# Novosibirsk State University Informational Technologies Faculty Computer Science and System Design

TV-Tennis

**Work Group 21215**

**Authors:**

Blaginin Artyom

Valikov Nikolai

Khlebnikov Vadim

**Novosibirsk, 2022**

Contents

[General Overview 3](#_Toc103285401)

[Part 1: Hardware: Playing Field and Videochip 4](#_Toc103285402)

[Part 2: Hardware: Kinematic Controller 6](#_Toc103285403)

[Reflect module 7](#_Toc103285404)

[Coordinates of the ball 9](#_Toc103285405)

[Has ball hit the bat? 9](#_Toc103285406)

[Right Bat Control 10](#_Toc103285407)

[Part 3: Software: CDM-8 and Control of Right Bat 11](#_Toc103285408)

[Memory Mapped IO 11](#_Toc103285409)

[Algorithm and Pseudocode 12](#_Toc103285410)

[Further developments of TV-Tennis 13](#_Toc103285411)

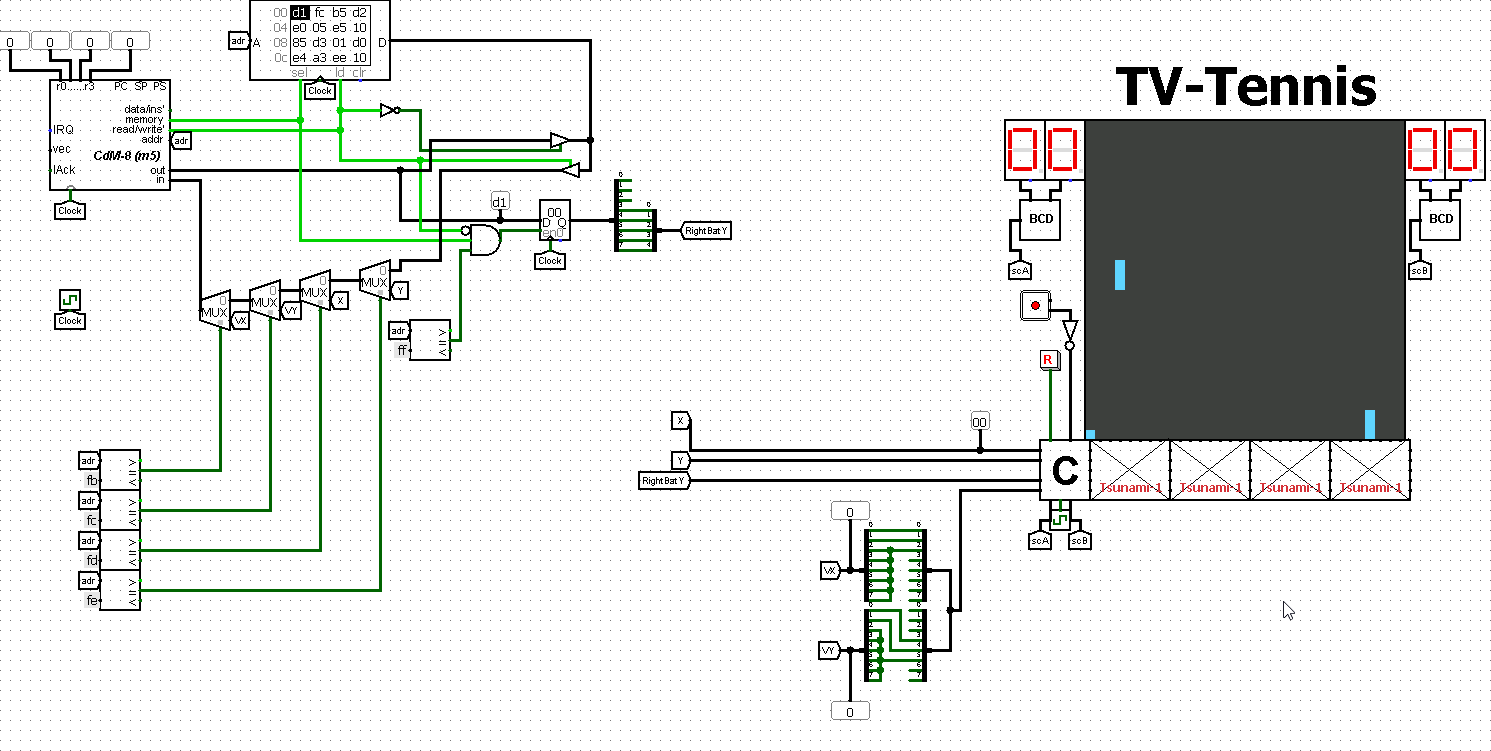
# General Overview

Welcome to the guide to the game **"TV Tennis"**, which was made by our group! This game originates in 1972. It was this year that **Atari** released one of the first commercially successful video games - **Pong**. We present to your attention a new generation of this game.

It is well known that there are a lot of tennis games. With the help of tennis games you can follow the evolution of computer games from a large game machine with primitive graphics to the most complex 3D games with ray tracing and other modern technologies.

However, many are wondering what the games look like on the hardware level? We want to show how the computer shows the picture of the game on the screen and why it all works! Of course, our game is far from the real state of affairs but it explains many principles of circuit engineering very well.

Our group spent more than two months creating a logic circuit in **Logisim** as well as writing artificial intelligence on a **CDM-8 processor**. In this documentation we would like to present all our developments as well as ideas that could be implemented in the future.



**The main view of the circuit: on the left - the CDM-8 processor controlling the right bat, on the right – the playing field**

On the right side are *the* *playing field*, *the score*, *4 video chips* drawing the playing field and *a kinematic controller* controlling the ball and bats. On the left is a *CDM-8 processor* that tries to predict the flight of the ball, thus moving the right bat to the place where the ball may fly.

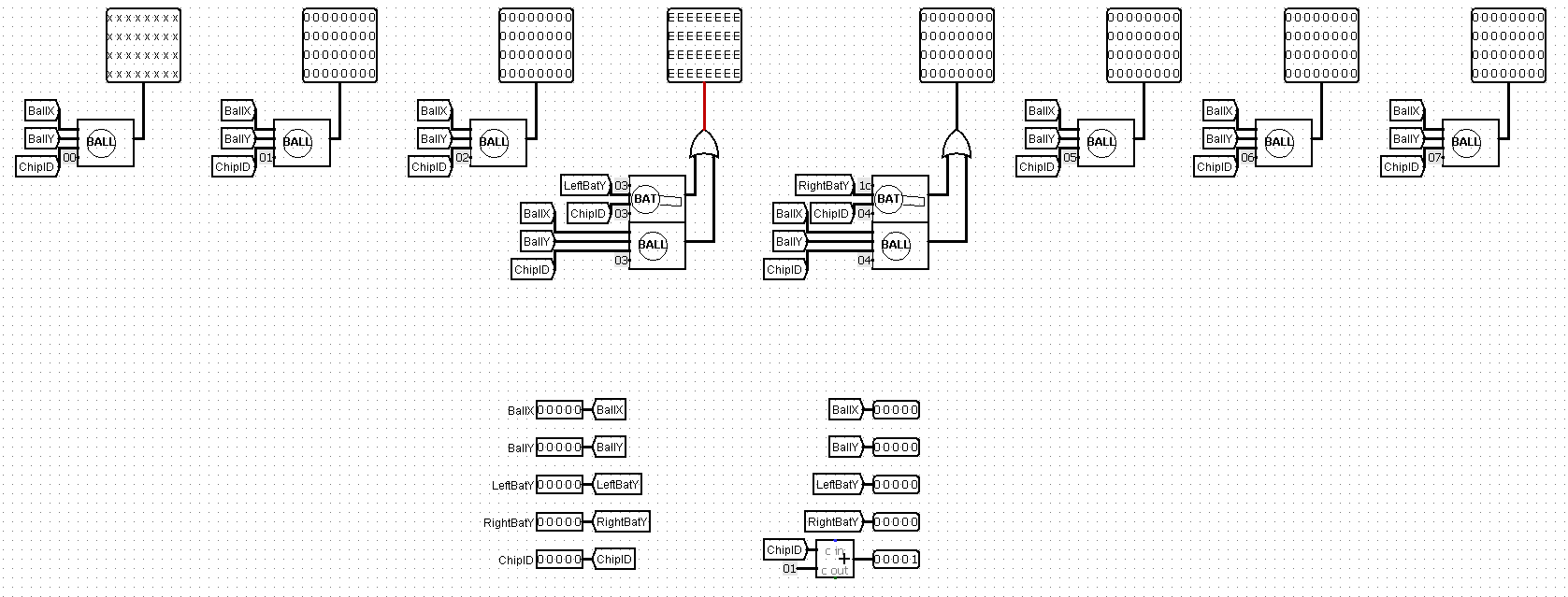
# Part 1: Hardware: Playing Field and Videochip

The playing field consists of several parts: a 32x32 screen on which only two bats and a ball are drawn, two scoreboards on which the current score of the game is displayed, a joystick, and the reset button of the game.

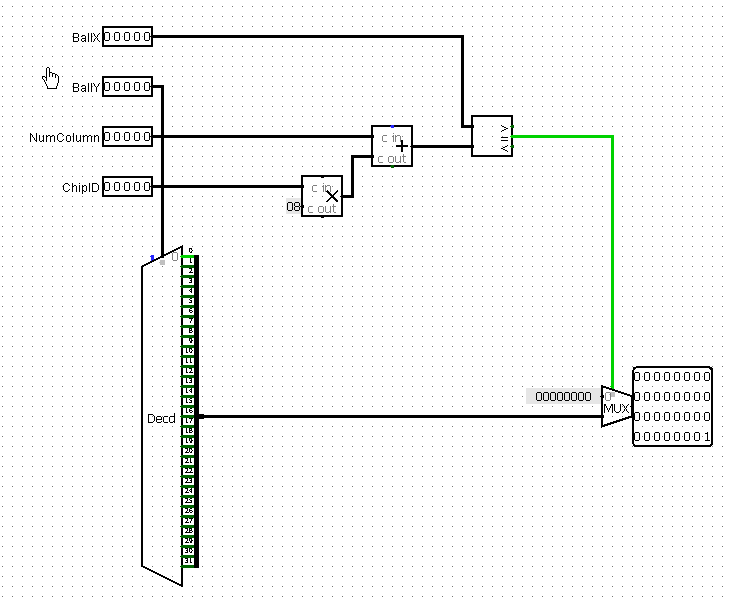
The player controls the left bat with a joystick. The **BCD** part translates a number from hexadecimal to decimal.

One of the important components of the game is the Tsunami-1 video chip which renders the whole game. It consists of 4 similar cores, each of which draws a picture in its columns (0-7, 8-15, 16-23 and 24-31). Each core has its own **ChipID**, the number of the first one is zero and the remaining chips receive an incremented number from the previous chip.

5 five-bit parameters are supplied to the core itself: the position of the ball on **X** and **Y**, the **Y** position for the first and second bat and the chip number. The positions of the left and right bats on **X** are 3 and 28 respectively.

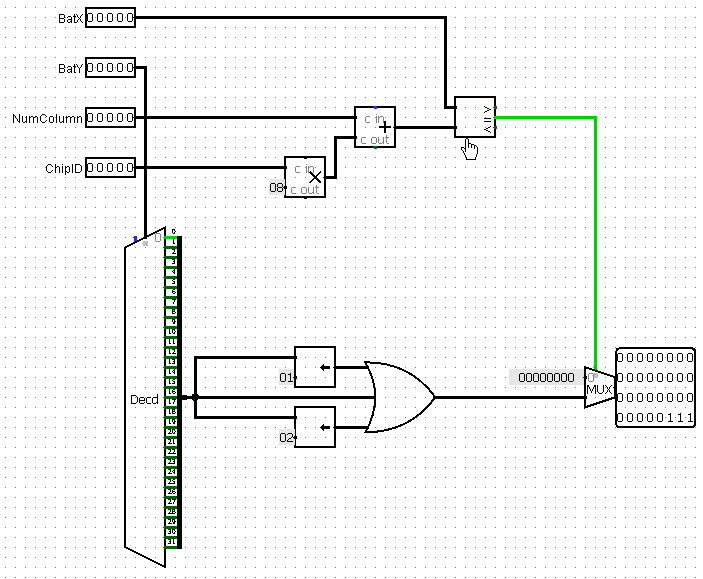


Let's take a closer look at the details of **BALL** and **BAT**.

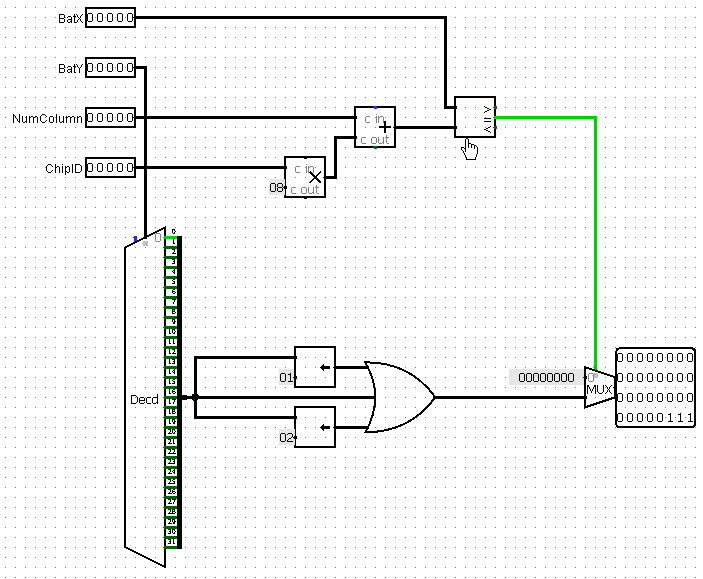
**BALL** draws the ball on the screen. To do this it calculates the current column in which to draw the ball and then compares the column number with the actual position of the ball.

If there is a match, the n-th **BALL** will draw the ball in the n-column and in the row with the number **BallY** (numbering goes from 0 from bottom to top).

If the column in which the ball is located differs from the current column in which the **BALL** is drawing, the chip just does not draw anything.

The **BAT** part works in a similar way. It draws two bats in the same corresponding columns and also adds 2 more pixels from above to the **BatY** coordinate.

Chip ‘BALL’’

Chip ‘BAT’’

# Part 2: Hardware: Kinematic Controller

So, this is the main part of controlling the whole game. It controls the coordinates of the ball, its speed and bats. In addition, the controller monitors when the ball bounces off the wall and off the bat and also keeps score of the game. Let's take a closer look at this controller and its components.

There are 4 inputs: **clock**, the clock frequency supplied to the controller, **Reset**, **LeftYIn**, the coordinate of the left bat controlled from the joystick, and **RightYIn**, the coordinate of the right bat controlled by artificial intelligence.

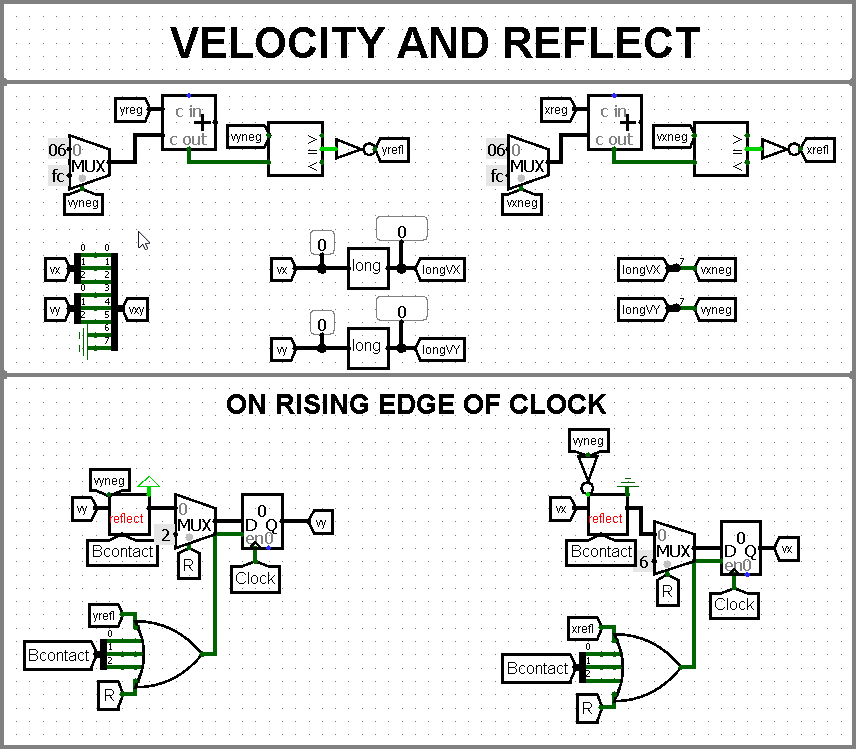
The controller has ten outputs. Five of them (**ballX**, **BallY**, **LeftBatY**, **RightBatY**, **ChipID**) are supplied to the video chip for further rendering and have a bit depth of 5. Three more on the left (**BallX\_8**, **BallY\_8**, **VXY**) have an eight-bit representation, they are needed by the processor so that it can predict the further flight of the ball but more on that later. The score of the first and second player (**scA** and **scB**) is applied to the last two exits.

Ball’s velocity

In order for the speed of the ball to appear let's start small. When you press the Reset button (tunnel with the name **R**) on the rising edge of the clock, the speed is reset to the default values, namely **vx** = -2, **vy** = 2 (five-bit tunnels).

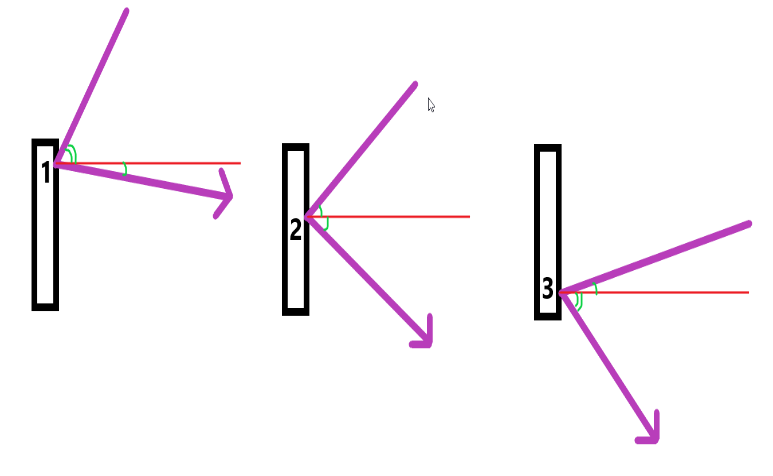
Let's add a few more tunnels:

* **vxy**(8 bit) = **vx**(0-2) : **vy**(3-5) : 00;
* **longVX**(8 bit) = **vx**-arithmetically;
* **longVY**(8 bit) = **vy**-arithmetically[[1]](#footnote-1);
* **vxneg** = **longVX**(7th bit);
* **vyneg** = **longVY**(7th bit);

The value of **vxy**, **longVX** and **longVY** is supplied to the output. Now let's understand at what point the ball hits the wall. It is clear that when controller adds the velocity value to the coordinate, we get an unsigned overflow. The ball has joined the wall and its velocity should change to the opposite. Thus, we get the values of the **xrefl** and **yrefl** tunnels.

The speed change occurs in this way:

* If the ball bounces off a vertical wall, the velocity along **X** changes sign and **vy** does not change.
* Similarly, if the ball bounces off a horizontal wall, **vy** changes sign.

The most interesting thing happens when the ball bounces off the bat. It would probably be boring if the ball had a constant module of velocity and initially the velocity module is equal to two. To make the ball fly at different angles we introduced such ball physics.

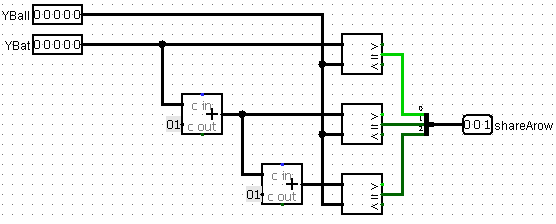
The ball can fly to the bat either from below it or from above. Let's consider the change in speed on the example of the drawing when the ball arrives from above. We have that the bat consists of three pixels. We number them from 1 to 3 from top to bottom.

|  |  |  |
| --- | --- | --- |
| Sector | VX | VY |
| 1 | |VX| + 1 | |VY| – 1 |
| 2 | |VX| | |VY| |
| 3 | |VX| – 1 | |VY| + 1 |

* *When the ball hits sector 1,* ***vx*** *increases by module and* ***vy*** *decreases*.
* *When the ball hits sector 2,* ***vx*** *and* ***vy*** *do not change*.
* *When the ball hits sector 3,* ***vy*** *increases by module and* ***vx*** *decreases*.

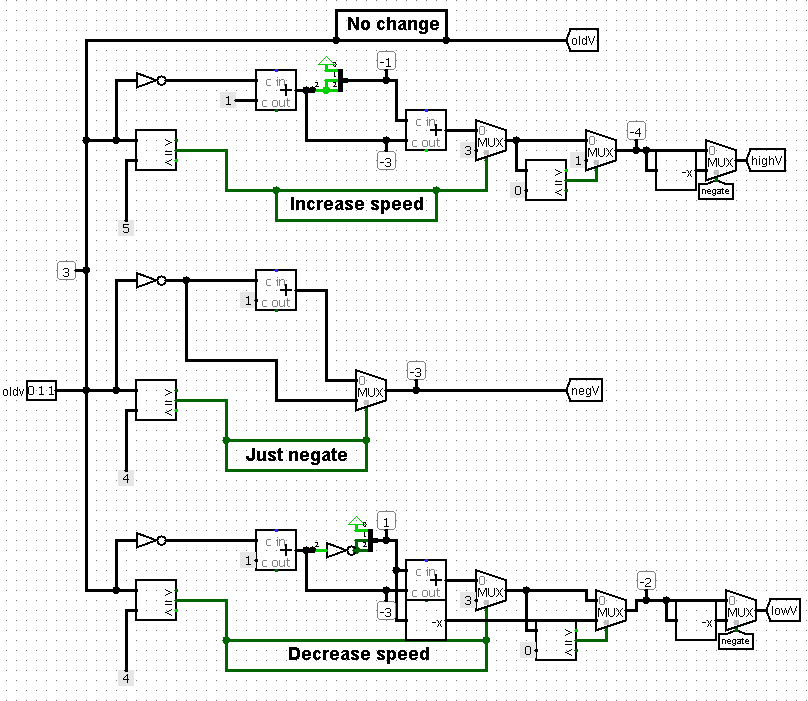
This works with both the left and right bat. If the ball is flying from below, then the situation is similar to the previous one but the numbering from 1 to 3 occurs from the bottom up.

## Reflect module

Let's talk about how the controller tracks the collision of the ball with the bat. We use the inrange module to understand whether the ball is on the same line with any sector of the bat.

The input is the **Y** coordinate of the ball and the coordinate of the first pixel of the bat. A three-bit number is supplied to the output, each bit indicates in which sector of the bat the ball is located. In our controller this value is stored by the **Bcontact** tunnel. We will tell you more about it later.

Chip ”inrange”

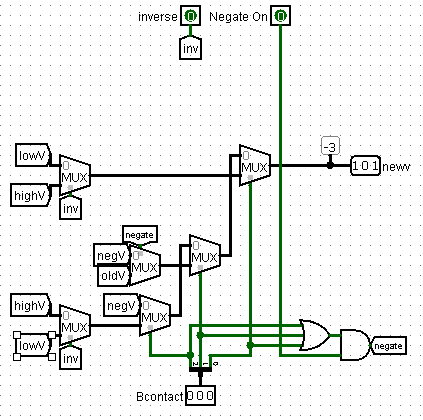
Now let's look at the large enough reflect module.

The first part of the module returns the values of the speeds that will be after the rebound from the bat. The initial **oldV** speed is applied to the input and the following speeds are applied to the tunnels:

* **oldV** = **oldv**;
* |**highV**| = |**oldv**| + 1;
* **negV** = -sign(**oldv**)\*|**oldv**|;
* |**lowV**| = |**oldv**| - 1;

Chip “reflect”

It is important to note that the speed changes only in the range from -4 to 3 and with the exception of zero! Therefore, it was important to consider special cases in the scheme (it is impossible to reduce by module 1 and it is impossible to increase by module -4).

It remains to understand what speed needs to be applied to the output. Since the speed on **X** changes its sign and the speed on **Y** does not change it, let's enter the **Negate On** bit indicating whether the speed is opposite in sign or not.

As we understood earlier, we need to recognize what we are doing with speed when the ball hits the bat from above or from below. If the inverse bit is zero, the ball swoops from above – all the default values, otherwise the speed ‘increase’ and ‘decrease’ are reversed (see the table above).

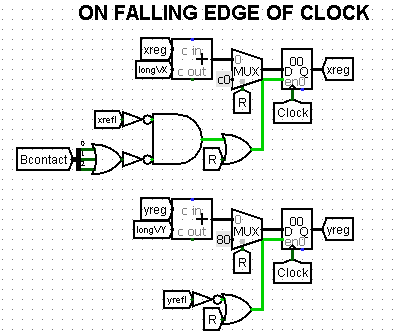
Another part of chip “reflect”

This module satisfies the physics we wanted to get above. This module also handles the case if the ball bounced off the wall, **Bcontact** is simply equal to zero.

Thus, we obtain the principle of speed change:

1. Use the **Reset** button to set the initial speed;
2. When colliding with a vertical wall and a bat, the velocity by module can change and the sign **vx** changes but **vy** does not;
3. When colliding with a horizontal wall, the **vy** sign changes but **vx** does not;
4. The speed change occurs when the **Clock** is equal to 1 and if there was a collision with a wall, a bat or the **Reset** button was pressed.

## Coordinates of the ball

When we have the speed of the ball, now we can talk about its coordinates on the playing field. It is important to note that changing the coordinate is possible only on the falling edge of the clock. This is done in order to first set the correct ball speeds before changing the coordinates.

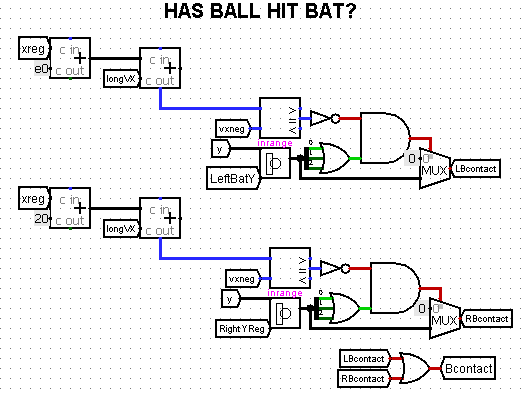
So, we have the speed of the ball. For each falling edge of the clock we add to the **X** and **Y** coordinates of the ball the values of its velocities **longVX** and **longVY**. We update the coordinate values in the registers and write these values to the tunnels **xreg** and **yreg** (8 bit).

However, the **X** coordinate is not added if we pressed the **Reset** button or the ball collided with a vertical wall or a bat. The **Y** coordinate is not updated if the **Reset** button is also pressed or the ball hit a horizontal wall.

When the **Reset** button is pressed, the initial position of the ball is written in **xreg** and **yreg**: 0xc0 and 0x80 respectively.

The values of **xreg** and **yreg** are converted to 5-bit strings and we get the values for the output of **BallX** and **BallY**.

## Has ball hit the bat?

Checking whether the ball has landed in the bat is carried out by two parts that return the values **LBcontact** and **RBcontact** by hitting the left and right bat respectively.

The scheme is quite simple: the bats are located at coordinates 24-31 and 240-247, so if the ball is in the range of the bat (**inrange** is triggered) and its coordinate becomes less than 32 (0x20) when the speed increases, the ball touched the left bat. Otherwise if the coordinate becomes more than 240 (0xe0), the ball touched the right bat.

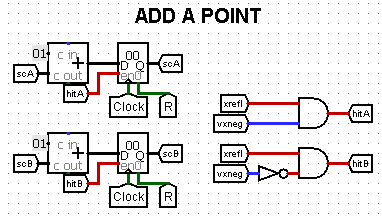
**LBcontact** and **RBcontact** store the sector in which the ball hit or contain zero. The latter means that the ball did not hit the bat.

Combining **LBcontact** and **RBcontact** we get the value of **Bcontact** meaning that the ball touched any bat as well as the sector in which the ball hit.

**Add a point**

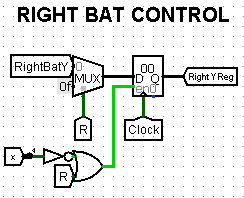
If a player misses the ball with a bat, the ball flies to the left or right edge of the screen, so **xrefl** = 1. Thus, if the ball hit the left edge, i.e. **vxneg** = 0, a point is scored to the computer and **hitA** = 1. Otherwise a point is scored to the human account and **hitB** = 1.

For each clock cycle of the rising front, if a point in someone's favor was counted, the player's score is saved in the register increased by 1. The scores are stored in the **scA** and **scB** tunnels, the human and computer score respectively. They are supplied to the controller output.



## Right Bat Control

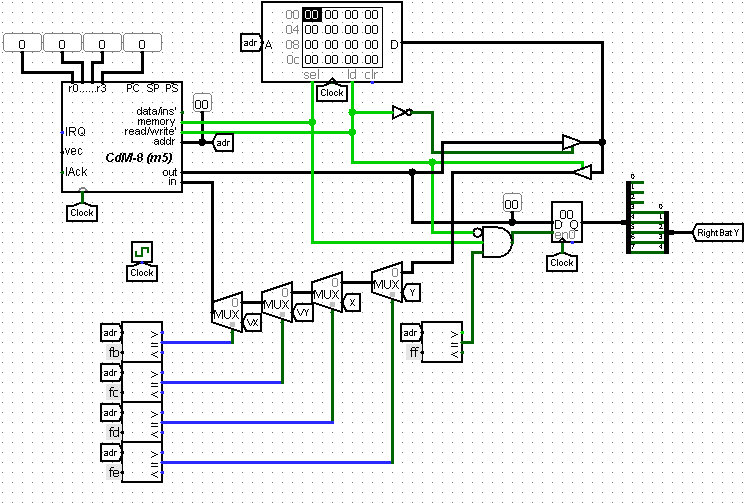
The **RightBatY** value is supplied to the input by the CDM-8 processor. This detail ensures that the bat stops on the right half of the screen. The game becomes more interesting because the computer may not have time to move the bat to the final position. The scheme also sets the default value for the right bat if the Reset button was pressed.



# Part 3: Software: CDM-8 and Control of Right Bat

Let's move on to the last element of our video game – artificial intelligence. The intelligence is made using a CDM-8 processor working with one 256-byte RAM bank.

## Memory Mapped IO

The processor receives four eight-bit values: **VX**, **VY**, **BallX** (X), **BallY** (Y) and places them in memory at addresses **0xFB**, **0xFC**, **0xFD**, **0xFE**. CDM-8 works with this data and then outputs the position of the right bat stored at **0xFF** to the Logisim register. The value is written to a five-bit **RightBatY** tunnel.

The full connection scheme is shown below in the picture.

## Algorithm and Pseudocode

Let's show the pseudocode of the algorithm for calculating the final coordinate of the right bat by the processor. The algorithm is close to the code that can be written in the CDM-8 assembler.

*Pseudocode of the algorithm*



This code is freely transferred to the CDM-8 assembler and 256 bytes of memory is completely enough.

# Further developments of TV-Tennis

Thank you for reading the documentation for our game! We tried to make the game quite interesting. However, our team does not stop there because we have some other ideas that could be implemented in the future.

1. ***Optimization of coordinate calculations****.* Unfortunately, the processor does not calculate the final coordinate in as fast a time as we would like. But we already have ideas for optimizing the code. For example, a quick division by 3 will allow us not to subtract the coordinate but to calculate the value regardless of the number. That is for a certain number of clock cycles.
2. ***Twisted punches****.* If you have played table tennis, you know that by hitting the ball with a moving bat you make the ball spin. Because of this it can suddenly fly away from the second player abruptly. We want to introduce this idea into our TV-Tennis so that we can make it even more interesting to play.
3. ***Multiball****.* A crazy mode in which you will have to play three balls at once! It will only be necessary to have time to hit the balls but it will be much more difficult to keep track of them.

1. This conversion to longVX and longVY is handled by the newVX module [↑](#footnote-ref-1)