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 Design

TERM PAPER

Authors:

Kuleshov Artemiy, Pivovarov Fedor, Baklanov Timur

"_____THE GAME OF TV-TENNIS_____"

Novosibirsk, 2024

CONTENTS

INTRODUCTION 1
1.1 PROBLEM STATEMENTS 2
ANALOGUES 2
USER MANUAL 5
HARDWARE 6
4.1 CIRCUIT ELEMENTS AND THEIR FUNCTIONS 7
SOFTWARE 14
5.1 INTERACTIONS WITH HARDWARE 22
CONCLUSION 24

INTRODUCTION

Tennis is one of the first games, the imitation of which was implemented in a computer. The first version of the game was in the first console in the early 70s of the last century.

Our project is a simulation of a tennis game using hardware based on a *CdM-8 microprocessor*.

1.1 PROBLEM STATEMENTS

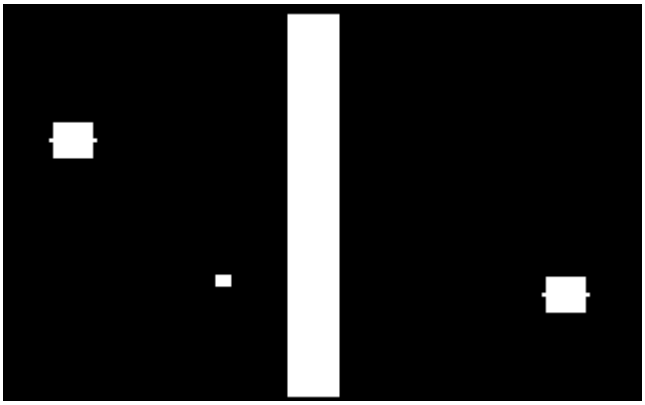
- To implement a kinematic controller chip that will move the bats and the ball around the display

- Write the logic for software CdM-8

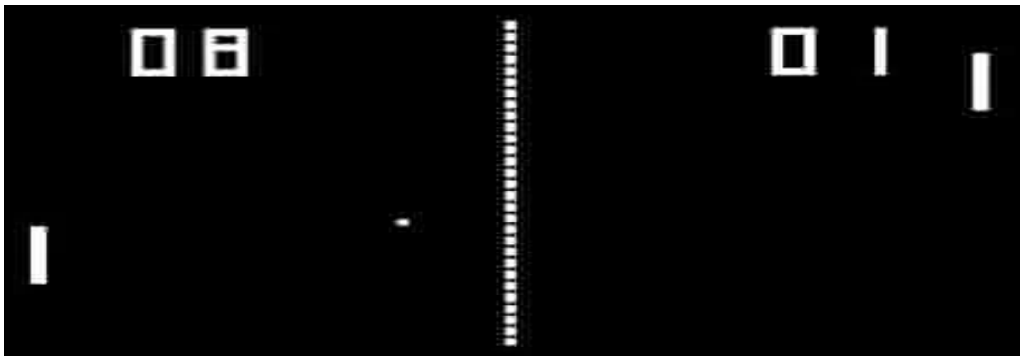
- Implement the interaction of software and hardware

ANALOGUES

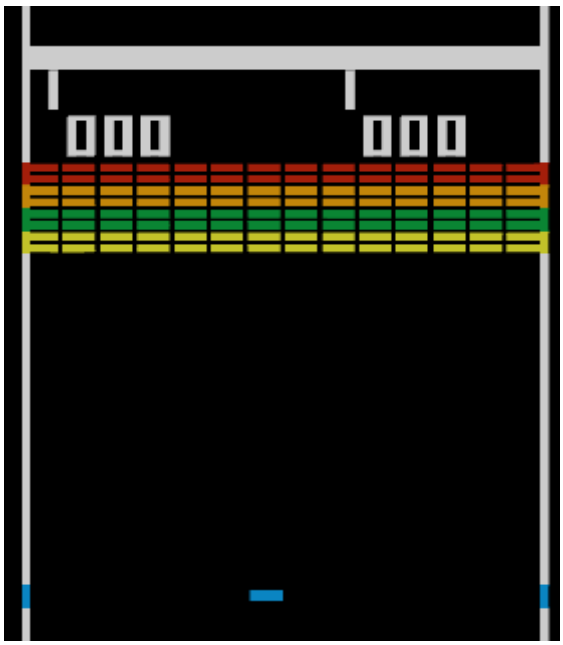
Magnavox Odyssey Tennis (1972) - This game was part of the first Magnavox Odyssey home video game console. It was a very basic version of tennis, where players could control vertically moving "rackets" to hit the ball.



Pong (1972) is a table tennis–themed arcade sports video game, featuring simple two-dimensional graphics, manufactured by Atari and originally released in 1972. It was one of the earliest arcade video games; it was created by Allan Alcorn.



Breakout (1976) - Although it is not tennis in the classical sense, the Breakout game was inspired by the mechanics of the Pong game. Players control a horizontal platform by hitting a ball that smashes bricks at the top of the screen.

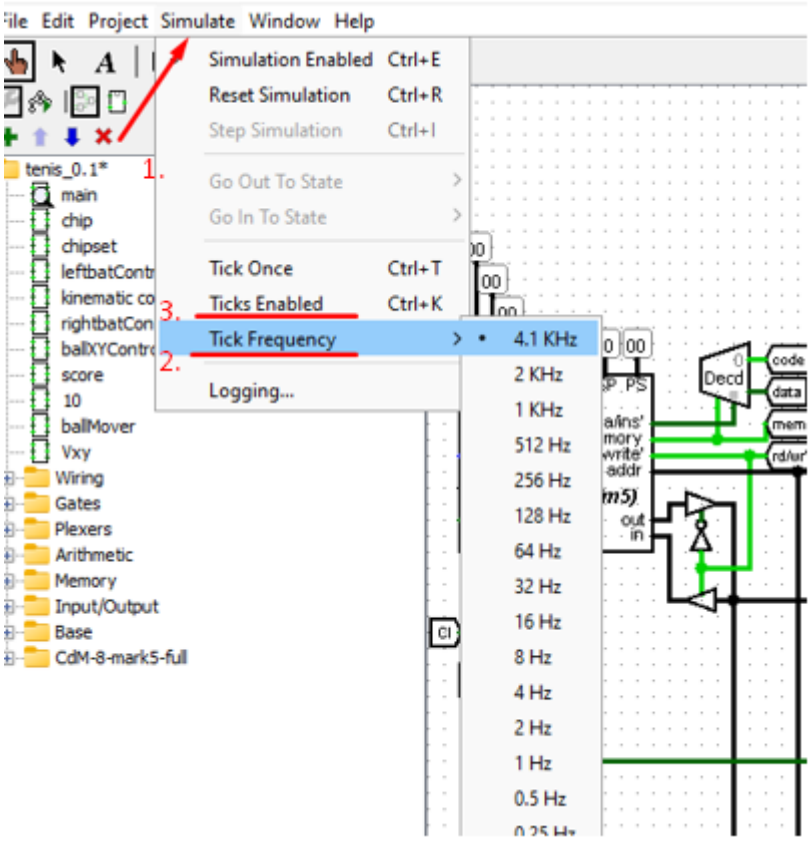


USER MANUAL

To start playing "*THE GAME OF TV-TENNIS*" you need to:

1) Download the following files: "*tv-tennis.circ*" ; "*CdM-8-mark5-full.circ*"

2) Place downloaded files in one directory (folder)



3) Turn on the game

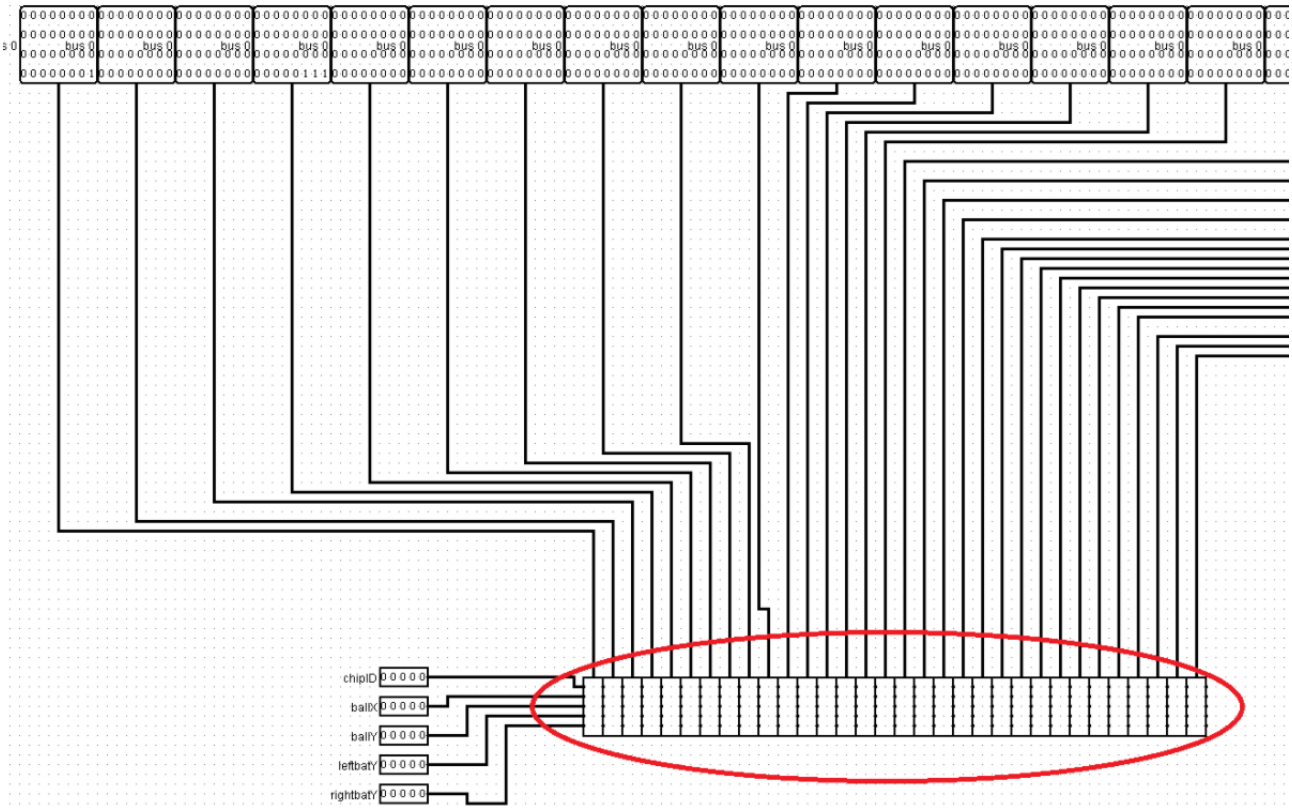
1. Press the button “Simulate” ;
2. Set the desired frequency ;
3. Press “Ctrl + K” or follow the instructions in the picture ;

HARDWARE

The main idea of the "*THE GAME OF TV-TENNIS*" gaming hardware is to use digital logic and microprocessor technology to create an dynamic tennis simulation to give users the ability to control the racket and track the movement of the ball in real time.

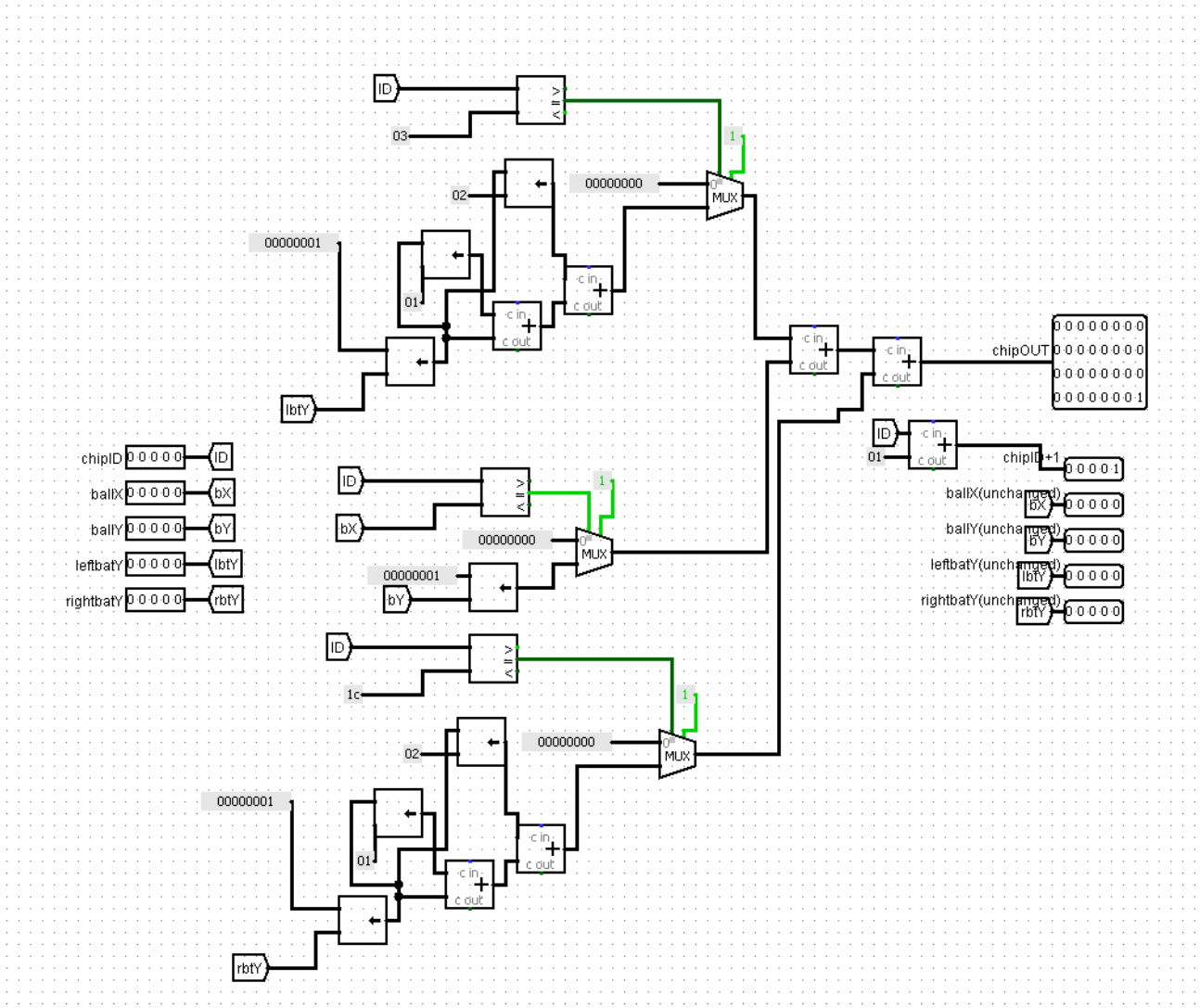
We use *Logisim* to provide the hardware for the entire game. The components as registers are used to store the current coordinates of the ball and rackets, as well as the speeds of their movement, the rackets are controlled using a joystick, data from which is sent to the corresponding controllers. The controllers adjust the vertical position of the rackets based on the input signals, keeping the current positions on the registers.

4.1 CIRCUIT ELEMENTS AND THEIR FUNCTIONS



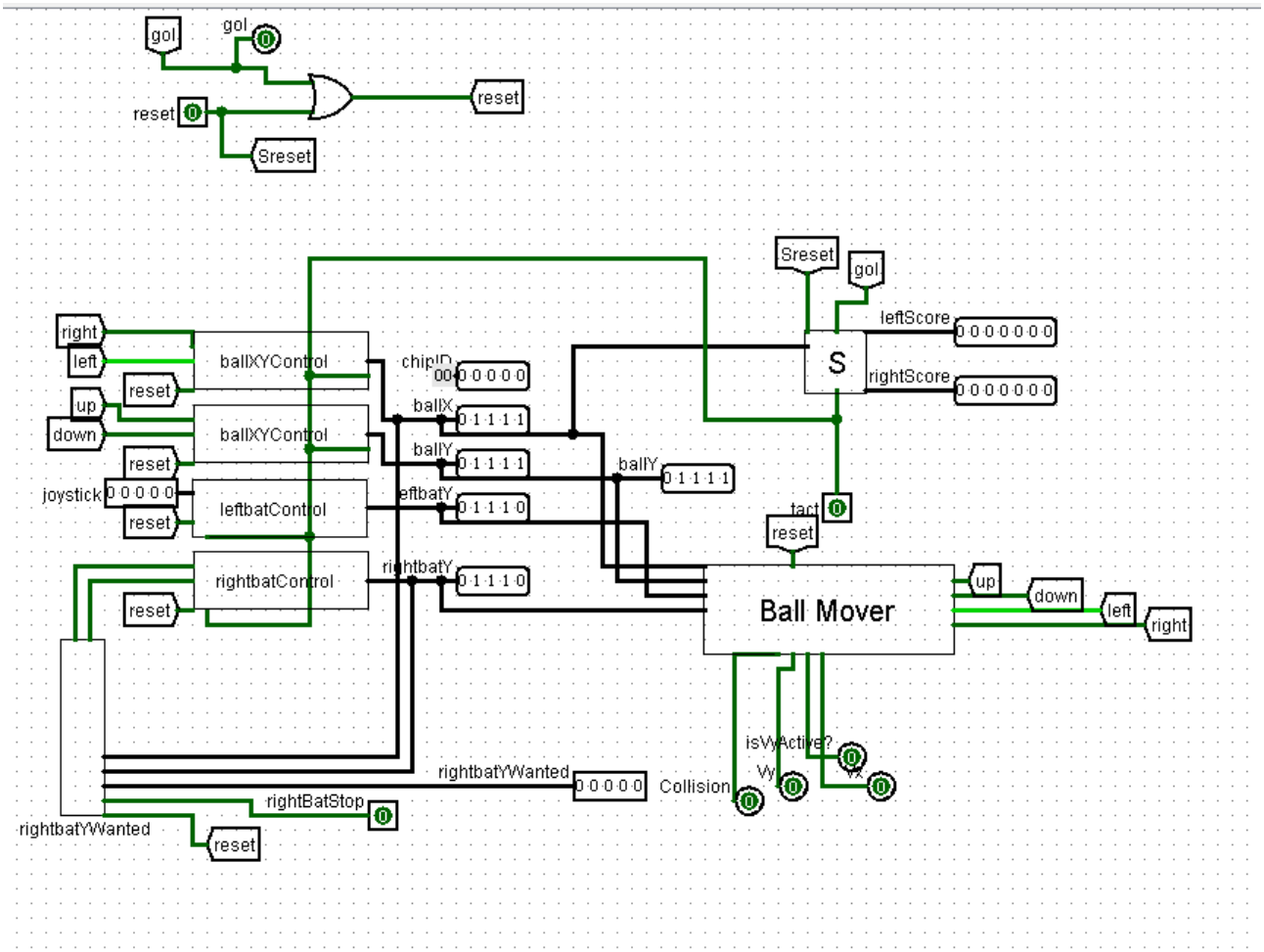
ChipSet circuit.

We started creating the circuit with the display and the chipset component, which contains 32 connected chip components. ChipSet receives data about the position of the ball and both bits, as well as the *Chip ID*, which is initially a constant 0. **ChipSet** passes this data to the first chip, which processes it and passes it to its neighbor, and displays the data in a column corresponding to the *ChipID*.



chip

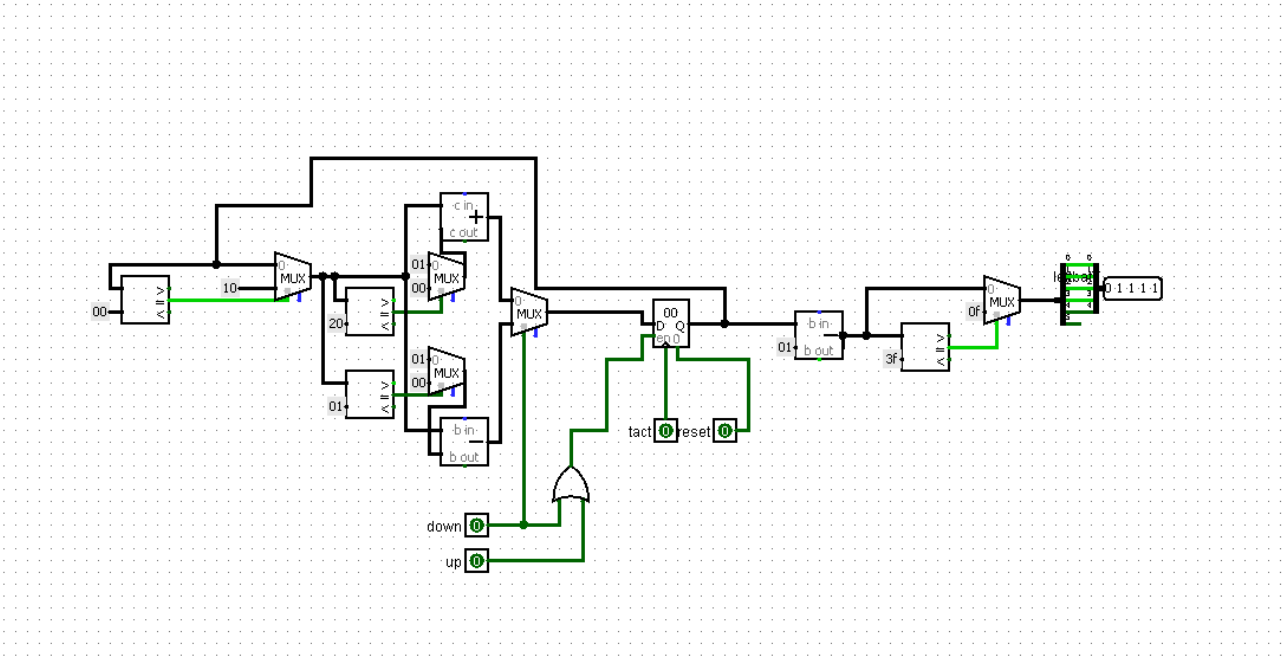
Chip receives the data in the **ChipSet** component, processes it and outputs its corresponding column by highlighting the necessary pixels on the screen, after which it transmits the data on the position of racquets and ball further, as well as the *ChipID* increased by 1.



Kinematic controller

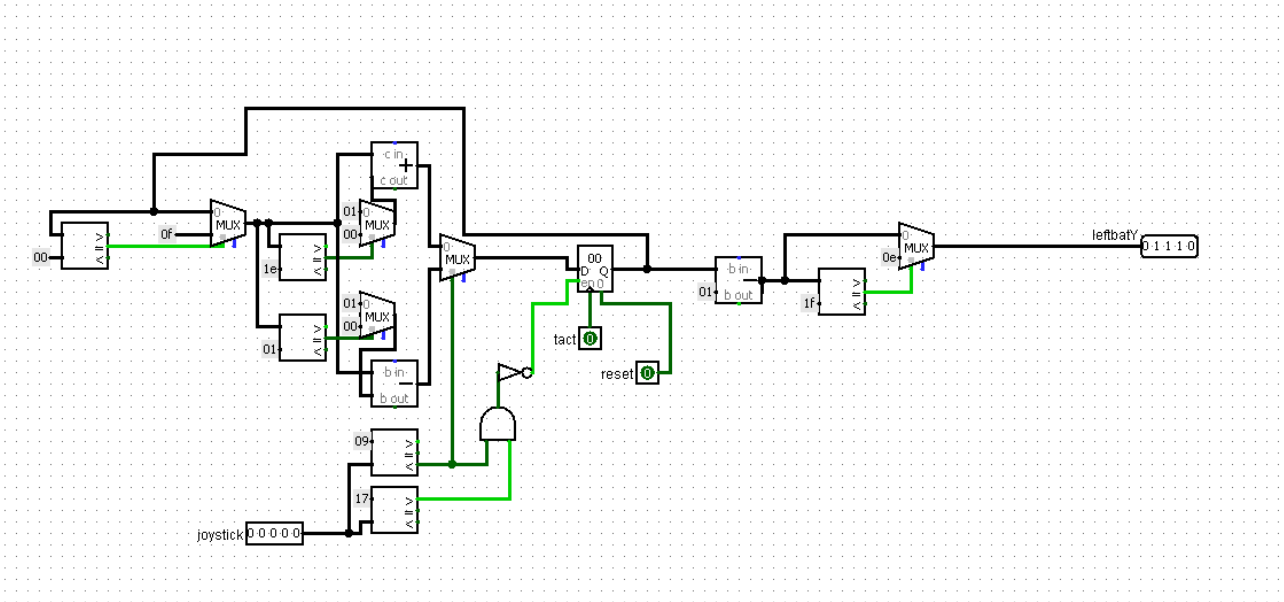
Kinematic Controller serves as the link for all of our components. It receives data from the joystick and the desired right bit position data from *CdM-8*, a signal to complete the calculation of the collision position of the ball with the right wall, as well as a signal from the reset button and a signal from the clock. Returns the score of both players, all necessary data for **ChipSet**, many other useful data for correct operation of *CdM-8*.

We will now look at the operation of all the components contained in **kinematic controller**.

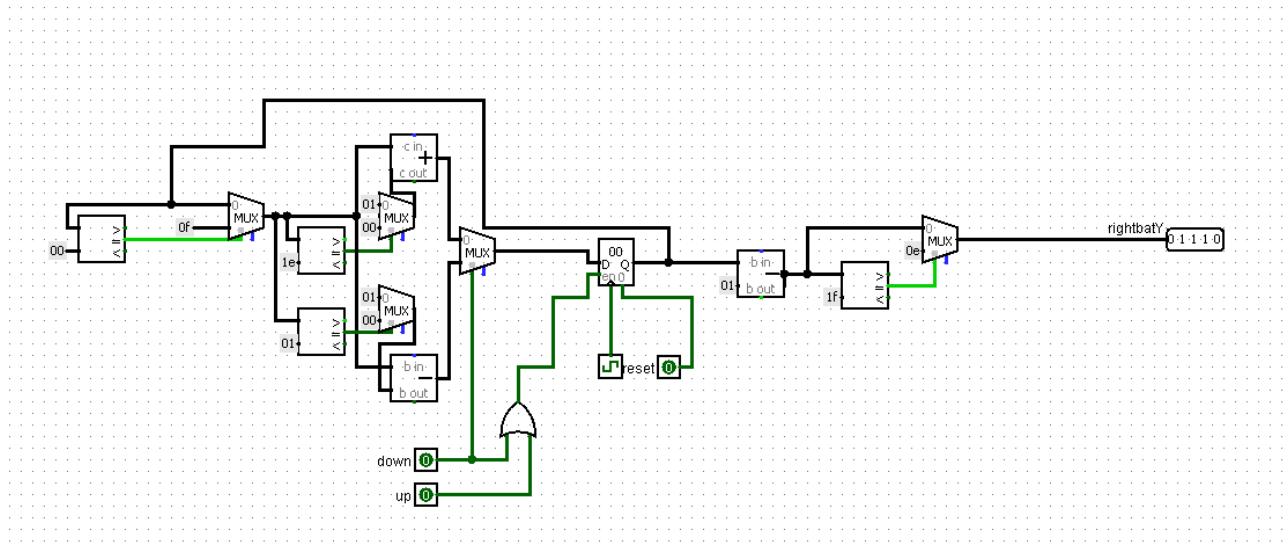


ballXYcontrol

Two almost identical **ballXYcontrol** components that store the ball's X-axis and Y-axis position, and modify it using data from **BallMover.**



leftbatControl

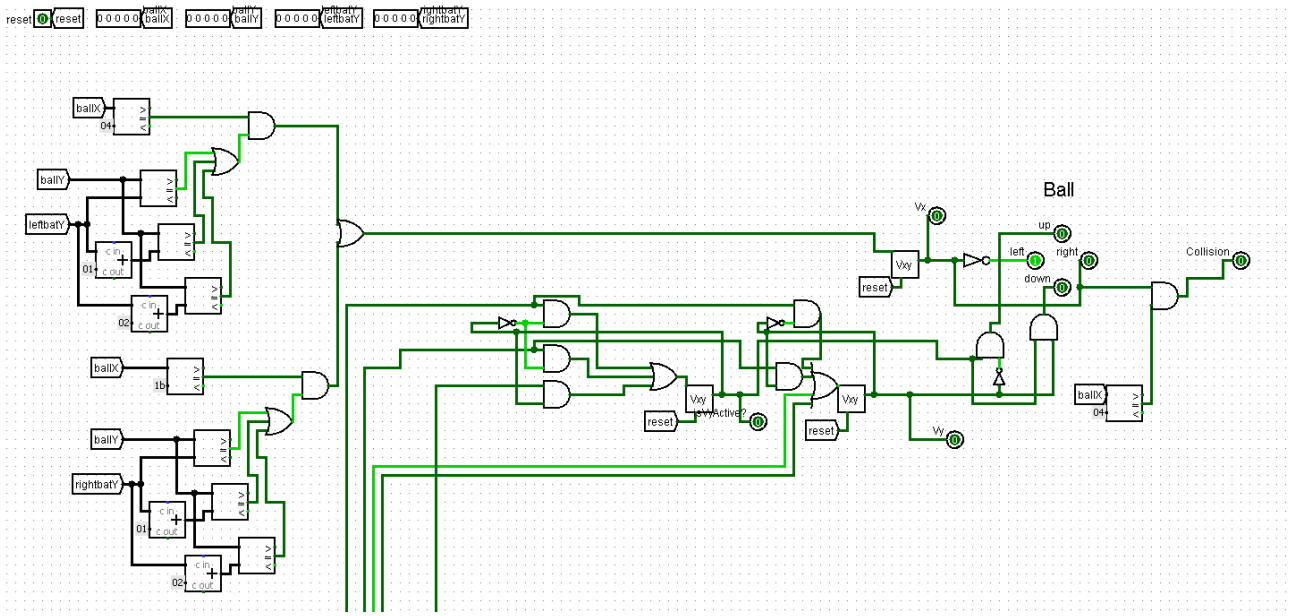


rightbatControl

Two similar components **leftbatControl** and **rightBatControl**, which store the positions of the left and right bits respectively, and change it by receiving data from the joystick and processed data from *CdM-8*.

All these four elements have a similar structure: a register that directly stores the value, an adder and a subtractor that change the value in the

register while taking into account the allowable limits due to comparators and multiplexers, and a reset button that resets the value of the register.



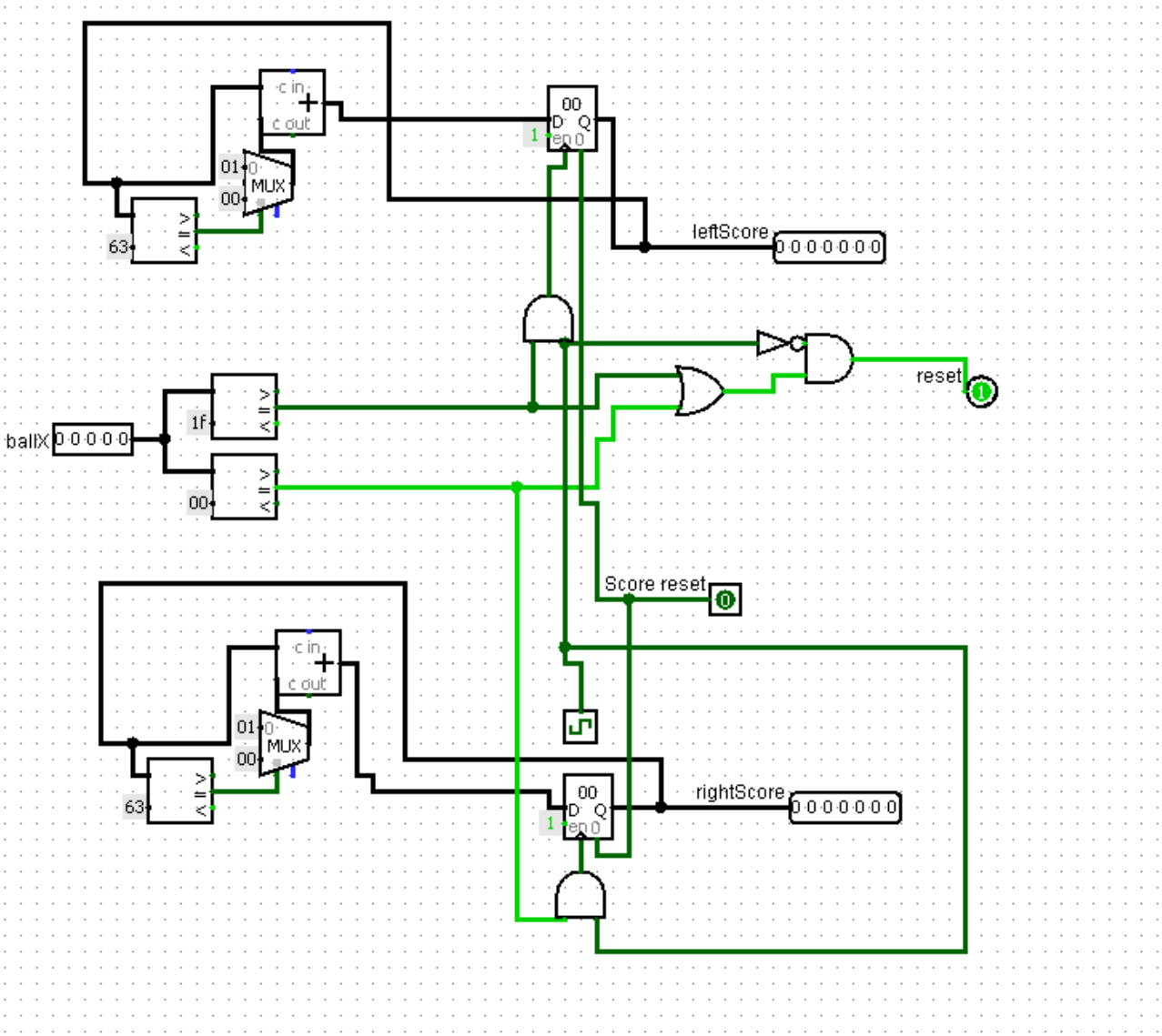
ballMover

BallMover receives data on the position of the ball and both bats, as well as the signal from the reset button. It stores the current direction of the ball in three small *Vxy* components, it handles collisions of the ball with the bats, with the top and bottom wall.

In the top *Vxy* we store the direction of the ball in the X axis, in the left Vxy whether the ball is moving in the Y axis, in the right Vxy the direction of the ball in the Y axis.

When the ball is in the fourth-to-last column and has the direction of travel to the right, we signal the collision output.

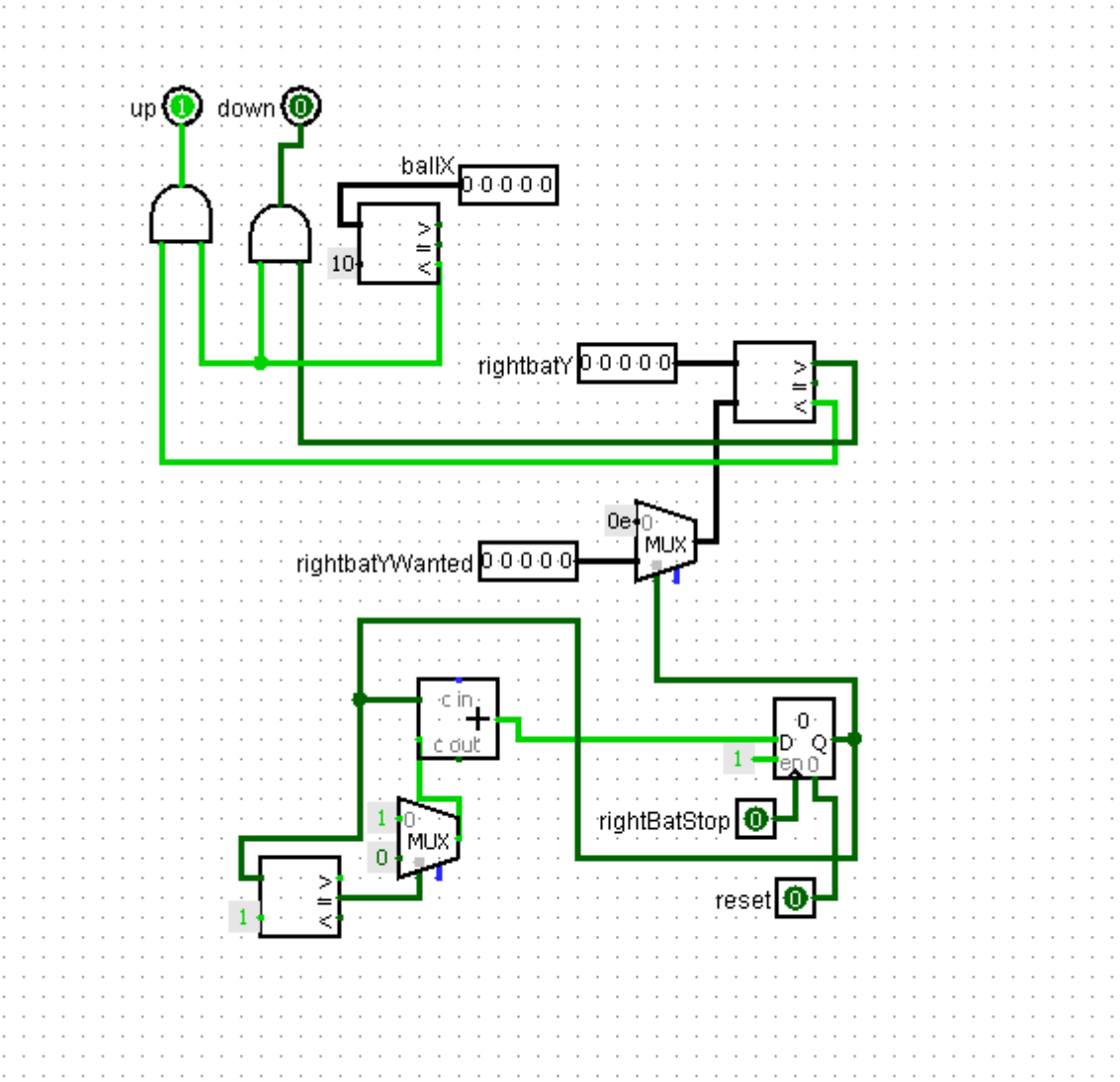
Thus **BallMover** component returns the direction of ball movement, the data required for *CdM-8* operation.



Score

Score accepts the ball position data on X axis, reset signal when the ball touches the left or right wall increases the value in the left or right score register respectively, resets the game without changing the score, the score changes if the game is reset by reset signal.

The module returns left and right player score data, which exit **kinematic controller** and go to the dials on the left and right of the display, respectively, passing through component 10, which breaks the score into tens and ones.

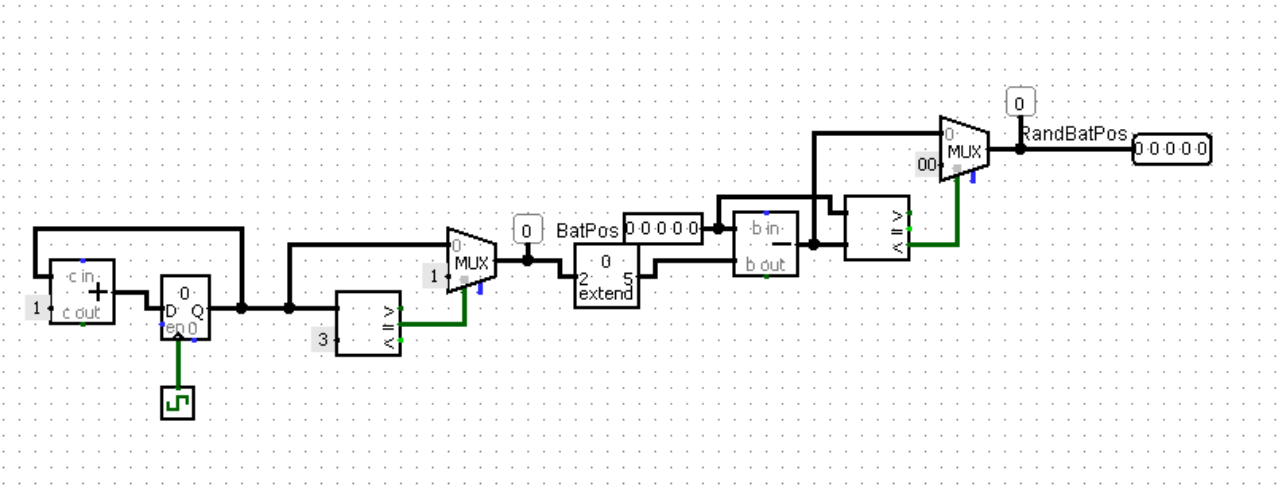


rightbatYWanted

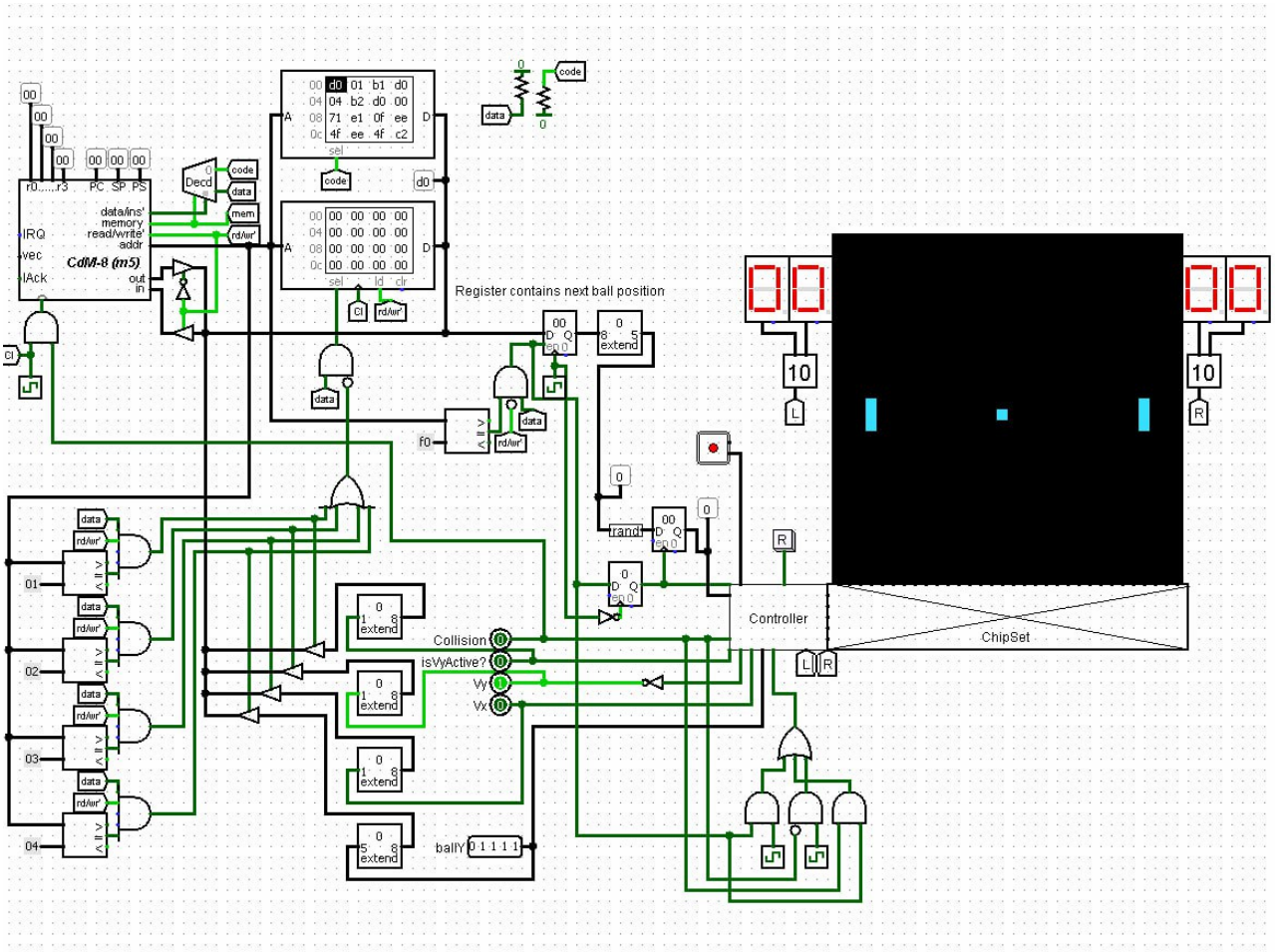
rightbatYWanted receives data about the position of the ball on the X-axis, the position of the right bit, the value of the right bit to occupy, a signal about the end of the calculation of this value.

The component makes sure that the right bat does not move until we have calculated the position it should take to reflect the ball, forbids it to move when the ball has crossed the center border of the display, signal *UP* or *DOWN*, depending on where the right bat needs to move to take the desired position.

Thanks to this the system can adaptively respond to changes in the game, using a combination of fixed logic and variable conditions to create a more dynamic and unpredictable gameplay.



RandBatPos gets the current desired right bit position and subtracts from it a number that is contained in a register and changes with each clock cycle in this order: 0-1-2-1-1-0-0-1-2-1-1-0, returns it.



Kinematic controller

In addition to the components described above, the program contains many logical elements necessary for the correct operation of *CdM-8*, which take data from **Kinematic Controller**, process them and pass them to our processor. We pass the value returned by *CdM-8* and the signal that it has been completed to **Kinematic Controller** by processing it with **RandBatPos** module.

SOFTWARE

To begin with, we will define some conventions.

Registers and their values:

0x01 -> r1 - isVyActive:

The isVyActive flag will be stored in the r1 register, which indicates whether the vertical velocity (Vy) is active.

0x04 -> r2 - ballYPosition:

The r2 register will store the current vertical position of the ball (ballYPosition).

A table with agreements will be provided below for a complete understanding

Condition:

IF isVyActive == 0, THEN RETURN ballYPosition

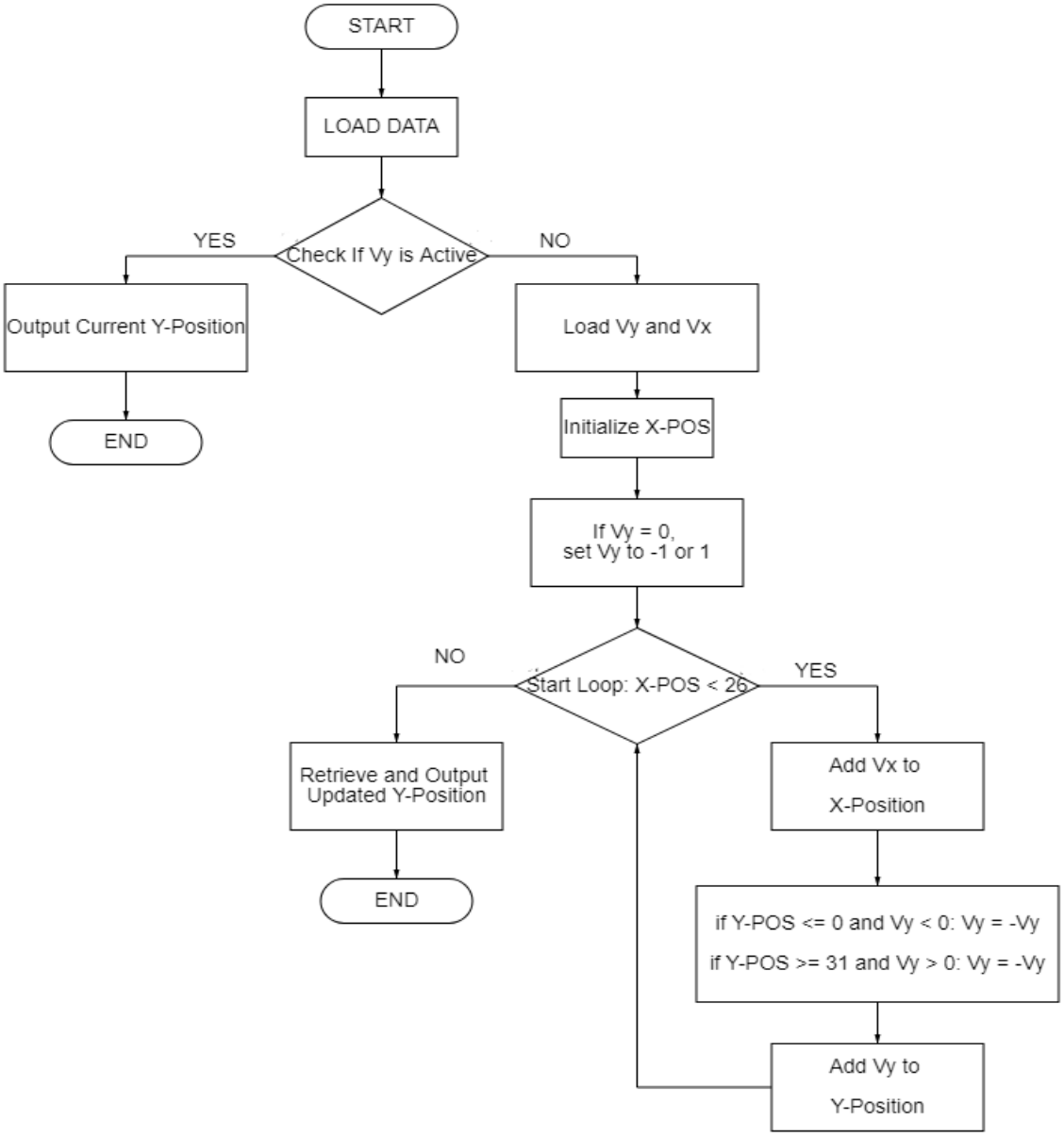
.....

ELSE the program must perform iterative calculations to find the next position of the ball

Address	Register	Description
0x01	R1	isVyActive flag (vertical velocity activity)
0x04	R2	The current vertical position of the ball (ballYPosition)
0x02	R1	Vertical Speed (Vy)
0x03	R2	Horizontal Speed (Vx)
-	R3	Current Horizontal position (Vx)
0xF0	-	Output address

The main idea is to update the position of the ball to simulate movement in a table tennis game. The code processes the vertical and horizontal speeds, as well as the corresponding positions, updating them based on the current speeds, and keeps the position of the ball within the playing field.

Here is a flowchart, which also appears in the presentation.



But here we will focus on *the code.*

```
21  asect 0x00
22
23  ProgrammStart:
24
25  ### --- LOAD DATA FROM CONTROLLER --- ###
26
27      # Load isVyActive from 0x01 to r1
28      ldi r0, 0x01
29      ld r0, r1
30      #load ballYPosition from 0x04 to r2
31      ldi r0, 0x04
32      ld r0, r2
33
```

We take the variables BallYPosition and isVyActive from memory addresses. *isVyActive* determines if the ball's vertical movement should be calculated, and *ballYPosition* is the current vertical position of the ball.

<u>Calculate Next Y-Position:</u>

```
34  ### --- CALCULATES NEXT Y-POS --- ###
35
36      ## IF Vy ISN'T ACTIVE
37      ldi r0, 0x00
38      if
39          cmp r0, r1
40      is eq
41          br answer
42      ## OTHERWISE -> ITERATIVE CALCULATIONS
43      else
44          # push ballYPosition to the stack
45          push r2
46          # load Vy from 0x02 to r1
47          ldi r0, 0x02
48          ld r0, r1
49
```

First checks if the vertical speed (*Vy*) is active. If not, it skips the calculation and moves to output the current position. If active, the position is saved for potential adjustments based on the speed.

Iterative Logic for Position Update**:**

Dynamic Position Adjustment**:**

A complex cycle starts here, and in order to understand it, we will break it into parts.

1) Iterating the settings for calculating positions and adjusting X-POS:

```
64          # load Vx from 0x03 t0 r2
65          ldi r0, 0x03
66          ld r0, r2
67          # define X-POS
68          ldi r3, 3
69
70          ## Iterates while X-POS < 26
71          ## Before all iterations, set r0 to 26
72          ldi r0, 26
73          while
74              cmp r3, r0
75          stays lt
76              # r0 - current ballYPosition
77              pop r0
78              # Vx + current X-POS
79              add r2, r3
80              # push Vx to the stack
81              push r2
```

Loads the horizontal velocity (*Vx*) into register r2 from the memory address 0x03. It then sets the initial horizontal position (*X-POS*) to 3 in

register r3. Sets a loop that will continue as long as *X-POS* (stored in r3) is less than 26. The loop's boundary is set by loading the number 26 into r0.

Vertical Positions Corrections:

The playing field has a bottom and a top position, and we have to make sure that the ball does not cross the boundaries. Therefore, if the ball hits either boundary, we reflect the *Vy velocity*.

```
83              ## COUNT NEXT Y-POS
84              #compare Vy with 0, Y-POS with 0 and 31
85              ##- IF Vy < 0
86              ##-- IF Y-POS < 0 | Y-POS >= 31
87              ldi r2, 0
88
89              if
90                  cmp r1, r2
91              ## IF Vy is negative (< 0)
92              is lt
93                  ldi r2, 0
94
95                  if
96                      cmp r0, r2
97                  ##:: Y-POS <= 0
98                  is le
99                      # (-Vy) + Y-POS
100                     neg r1
101                     add r1, r0
102                 ##: Y-POS > 0
103                 else
104                     # push Vy to the stack
105                     push r1
106                     # Y-POS + Vy
107                     add r1, r0
108                     # pop Vy from the stack
109                     pop r1
110                 fi
```

Handling Negative Vertical Velocity: This enclosed conditional structure checks if the vertical velocity (*Vy*) is negative. If so, and if the current vertical position (*Y-POS*) is less than or equal to zero, it adjusts *Y-*

POS by negating *Vy* and adding it to *Y-POS*. If *Y-POS* is positive, Vy is added directly.

```
112             ##: Vy is not-negative (>= 0)
113             else
114                 ldi r2, 31
115
116                 ##: Y-POS > 31
117                 if
118                     cmp r0, r2
119                 ##:: Y-POS < 31
120                 is lt
121                     # Vy + Y-POS
122                     add r1, r0
123                 ##:: Y-POS >= 31
124                 else
125                     # (-Vy) + Y-POS
126                     neg r1
127                     add r1, r0
128                 fi
129             fi
```

Handling Positive Vertical Velocity: If *Vy* is not negative, checks if *Y-POS* exceeds 31. If it does not, *Vy* is simply added to *Y-POS*. If *Y-POS* is 31 or more, Vy is negated and added to *Y-POS* to keep it within bounds.

```
129                 fi
130                 # pop Vx from the stack
131                 pop r2
132                 # push ballYPosition to the stack
133                 push r0
134                 ## Before all iterations makes r0=26
135                 ldi r0, 26
136             wend
137
138             ## NOW WE NEED ONLY ballYPosition ##
139
140             # get Y-POS from stack
141             pop r2
142         fi
```

Finalize Iteration and Reset Loop Conditions.

Summarizing, The loop adjusts the ball's position iteratively based on Vx and Vy, applying corrections for collisions and boundaries within the game area.

```
143 ### --- WRITE ANSWER --- ###
144
145 answer:
146     ldi r0, 0xf0
147     st r0, r2
148
149 ### --- THE END: ALL CALCULATIONS HAVE BEEN DONE --- ###
150
151     br   ProgrammStart
152
153     halt
154
    end
```

After the computations are complete, the new *ballYPosition* is stored to a designated output register (*0xF0*). The program then either loops back to start or halts, based on the end condition.

5.1 INTERACTIONS WITH HARDWARE

Role of Logisim:

The ball control system is fully implemented at the hardware level in Logisim. This means that the movement of the ball and its

interaction with the elements of the game (walls, bats) are controlled by logic circuits developed in Logisim.

Role of Software part:

The code is used to calculate the *expected position* of the ball in the future. This calculation is necessary to determine the position of the bats so that they can effectively hit the ball. Thus, the code predicts where the ball will be in order to adjust the position of the bats accordingly.

Interactions:

As part of the CdM-8 code, the processor reads data from external sources into registers. These data include:

- *Vx* and *Vy* — the velocity of the ball horizontally and vertically, respectively;
- *isVyActive* — the flag of the activity of the vertical movement of the ball;
- *ballYPosition* — the current vertical position of the ball

These values are used to calculate the future position of the ball, and then the position of the rackets is adapted based on these calculations.

CONCLUSION

THE GAME OF TV-TENNIS is a practical example of how *CdM-8* programming and hardware components can work together to simulate a video game, where the software directly controls the hardware to create dynamic and responsive user interactions.

This coordination is crucial for understanding how low-level programming affects the operation of hardware.