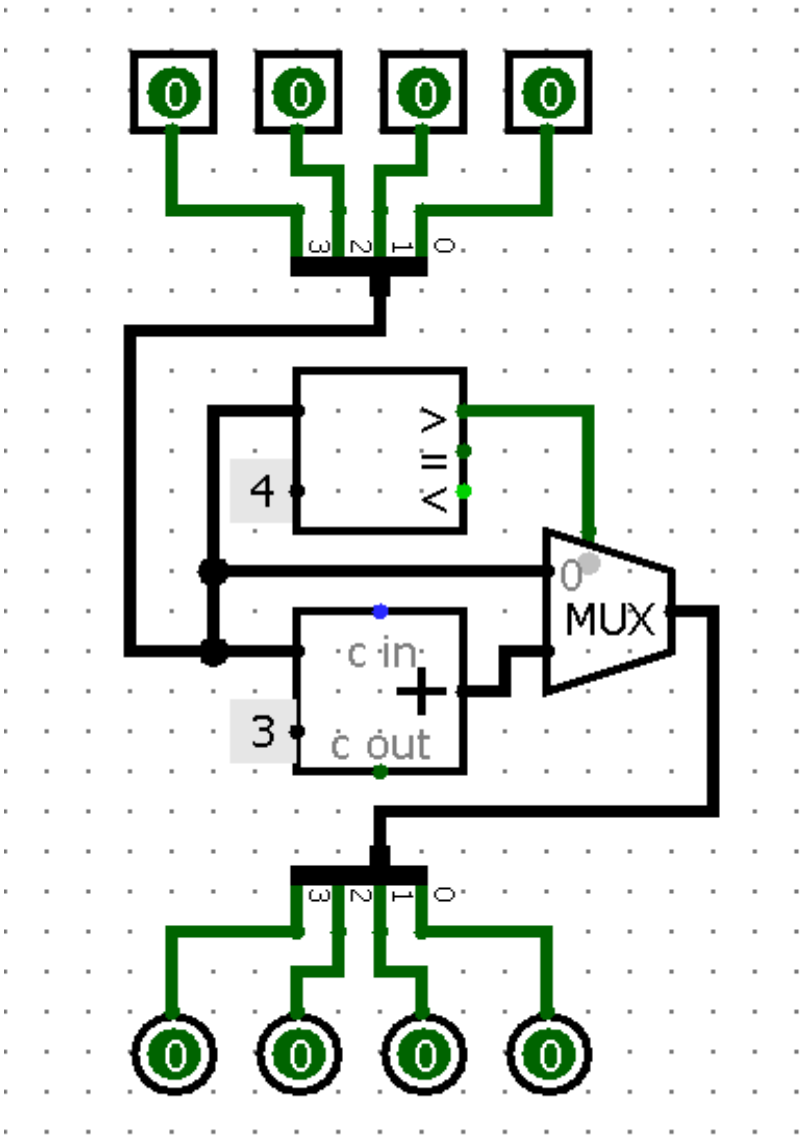


Picture 6: Bin_To_BCD

This scheme is used to convert binary numbers to binary coded decimal (BCD) notation. It uses the Double dabble algorithm, also known as shift-and-add-3 algorithm, and can be implemented with just 3 simple Logisim elements: comparator, adder and multiplexer, but the disadvantage of this algorithm is its high latency. The circuit takes values in binary notation, converts it into binary coded decimal notation (BCD) and splits it into two values: tens and ones, which are then output for display on hexadecimal displays.

Name	Description
Bin (Top)	Value in binary notation, 8 bit. Input, west side.
Tens (Bottom left)	Tens in BCD notation, 4 bit. Output, east side.
Units (Bottom right)	Units in BCD notation, 4 bit. Output, east side.

If more than 4 add 3 (> 4 + 3)



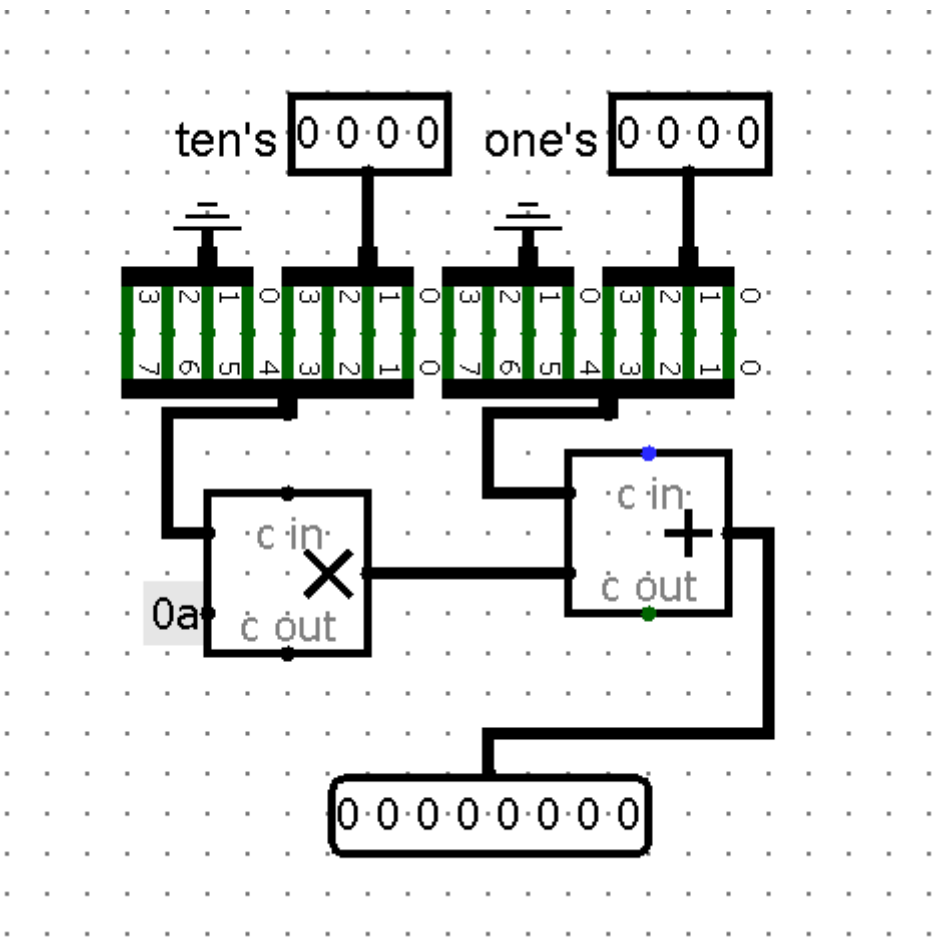
Picture 7: > 4 + 3

This circuit is necessary for the operation of the Binary Value Conversion chip in BCD notation, it also reduces the number of wires on the Bin_To_Bcd circuit and visually unloads it to make it easier to understand its operation. The circuit accepts a 4 bit value, checks whether it is greater than 4 using the comparator, if so, adds 3 to it

using the adder, if not, the value remains unchanged, the value selection is done using the multiplexer.

Name	Description
Input bins	North side
Output bins	South side

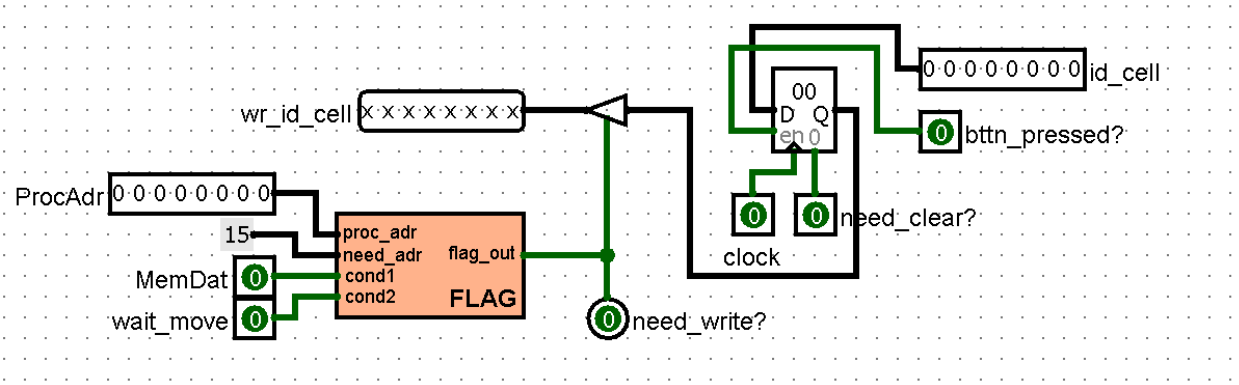
BCD to Bin



Picture 8: BCD_To_Bin

This circuit performs the inverse of the Bin_To_BCD circuit, it converts values from binary-coded decimal (BCD) notation and converts it to binary. It takes two 4-bit values, the values of tens and ones, multiplies the value of tens by 10 and adds the values of ones to them, returns the resulting 8-bit value as the result

User Move

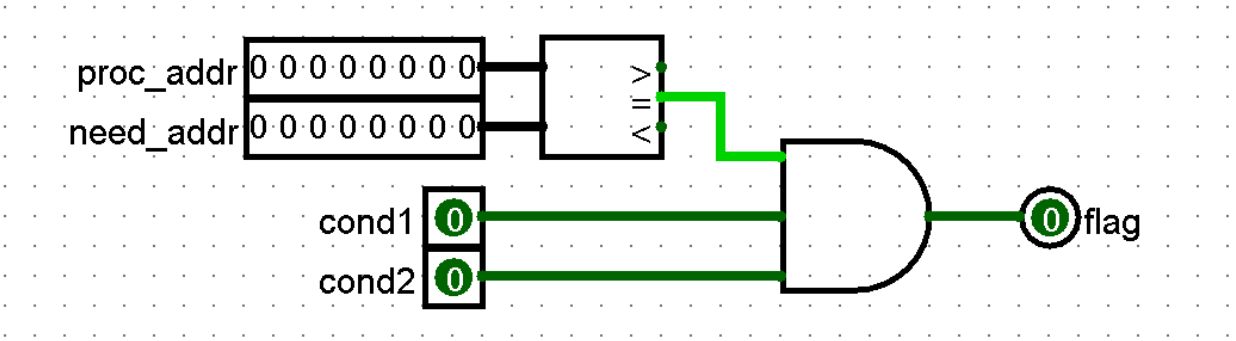


Picture 9: User_Move

One of the schemes, which was not in Alex Shafarenko's documentation, our team developed independently. This circuit determines whether the user pressed the button, if it was pressed, the number of the hole is transmitted on the CELL bus, and then this value is stored in the register. Then, at the appropriate time, which is determined by the scheme Flag_Base, the value that is stored in the register is sent to the processor. When the processor has written a value to memory, specifically to cell 0x15, the processor sends a signal and the register is cleared.

Name	**Description**
ProcAdr	Processor cell address, 8 bit
Id_cell/wr_id_ceil	The number of the cell chosen by the user, 8 bit
Need_write	True if the value needs to be written into memory, 1 bit
Clock	Trigger for writing into display register, 1 bit
Need_clear?	True, if the register needs to be cleared, 1bit
Bttn_pressed	True if the button is pressed
Wait_move	True, if processor is waiting for player to make move

Flag Base

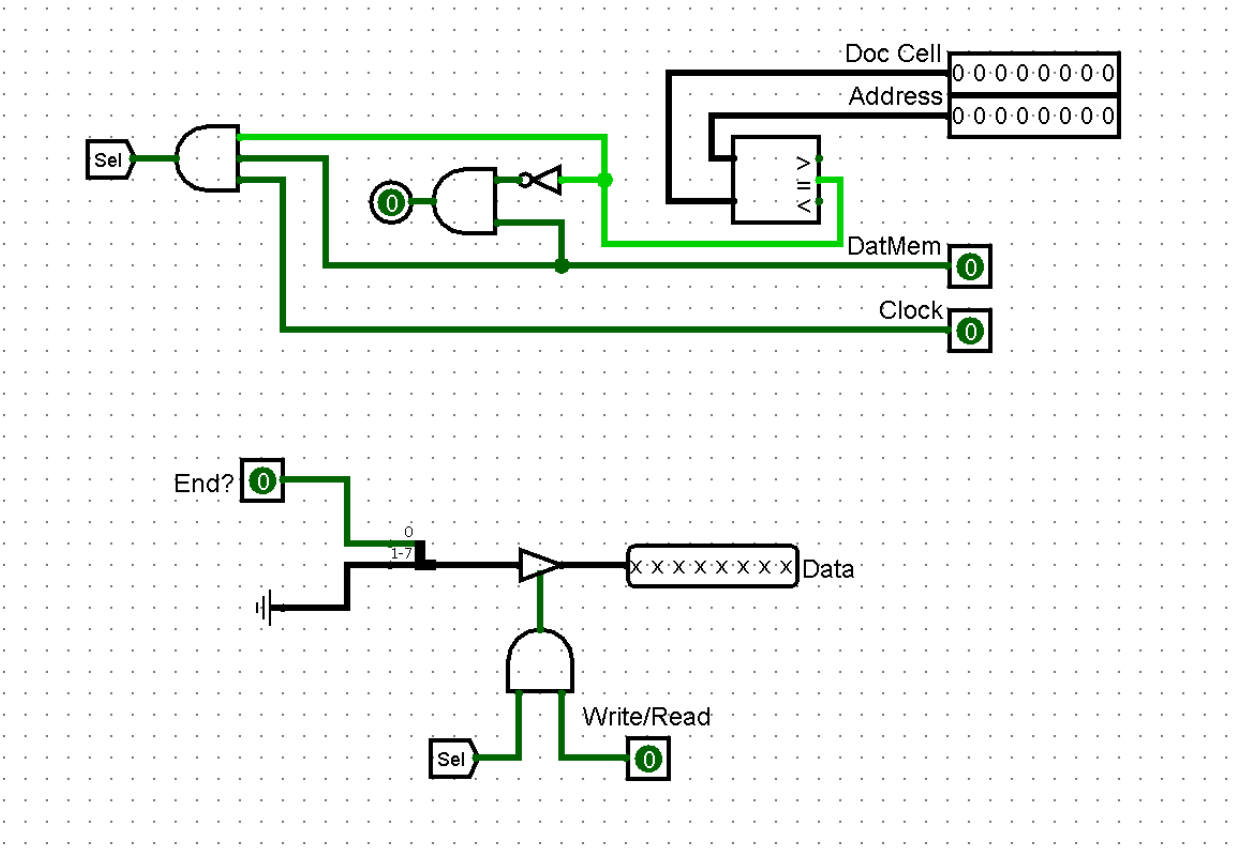


Picture 10: Flag_Base

This scheme is used by many others in our project. It compares two values: the current CPU address and the address where we want to store the value. Also, in order for the scheme to return true, two more conditions must be met, only if they are met at the same time will the scheme return true.

Name	Description
Proc_addr	Address of processor memory cell
Need_addr	Needed address
Cond1, cond2	2 conditions, if both of them are True, scheme can return True

WLD



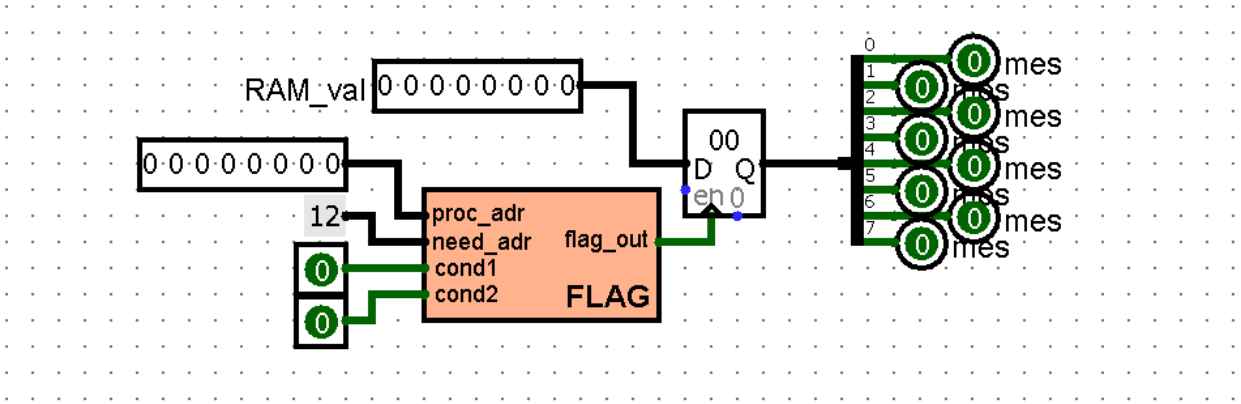
Picture 11: WLD

The chip shown above is needed to see if the game is over. Each memory cell has a role, 2 can be used in this scheme: the 0x17 memory cell is a predicate which is true if and only if all the player's holes are empty, and the 0x16 memory cell is a predicate which is true if and only if all the player's holes are empty. This scheme takes from the north side the address of the memory cell, the cell number, if the address value of the RAM memory cell is 17 or 16, and can be written to the processor, then the value of End? a parameter is written to it, which is taken from the west side and equals 1, when all holes of the robot are empty or when all holes of the player are empty. In other words, if all player's holes are empty, the memory address value is 0x17 and a write to memory is made, then the game considers that the player can no longer make a move, similarly for the robot and cell 0x16.

Name	Description
Doc Cell	Needed processor cell address(0x16 or 0x 17), 8 bit.
Address	Processor cell address, 8 bit
DatMem	Boolean value of RAM/ROM selection, 1 bit. Input, north side.

Clock	Trigger for writing into the display register, 1 bit. Input, north side.
Write/Read	Boolean value of read/write selection, 1 bit. Input, north side.
End?	True if all robot's or all player's holes are empty, 1 bit. Input, west side.
Data	Result. True if game is over. Output, east side.

Messages



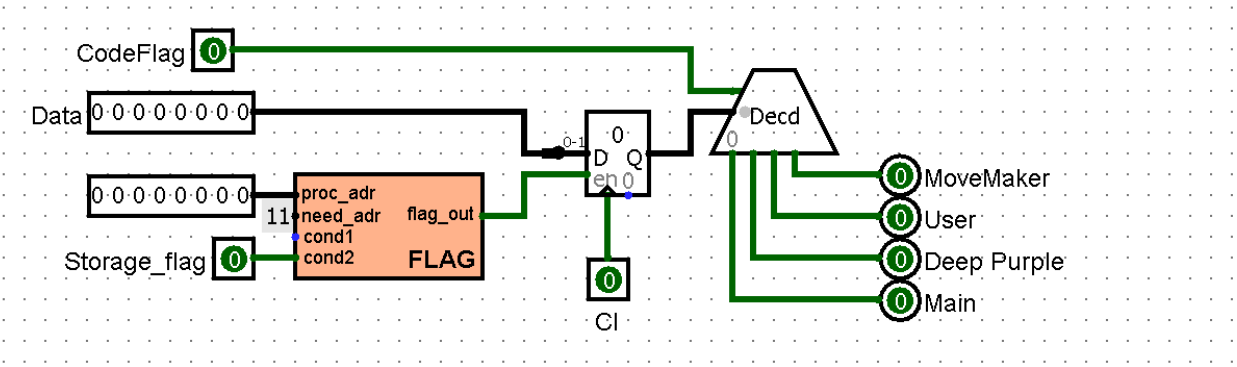
Picture 12: Messages

This circuit is necessary for displaying messages on the main screen. On the west side of the chip takes the value of the memory cell address and if it is equal to the address of the memory cell used for special values (0x12) and is now reading from the memory, then the register is written to the value of the cell 0x12. Then, depending on the value of the cell to the east side returns a signal for a particular message, depending on the bit of the cell, which is not equal to zero. Detailed description of 0x12 memory cell is provided in the **Memory planning** section.

Then the tunnel values are transmitted to the corresponding LEDs and if the value in the tunnel is 1 LEDs are lit.

Name	Description
RAM_val	Value of the cell in RAM memory
Proc_addr	Processor cell address, 8 bit. Input, west side
Need_adr	0x12, 8 bit. Input, west side
Cond1	RAM/ROM switcher, 1 bit. Input, west side
Cond2	Clock, trigger for writing into display register, 1 bit. Input, west side.
Write/Read	Boolean value of read/write selection
Output bit from 0 to 7	Messages(Error, clr_btn_reg, Choose seeds, etc.), 1 bit. Output, east side

Bank Switch



Picture 13: Bank_switch

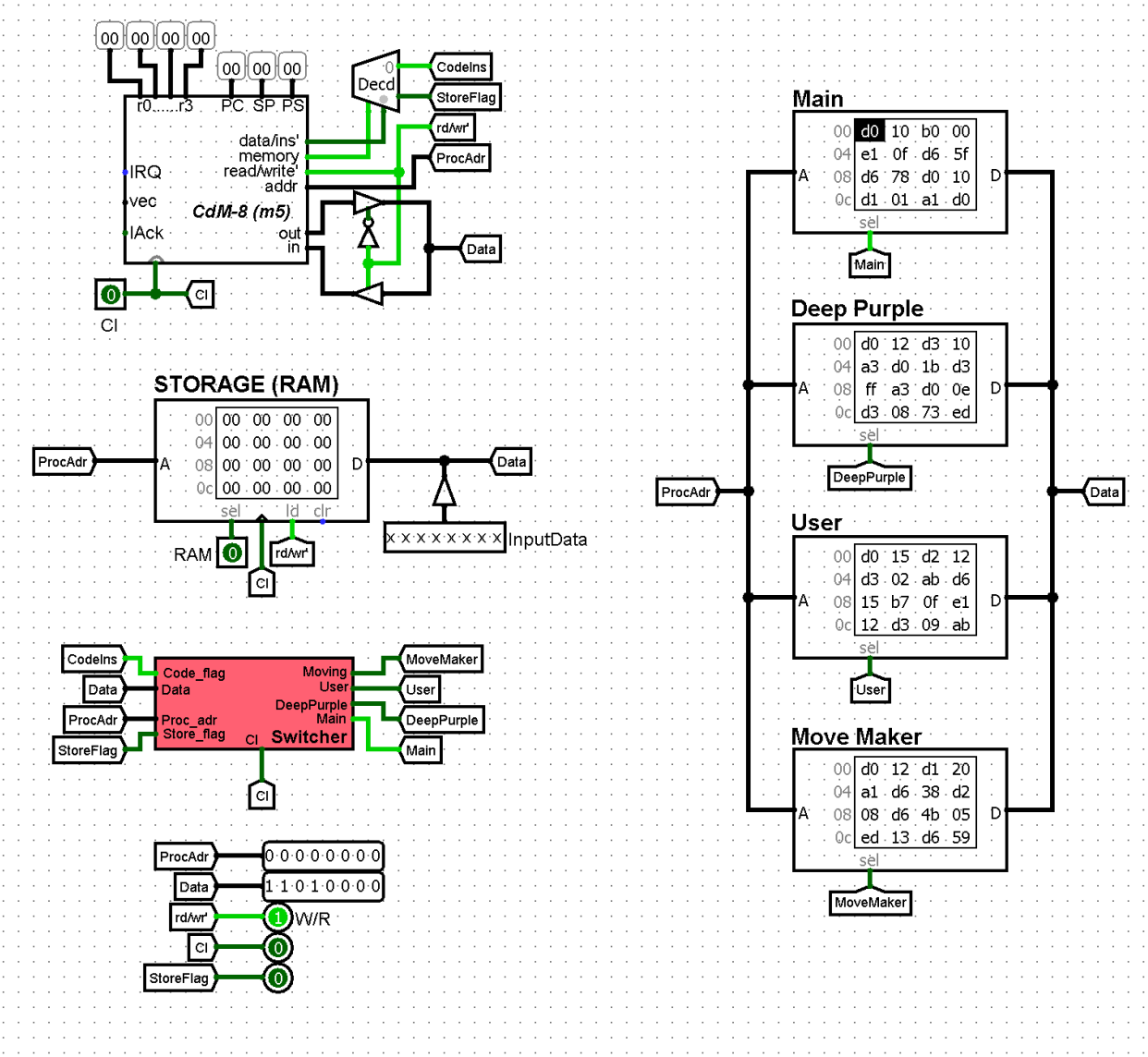
The memory banks also need an algorithm and circuit for switching them, which is what this chip is for. It receives from the west side the value of the memory cell of the processor and if it is 0x11 (the cell number of the memory bank) the value of this cell from 0 to 3 is written into the register, then with the decoder the corresponding memory bank is selected, the output of all values is from the east side. Values for each memory bank:

1. MoveMaker = 3 (0b11)
2. User = 2 (0b10)
3. Deep Purple = 1 (0b01)
4. Main = 0 (0b00)

(*Deep Purple is the name of our game AI. You can find the details in the paragraph **AI design**.)

Name	Description
Data	Value of the cell in RAM memory, 8 bit. Input, west side
Proc_addr	Processor cell address, 8 bit. Input, west side
Need_adr	0x11, 8 bit. Input, west side
Cond1	RAM/ROM switcher, 1 bit. Input, west side
Cl	Clock, trigger for writing into display register, 1 bit. Input, south side.
CodeFlag	Boolean value of RAM/ROM selection, 1 bit
Output bit from 0 to 3	Memory bank numbers, 1 bit. Output, east side

Game Operator

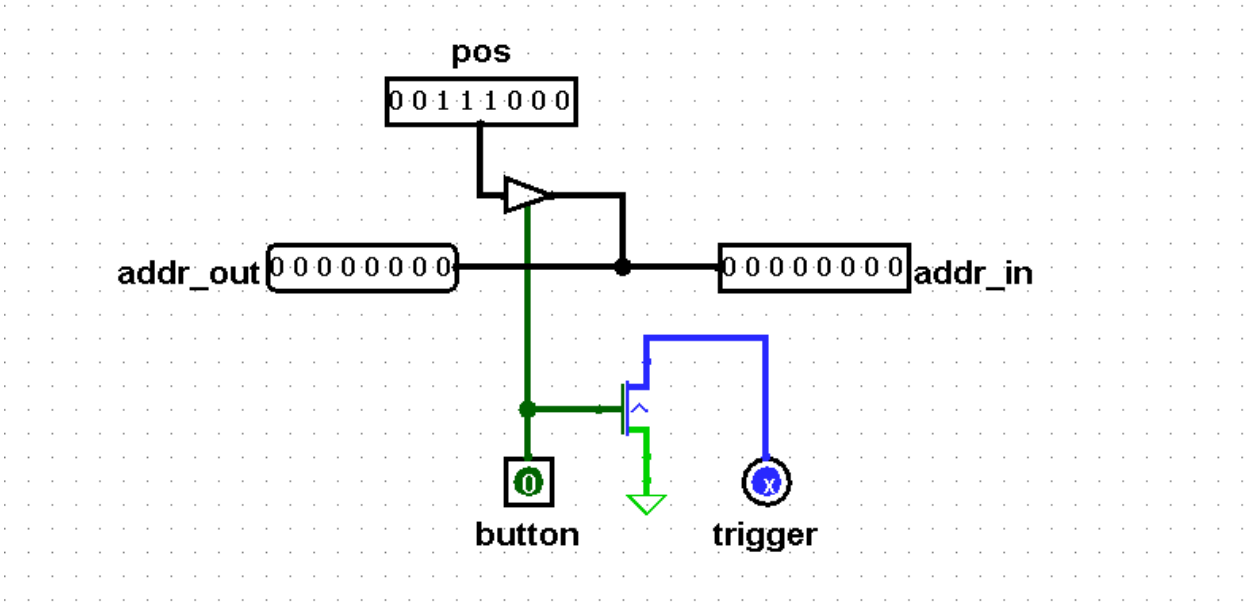


Picture 14: Game Operator

One of the key circuits of our entire project, its brain, which coordinates the work of many processes. On the input from the western side, it receives the beat values and uses them to build the project's work, the circuit returns the values to the game, and then the game processes the received values. Also, since the circuit contains the processor, and the game must affect the operation of the processor, the circuit receives two more values from the western side, namely RAM and Value. These two tunnels affect the operation of this circuit, memory and, primarily, the CPU. The Bank_switcher circuit described above, in the picture just Switcher, on the left side receives data and determines which memory bank should be used now, the Main memory bank is used on the program stratum. The work of all executable programs will be described later in the documentation, this scheme executes them as well as operates them and sends to the game

Name	Description
RAM	Ram address, 8 bit. Input, west side
ProcAdr	Processor cell address, 8 bit. Output, east side
StorageFlag	RAM/ROM switcher, 1 bit. Output, east side
W/R	Boolean value of read/write selection. Output, east side
Clock/Cl	Clock, trigger for writing into display register, 1 bit. Input and Output, west and east side.
Write/Read	Boolean value of read/write selection. Output, east side
Value	Values of memory

Button Driver

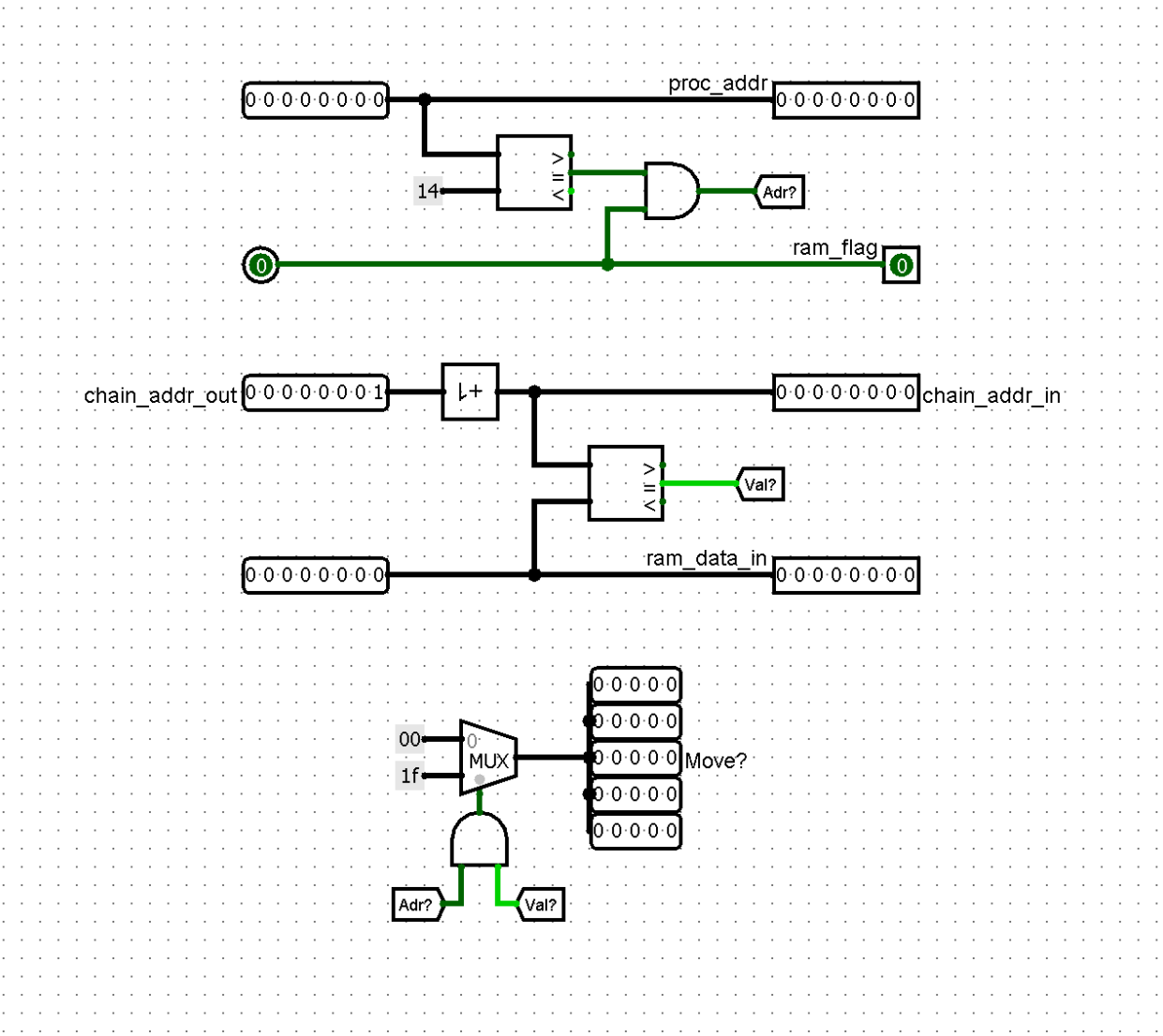


Picture 15: Button Driver

This circuit receives data from the north side, outputs it to the west side. Also the chip, when the button is pressed, grounds the trigger pin on the south side and transmits the value to the north side, if the button is not pressed, the value sent to the east side is transmitted unchanged.

Name	Description
Addr_in	Address bus for asserting pos on. Input, east side
pos	Chip position. Input, north side
Addr_out	Address bus for asserting pos on. Output, west side
Trigger	Trigger for latching button pos. Output, south side.
Button	Button terminal. Input, south side

AI move



Picture 16: AI_move

On the east side the circuit takes the address the processor is currently accessing, this circuit changes its behaviour if the address is 0x14, that is, the processor checks what move Deep Purple made. Also from the east side comes the RAM_flag - you need the RAM to work. Finally on the right side we get the number of the given robot hole and the value in cell 0x14 (i.e. the number of the hole Deep Purple came from). If the processor accessed cell 0x14 and the number of the given well was in it, the value is fed to the north side so that all the fields of the 5x5 matrix light up, otherwise zero is fed.

Name	**Description**
Proc_addr	Address of processor memory cell, 8 bit. Input and Output, east and west sides.
Ram_flag	Boolean value if the RAM is working, 1 bit. Input, east side.

Ram_data_in	The value of what cell Deep Purple chose, 8 bit. Input, east side.
Chain_addr_in	Game hole number, 8 bit. Input and Output, east and west sides.
Move?	Values that determine whether the matrix will light up for each hole separately, 5 bits. Outputs, north side,

Software

Memory planning

The memory planning for our Mancala game is carefully structured to efficiently store and manage the necessary data and control information. Here is a description of the memory cell roles in our game, using hexadecimal addresses:

- **0x00: User's initial stone selection** - This memory cell stores the user's input for selecting the initial number of stones.
- **0x01-0x07: User's pits** - These memory cells represent the pits or cups owned by the user. Each memory cell corresponds to a specific pit, allowing us to store and track the number of stones in each pit.
- **0x08: User turn selection** - This memory cell holds the user's choice of who will play first, determining the initial turn.
- **0x09-0x0F: AI's pits** - Similar to the user's pits, these memory cells store the number of stones in each pit owned by the AI.
- **0x10: Initialization predicate** - This memory cell acts as a predicate to indicate whether the memory has been properly initialized.
- **0x11: Memory bank number** - This cell holds the current memory bank number, allowing for efficient memory access and organization.
- **0x12: Special message cell** - The bits of this memory cell are used to convey specific messages or states during the game:
    - 1 bit - Error!
    - 2 bit - Clear button register
    - 3 bit - Please, choose settings
    - 4 bit - Wait user move
    - 5 bit - Deep Purple deciding...
    - 6 bit - Making move
    - 7 bit - Game over!
    - 8 bit is not used

- **0x13: Current player's turn** - This memory cell stores the value (1 or 2) representing whose turn it is to play.
- **0x14: AI's selected pit** - This memory cell stores the number of the pit selected by the AI for its move.
- **0x15: User's selected pit** - Similarly, this memory cell stores the number of the pit selected by the user for their move.
- **0x16: AI's pit emptiness predicate** - This memory cell acts as a predicate to indicate whether the AI's pits are empty.
- **0x17: User's pit emptiness predicate** - This memory cell acts as a predicate to indicate whether the user's pits are empty.
- **0x19: Predicate for enabling the Capture rule** - This memory cell acts as a predicate to determine if the Capture rule should be active on at least one pit.
- **0x1A: Predicate for requiring a move from a pit for the Repeat rule** - This memory cell acts as a predicate to determine if a move from a pit is required for the Repeat rule to trigger.
- **0x1B: Current maximum value for increasing the stones in the mancala** - This memory cell stores the current maximum value to which the number of stones in the mancala can be increased.
- **0x1C: Pit number from which the current maximum value was obtained** - This memory cell stores the pit number from which the current maximum value was obtained.
- **0x1D: Current increase in stones for the mancala** - This memory cell stores the current increment value for increasing the stones in the mancala.

AI design

Our team decided to give a name to game AI. His name is Deep Purple. This is a reference to the chess AI Deep Blue, which opposed Garry Kasparov in the 90s.

The AI design for the Mancala game involves determining the best move by considering various factors and applying a greedy strategy. The primary goal is to maximize the number of stones in the AI's Mancala (Deep Purple) after a move.

The AI evaluates each possible move by iterating through all the available pits (from pit 0x0E to pit 0x09). It starts a new iteration by selecting the next pit and loading the number of stones from that pit. This value helps determine if the initial conditions for the Capture and Repeat rules will be met.

For example, if there are more than 13 stones in a pit and the AI chooses to move from that pit, the Capture rule will not be triggered regardless of the values in

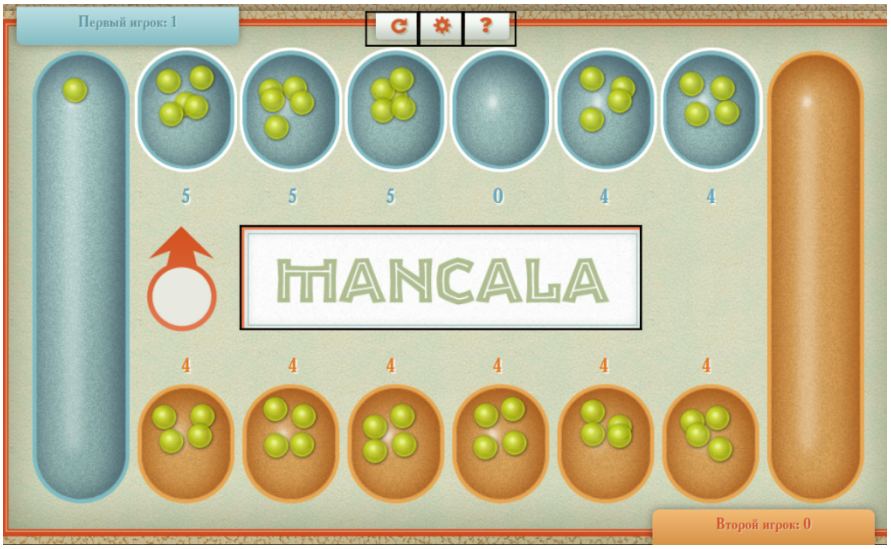
other pits. On the other hand, if there are exactly 13 stones in a pit, the Capture rule will always be triggered.

Another important thing is the distance from the current pit to the AI's Mancala. If the distance matches the number of stones in the pit, the Repeat rule comes into effect. If there are fewer stones in the pit than the distance to the Mancala, there is a possibility of triggering the Capture rule (although it's not guaranteed).

Manipulating these two values, along with checking the value of the AI's Mancala pit (where the last stone will fall) and the corresponding pit of the user (for Capture rule verification), helps determine how many stones the Deep Purple Mancala will increase by if it makes a move from that pit.

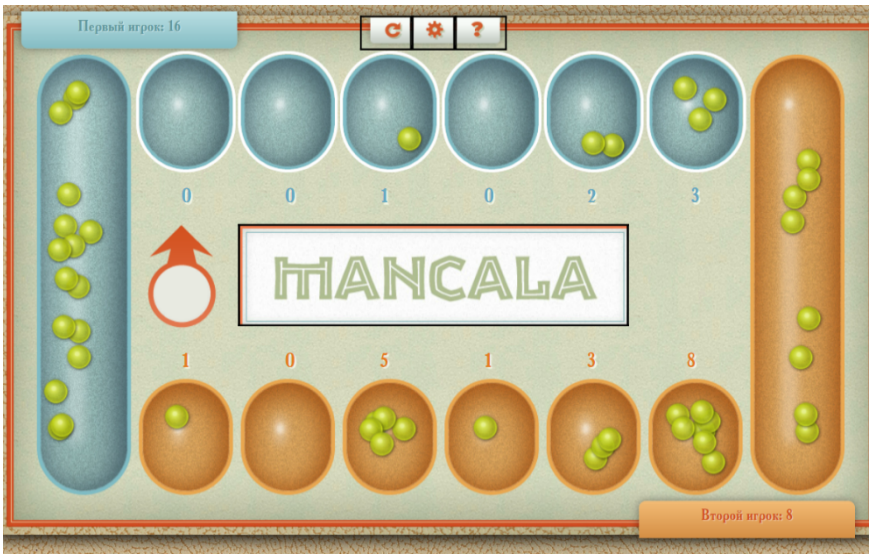
However, the AI's strategy goes beyond simply calculating the increase in the number of stones in the Mancala after a move. Additional conditions are introduced to prioritize move selection and identify the best move within a single move calculation.

First, it is essential to explain what happens when all possible moves are routine, meaning no special rules are triggered.



Picture B: Illustration of the game state 1

In this position, the AI (represented by the blue colour) chooses a move from the pit indicated by the arrow. Here, there are three different moves that add 1 stone to the Mancala (the other 2 moves don't add anything). In such situations, Deep Purple selects the pit closest to its Mancala. For example, in the position below, AI moves from the pit containing 1 stone (all moves are equivalent as they add exactly 0 stones to the Mancala, and no rules are triggered).



Picture C: Illustration of the game state 2

Now, let's discuss special cases. In our program, we have introduced a special flag to indicate whether a move leading to the Capture rule was encountered during previous iterations. If the flag is not set (indicating no such move occurred in the loop), and the current move triggers the Repeat rule, we immediately stop further iterations and make a move from the current pit. However, if the flag is set, we do not make a move from a pit triggering the Repeat rule and continue with the calculations. Finally, at the end, we make a move from the pit that results in the highest stone gain. This flag is entered so that moving from a Repeat rule pit does not interfere with moving from a Capture rule pit.

Software and hardware integration

Difficulties

During the creation of our Mancala game using CDM-8 assembly and Logisim, we faced several challenges. The problems we encountered are described below:

1. **Limited memory resources**: One major difficulty we faced was the limited availability of memory resources. As our assembly code grew in size, it exceeded the memory capacity we had. To overcome this challenge, we divided the code into four memory banks. Although only three banks were necessary, we decided to use an additional bank to improve the organization and readability of our assembly code. This approach allowed us to effectively manage the memory limitations while keeping our code structure clear.
2. **Selecting the best move for AI strategy between similar moves:** When there are several moves that did not immediately bring a win, choosing the best

move became a difficult task. While it's easy to identify a move that results in a significant gain, there are situations where all available moves offer no advantage. In such cases, we decided to prioritize the move closest to the Mancala pit among equally beneficial options. By freeing up pits adjacent to the Mancala pit, we increase the chances of utilizing the Repeat rule in subsequent moves.

3. **Limited potential for strategy improvement:** We had difficulty finding ways to improve our gameplay strategies because we could only analyse one move at a time. While we brainstormed ideas to enhance our approach, we realized that certain improvements needed a more thorough analysis that went beyond considering just the next move. It was hard to determine which enhancements were possible within the limitations of analysing only one move and which required more extensive calculations. This restricted our ability to make substantial improvements to our strategy given the constraints we had.

Integration

The integration phase of our project involved combining the hardware components implemented in Logisim with the software functionality developed in CDM-8 assembly language. One of the key aspects of this integration was storing useful variables in RAM, so they are accessible both from Logisim and CDM-8.

Here is a description of our banks of memory:

- **calculate_holes.asm**: The main function, "MAKE_MOVE," implements the distribution of stones from a selected pit to the other pits, following the rules of the game. This program implements the game logic of Mancala, including the stone distribution, capturing of stones, and passing the turn between players.
- **deep_purple.asm**: This program determines the optimal move for the AI player in our Mancala game. Detailed information about how it works is provided in the **AI design** section.
- **main.asm**: This program serves as the entry point for the Mancala game. It starts by waiting for user input, specifically the number of stones in a pit and which player will take the first turn. Once the input is received, the program initializes the game board. Also, after each player takes a turn, the program returns to main.asm to check conditions of game over.
- **user.asm:** This program is responsible for processing user input in the Mancala game and sending messages to the main game circuit in Logisim. The main function of this program handles the data about the player's move and communicates the current game state to the game circuit. Initially, a message is

sent to the game circuit to clear any accidental data that might be present in a special register responsible for storing the selected pit number. The program then calls a helper function that saves the number of the selected pit in register r1. It also checks the number of stones in the selected pit. If the pit is empty, indicating an invalid move, an error message is left in the game, and the helper function is called again. Once the player makes a valid move, the number of the selected pit is saved to a special address. The function then stops, and there is a switch to another bank of memory.

The uniqueness of our implementation

The uniqueness of our implementation lies in the incorporation of several innovative features that differ our Mancala game from the basic version. Firstly, we have introduced an artificial intelligence (AI) component, adding strategic depth. Additionally, users can customize the initial game setup by choosing the number of stones in each hole. Furthermore, the option to choose who takes the first turn adds a strategic element, enhancing replayability and allowing players to explore different tactics.

Conclusion

In conclusion, our team successfully developed a Mancala game using CDM-8 assembly and Logisim as our study project. Throughout the development process, we encountered several challenges due to the limited memory and processing speed inherent in the chosen platforms. However, we successfully implemented our unique, efficient and functional version of the Mancala game.

In comparison with the basic version, we improved the basic version of the project software, which are described below. Also we enhanced the visual part of the board and created our own unique logic schemes.

Source code can be found on GitHub [3].

Links

[1] Technical description document [Digital source] // drive.google.com

URL: https://kurl.ru/LsTCa

[2] Example of online Mancala game [Digital source] // ollgames.ru

URL: https://ollgames.ru/mancala/

[3] Mancala game project repository [Digital source] // github.com

URL: https://github.com/AntonTsoy/Mancala