

Co-design Group Project C

“The Game of Mancala”

22.05.2021

**─**

Dolgii Alexander, Guselnikova Alina, Medvedev Valdemar

# Rules of the game[[1]](#footnote-0)

The above is a computer rendition of the board for playing the classical game Mancala, taken from <http://play-mancala.com/>. It has 6 pits (or holes) on either side of the board and two stores, also known as “mancalas”

Here are the rules of the most common version:

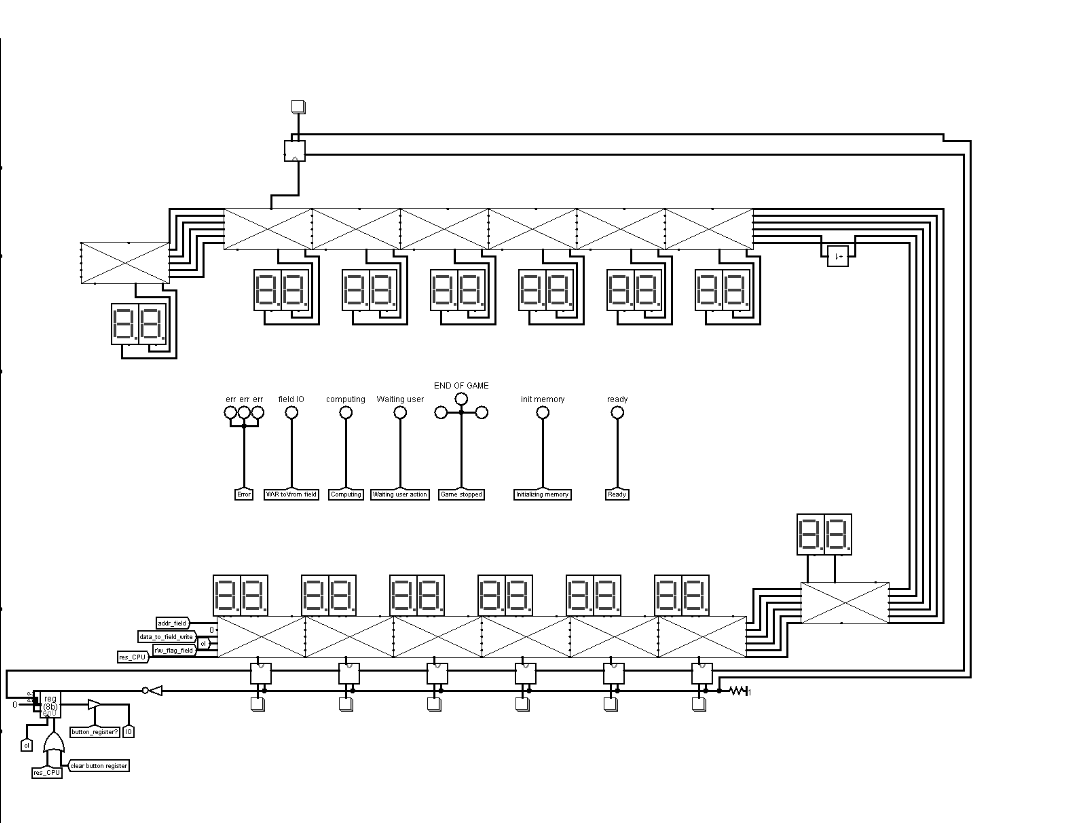
1. Two players play the game facing each other. A player’s holes are in front of the player and the player’s mancala is to his or her right. Initially four seeds are placed in each hole. That amounts to 48 seeds in total. (They could be the same or different colours, a mixture of seeds and pebbles –– it does not matter.) Players take turns to make a move.

2. The move consists in taking all the seeds out of one of the non empty holes that belong to the player and spreading them anticlockwise. One seed is dropped in each hole or the player’s mancala on the way, but the opponent’s mancala is bypassed.

3. [Capture rule] If the last seed of a move drops in the player’s own empty hole and the opposite hole is nonempty, that seed and all the seeds from the opposite hole are captured and transferred to the player’s mancala.

4. [Repeat rule] If the last seed of a move drops in the player’s mancala the player makes another move. The game stops when one of the players has no seeds/can’t make a move. If their mancala has 24 seeds, it is a draw, more: a win, less: a loss.

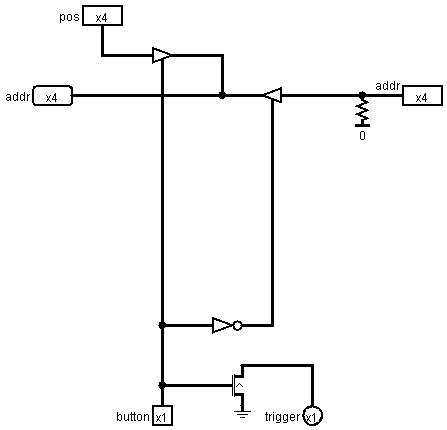
# Game Pad



The above is a Logisim implementation of a Mancala game pad. There are 8 components that helped us make the “Mancala” game.

**Display driver.[[2]](#footnote-1)** The criss-crossed boxes are display drivers: they have a register inside that holds the value to be displayed. The chip takes all the values it needs from the west side; it passes them over to the east side for the next chip in the chain. As we do in other projects, we maintain a cellular design here to avoid a jumble of wires. The chips are differentiated by a signal **pos** fed into the chip from the west (second pin from the top), which gets incremented before being passed over to the corresponding eastern pin. Here is a complete inventory of the west- and east-side pins, listed top down:

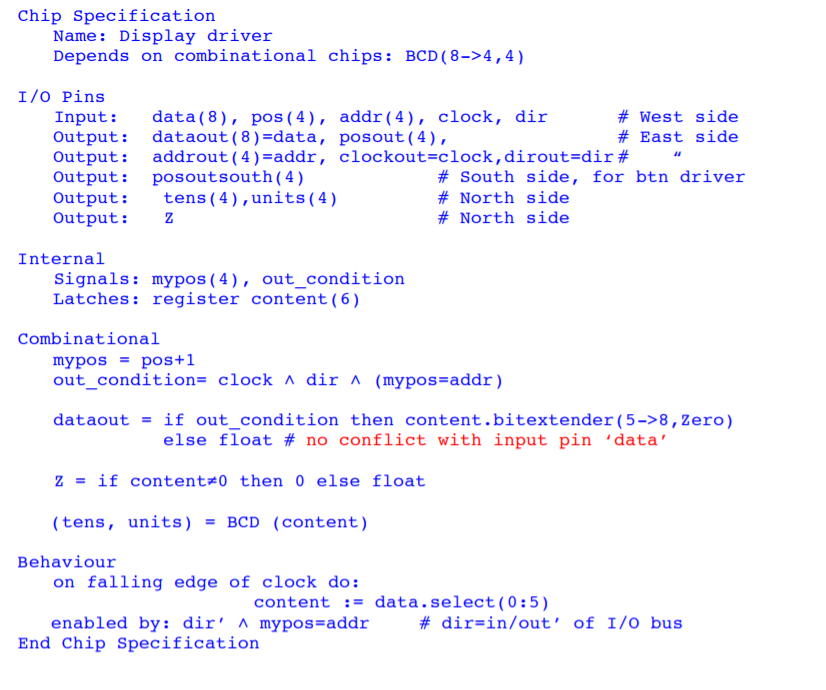
|  |  |
| --- | --- |
| Name | Description |
| data | Display data, 8 bit |
| pos | Chip position in chain, 4 bit |
| addr | Address (chip position) for read/write operations |
| clock | Trigger for writing into display register |
| in/out’ | Read/write selector |

Essentially the display driver is little more than an I/O register, similar to the one presented in the document “**Working with a full-core CdM-8 system**”. Here are the add-ons: 

The chip has two output pins on the north side, 4-bits each. Those drive hex displays (using only the decimal figures). The decimal to BCD conversion is done inside the chip (possibly by brute force, using Logisim’s own division chip).

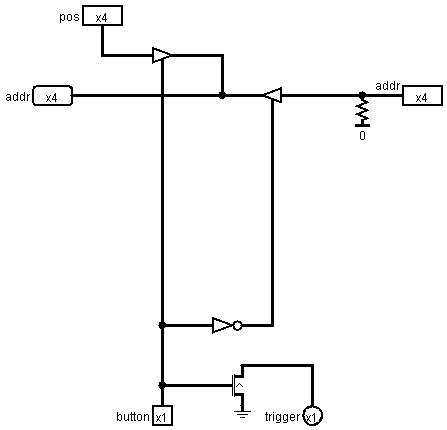
There is an additional output pin on the north side, Z, which is there to help detect that the player’s holes are all empty using wire-OR. That helps to establish that the game has ended. It can be done in software of course, but the reference implementation ran out of code memory . The Z pin is asserted with 0 if the register content is greater than 0 or left floating otherwise.

The display driver chip has a 4-bit pin on the south side, on which it asserts its position in the chain. This is required by the chip that is attached to the south edge of the display driver: the button driver, which we will consider next, but first here is the pseudo-code specification of the display driver:



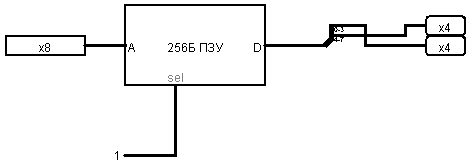
There is no conflict between the combinational result on **dataout** and the direct connection to **data** declared in the header because **out\_condition** is up only when the latter floats.

**Button driver.**[[3]](#footnote-2) This is a very small circuit.

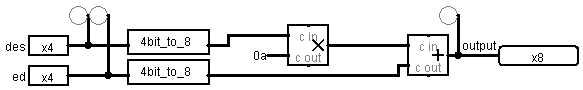


All it does is assert what it receives from the north on the west pin when the button is pressed (the west of the two pins on the south side is for connecting the button to). Also, when the button is pressed the chip grounds the trigger pin on its south side. The trigger pin is intended for participating in the wire-OR that delivers the trigger for latching the button code (in this case, the chip position). The document “Working with a full-core CdM-8 system” provides sufficient information about how buttons are interfaced with processing circuitry to make any further explanations here redundant. Here is the pin-out of the chip:

|  |  |
| --- | --- |
| Name | Description |
| pos | Input, north side, chip position |
| addr | Input, east side, address bus for asserting pos on |
| addr | Output, west side, address bus for asserting pos on |
| trigger | Output, south side, trigger for latching button pos |
| button | Input, south side, button terminal |

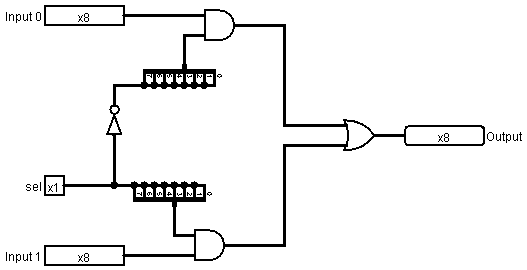
**Bin to Bcd Converter.**

Bin to Bcd Converter is used to display the numbers written in the display driver's internal register. In order not to create complex schemes for translating numbers, it was decided to take a bank of permanent memory and write there the bcd value for each binary number.

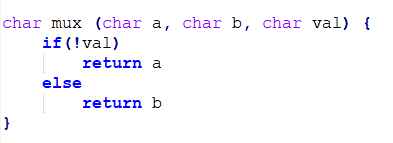
**BCD to BIN Converter.**

It followed from the documentation that a value can be read from a field. Since initially we did not understand that this functional unit would never be needed by anyone, a system for translating bcd into the usual bin format was implemented in the following way. Let's write a binary number in bcd format as xxxxyyyy. then the chip does the following return (xxxx \* 10) + yyyy.

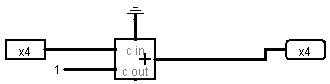
**MUX.**

****

One more service chip a.k.a binary if. If written in the form of a program in C language, MUX does the following:

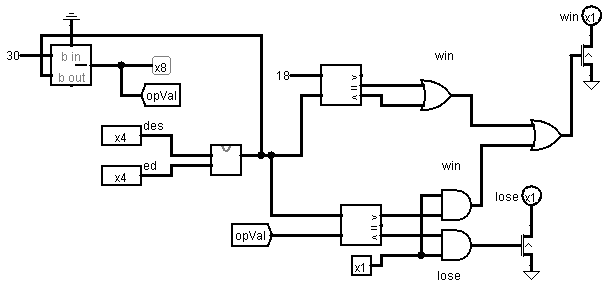


**Incrementer.**

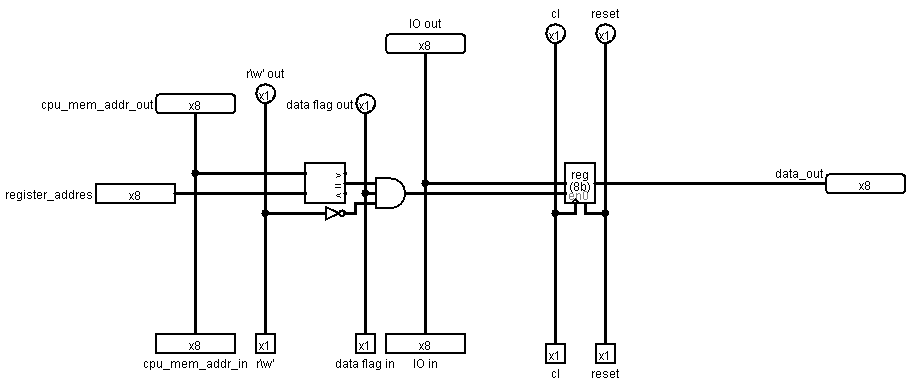
****

Since logisim does not know how to rotate the adder, this circuit simply takes 8 bits at the input and adds 1 to it.

**WLD (Win-Lose-Draw) chip.**



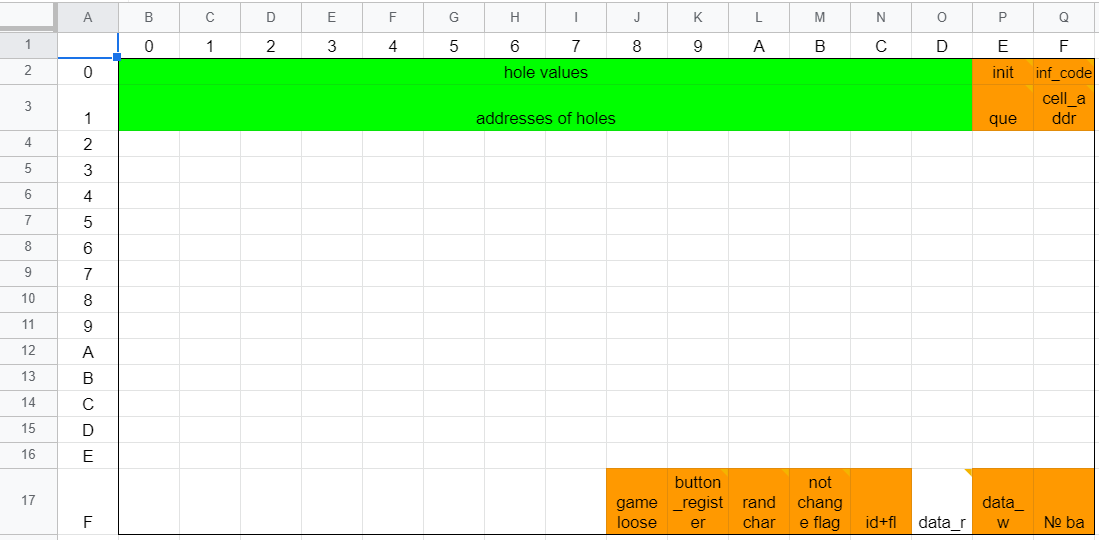
The WLD chip is used to determine the winner of the game. Firstly, this chip checks the number of balls in each player's playing holes and the main hole. If the player has no balls left, then the winner is determined by comparing the number of balls the players have. If the player has 24 balls in the main hole, then he won. The circuit inputs the wire-OR signal from the display drivers on the home side to establish that all holes are empty and so the game has ended.

**Ext\_register.**

In order to write data to RAM and to the field, we came up with this scheme. If we write data into RAM at the address of this external register, then the data will appear simultaneously in RAM and in this register.

**Memory planning**

The memory of this project was designed in this way:



Where:

**hole values** - the number of balls that lie in each hole

**addresses of holes** - the addresses of the holes on the field

**init** - a flag that is set if the memory was initialized for the first time

**inf\_code** - a register that allows you to display information in the form of lights up, an additional function - reset the register responsible for storing buttons

**Displaying the inf\_code flag**

0b00000000

^^^^^^^^

||||||||

|||||||> Error

||||||> W\R to\from field

|||||> Computing

||||> Waiting user action

|||> Game stopped

||> Initializing memory

|> Set button register to 0

> Ready

**queue** - id of the bank that needs to be started to continue the game

**cell\_addr**  - the address of the cell from the field from which we move the balls

**game\_loose** - if it is not 0, then the game must be stopped and someone has won

**button\_register** - if the button was pressed by the player, then the id of the hole corresponding to the button will be written here

**rand char** - every time there is a new random number here, it is generated each time it is read from this address

**not changing flag** - if there is 1 here, then the player \ computer moves a second time

**id+fl** - the id display driver from which we read / write. If the flag is set to 1, then we write on this id, if it is 0, then we read

**data\_r (not implemented)** - if we read data from the field, then there will be information about the number of balls.

**data\_w** - if we write on the field, then the number from here will be the number of balls that lie in the hole

**Structure of data w**

0b00000000

^^^^^^^^

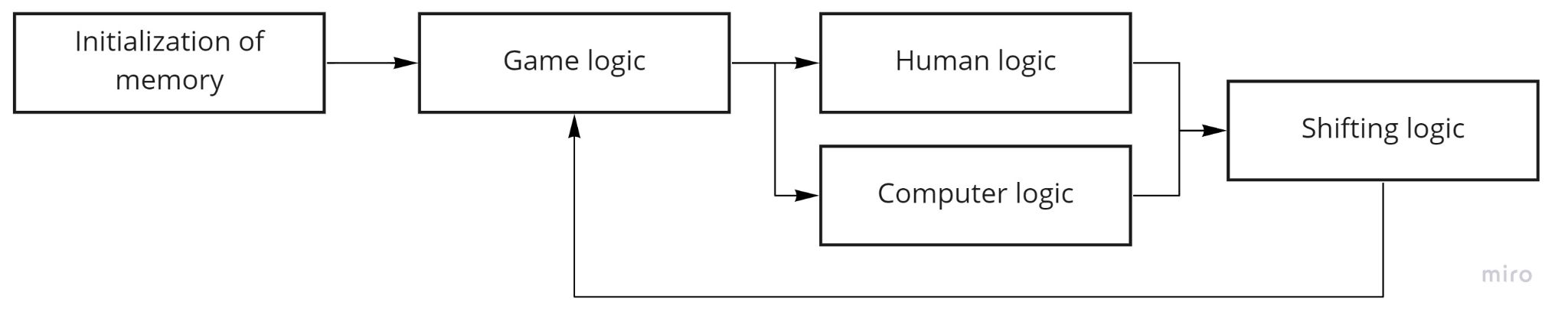
||---> write / read address(4 bits)

> Flag R\W (1 bit)

**№ ba** - the register responsible for the number of the selected bank. If you change the number here, then the bank from which the code is executed will also change.

**Difficulties**

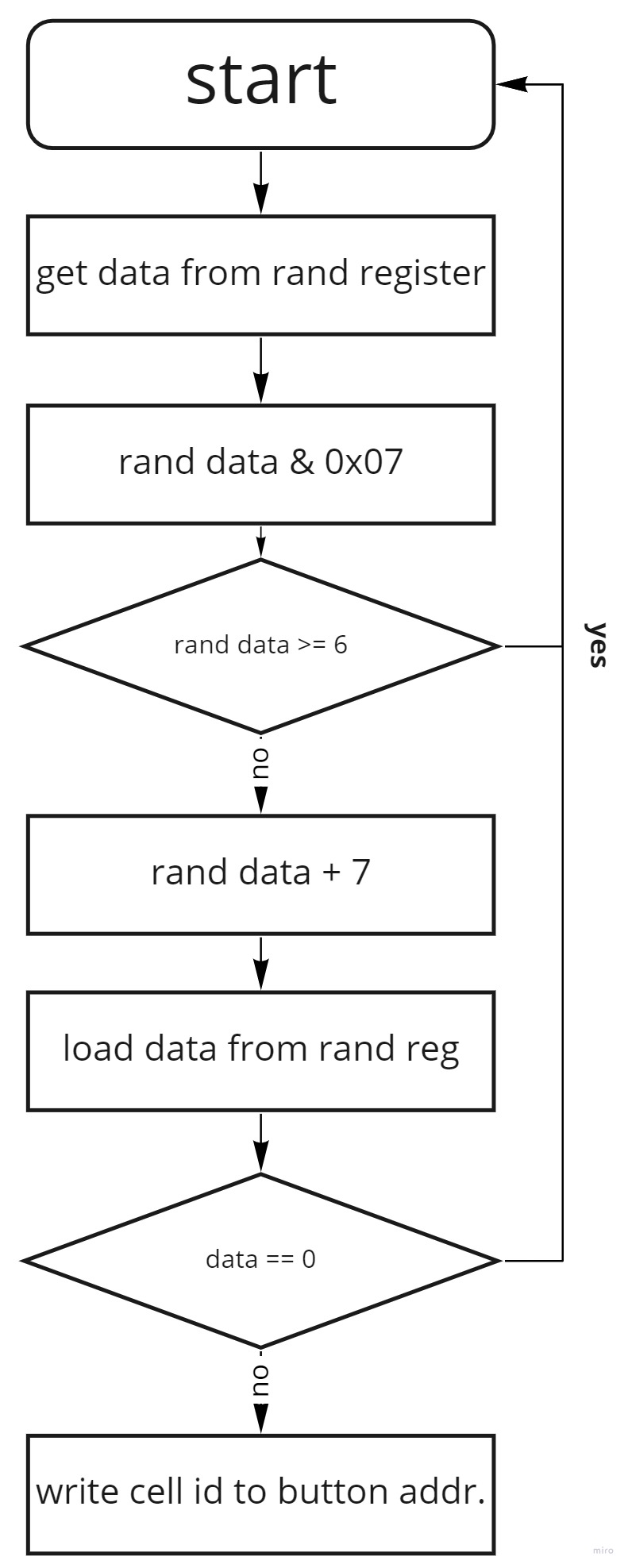
The project takes up too much memory, so we can’t use 1-2 banks of memory. The solution to this problem is to create several memory banks and assign each of them its own purpose.

**The structure of banks of memory**

**Initialization of memory** - this bank is executed only once, in order to load data into RAM

**Game logic** - data output to the field, if there is information that someone won or lost, then the program stops

**Human logic** - getting \ validation of data from the external register of the button

**Computer logic** - data processing from the register of random numbers, data generation

**Algorithm of computer logic**

**Shifting logic** - shifting balls + all the logic associated with this (the last two rules of the game)

**Links**

Github: <https://github.com/AnarCom/cdm-8-mankala>



1. The rules of the game “Mancala” were taken from the description of the project by Alex Shafarenko [↑](#footnote-ref-0)
2. Display driver was designed according to the description of the project by Alex Shafarenko [↑](#footnote-ref-1)
3. Button driver was designed according to the description of the project by Alex Shafarenko with some twists. [↑](#footnote-ref-2)