**Introduction**

Our project is a classic TV-Tennis game, developed using Logisim digital circuit design program and the CDM-8 assembler.

The project consists of the following main components:

* A 32x32 video display that visualizes the playing field, including the ball and bats.
* A kinematic controller responsible for the movement of the ball and bats, handling collisions with the field boundaries and bats.
* A scoring system using 7-segment displays to show each player's score.
* An artificial intelligence that predicts the ball’s trajectory and calculates the most optimal hit.

Additionally, our game has the following enhancement:

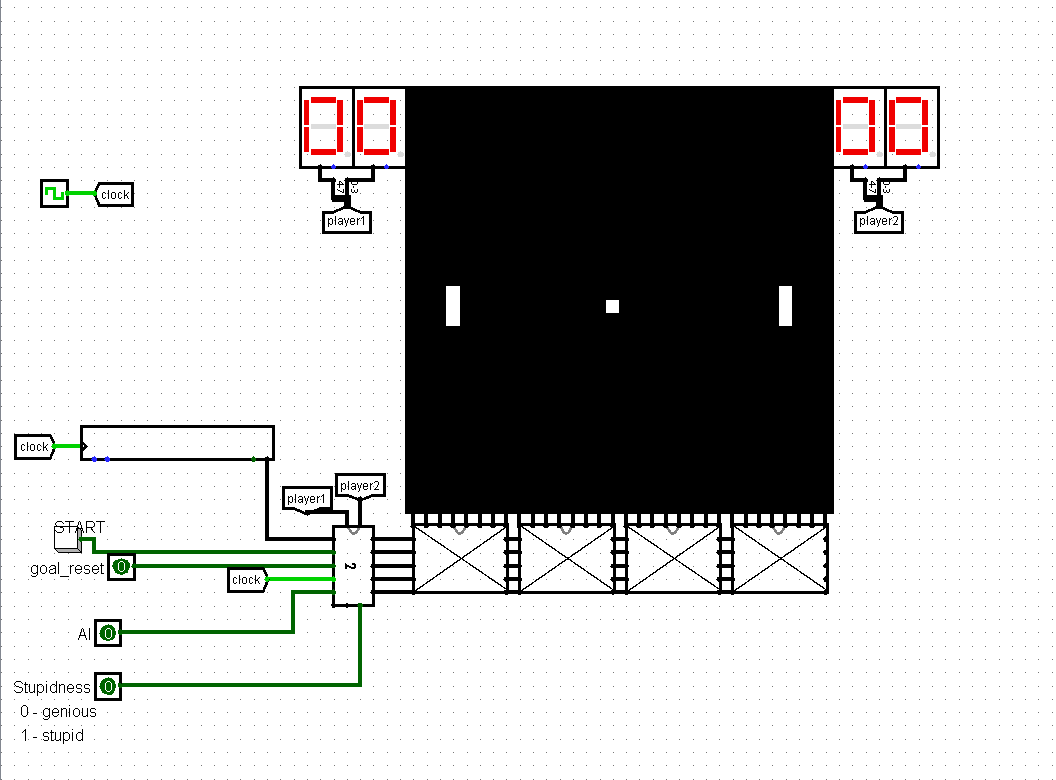
* The ability to play with two players.
* The Implementation of speed acceleration after consecutive successful hits (three different speeds).
* An Improved collision physics system for the ball’s interaction with bats, depending on where the ball hits the bat.

The gameplay is a classic arcade game with two bats and a ball bouncing off the field boundaries and bats.

The game is balanced by the manual difficulty of human control and the non-trivial logic of the ball hitting the bat, requiring the player to choose the most optimal hit.

There are two modes of “intelligence” of the computer player: "genious" and "stupid". In the "genious" mode, the AI tracks the ball flawlessly, in the "stupid" mode, it randomly makes mistakes in position.

**Game Interface**

****

The game field is a display of 32 vertical columns, each containing 32 pixels.

Three active elements are displayed on the screen: the ball and two bats. The ball is represented by a single active pixel, whose coordinates are determined by the values ballX(column number) and ballY(row number). The ball can appear at any point on the screen. The bats are shown as vertical strips made up of three consecutive pixels and are always located in fixed columns: the left bat is in column 3, the right bat is in column 28. The position of each bat is defined by the coordinate of its bottom pixel(l\_bat and r\_bat respectively). Thus, four 5-bit values are required to control all game objects, which are generated and sent to the display input.

To visualize the score, two 7-segment displays are used, showing the score of player 1 on the left and player 2 on the right. These displays receive data from a controller that increments the score of the appropriate player when the ball is missed.

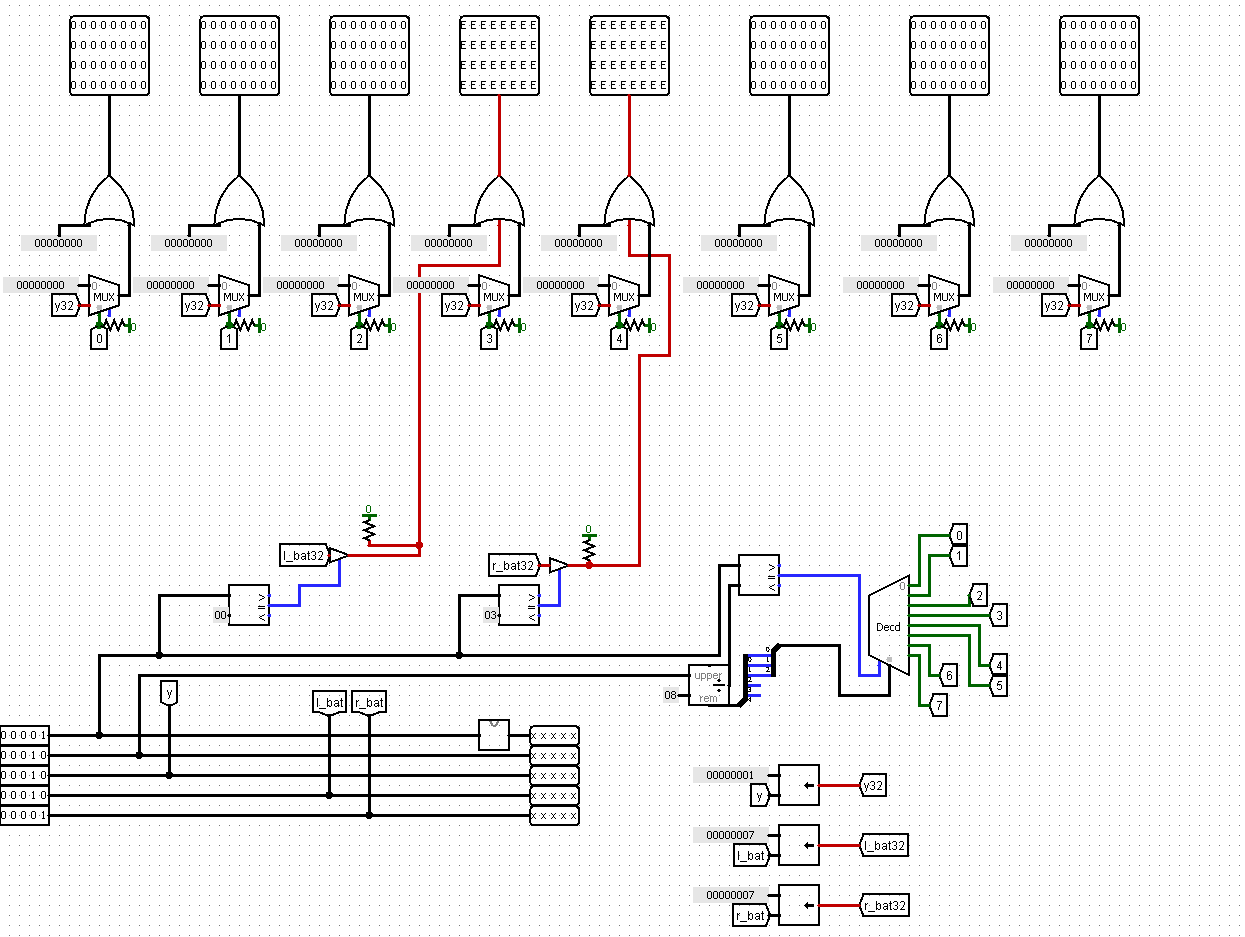
The AI button allows you to turn on/off artificial intelligence for the right bat. The goal\_reset button allows you to turn on/off the processing of the scored ball (if turned on, after the goal is scored, the bats return to their initial positions, the ball is placed in the center of the field, moving towards the player who missed).

An additional feature is the Stupidness switch, which sets the “intelligence” level of the AI bat. It has two modes:

* 0 — genious (smart mode): the AI tracks and predicts the ball ‘s movement flawlessly, positioning the bat accurately for every hit.
* 1 — stupid (stupid mode): the AI makes random errors in position by ±3 units.

This allows you to adjust the AI's behavior depending on the desired difficulty of the game.

**Display**

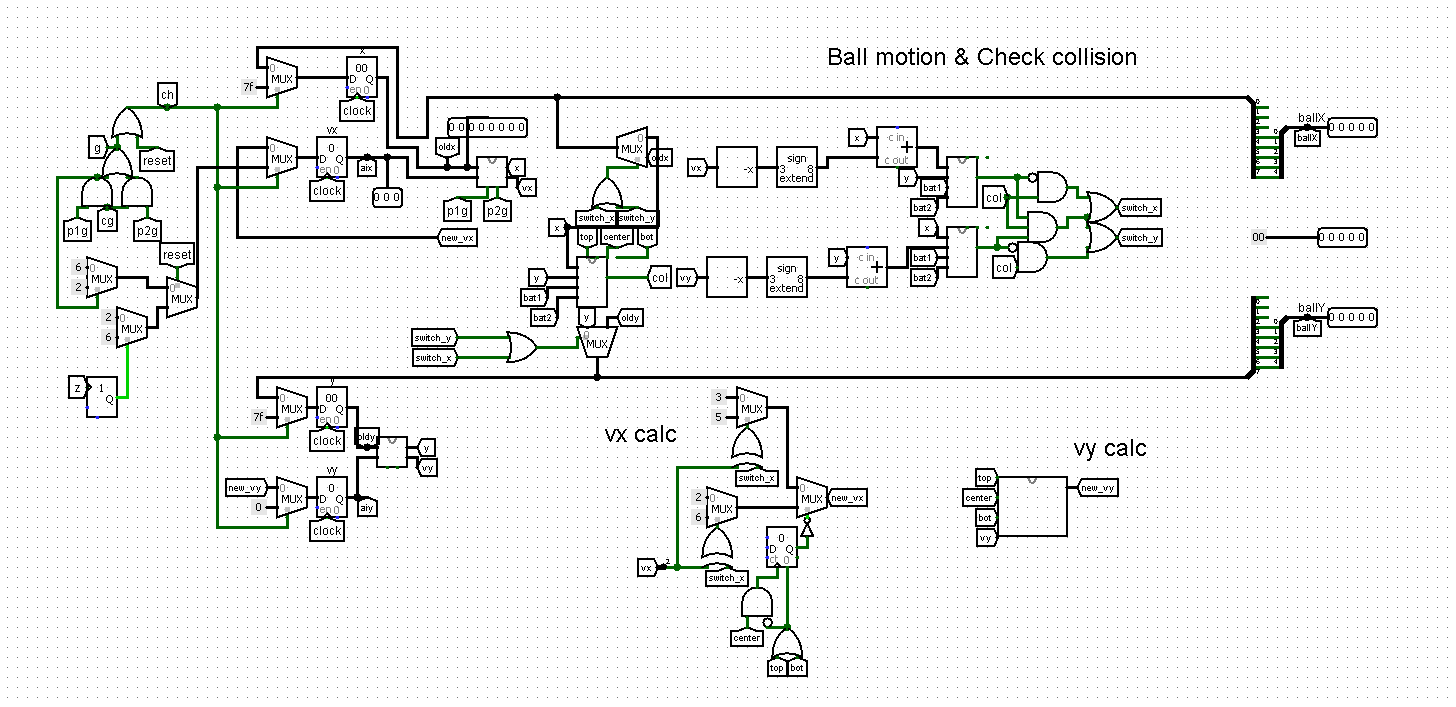


The screen output works as follows: we have 4 display circuits connected to each other and linked to the main screen. Each display circuit has 5 inputs: display\_id, the ball’s x and y coordinates, the y coordinate of the bottom pixel of the left bat, and of the right bat.

We process the ball’s coordinates as follows: if x // 8 == display\_id, then we output the value 1 << y on the line with index x % 8 to turn on the pixel in the desired row.

The bats are drawn in a similar way: since their x coordinate is fixed, we do not need to check it, we output the value 7 << y to enable three vertical pixels and display the bat.

**Ball motion**



We store the ball’s x and y coordinates in two separate 8-bit registers. Its horizontal vx and vertical vy speeds are stored in two 3-bit registers.

Allowed values for vx are 2, 3, -2, -3.

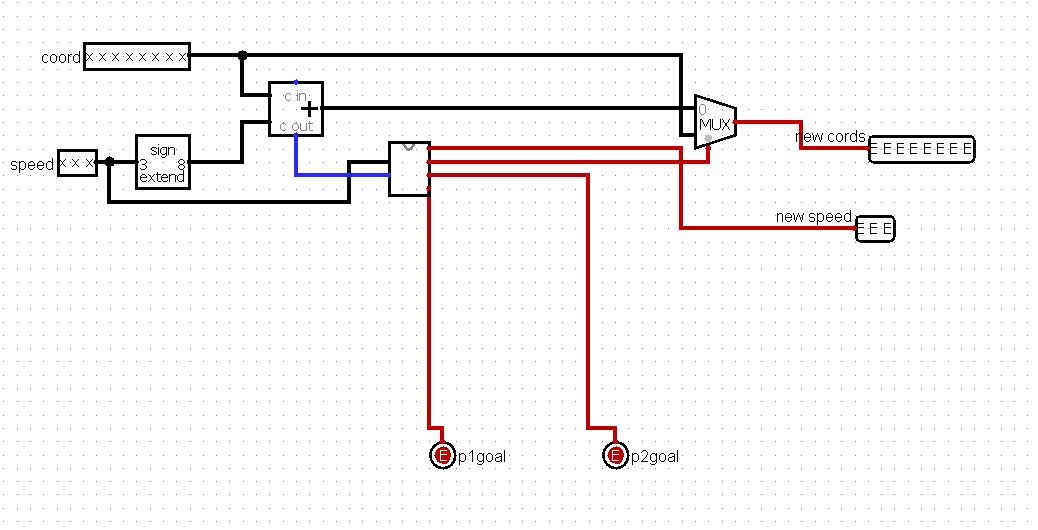
For vy: 1, 0, -1.

The input of each register can receive either a value from the previous clock or a predefined constant depending on the current state of the game.

At the start of the game, the ball is placed in the center of the field with coordinates (127, 127), with a vertical speed vy = 0 and a horizontal speed vx = 1 or vx = -1, randomly selected.

Then, the ball’s movement, based on its current speed and coordinates, is processed using the following circuit.

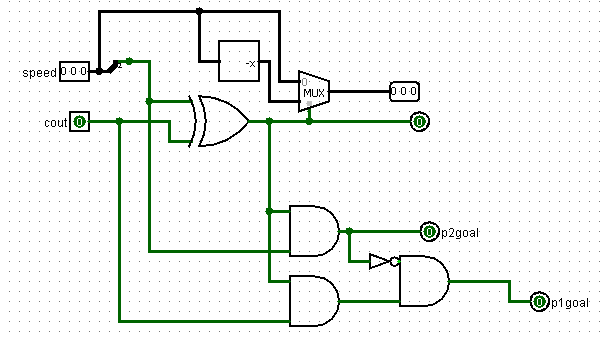
**Motion**

****

Two values are given as input: the coordinate and the speed. We add up the speed and the coordinate, and if there is an overflow on the right or left, we reverse the speed sign, trigger a goal signal, and keep the coordinates unchanged. Otherwise, the updated coordinate and the same speed are output.

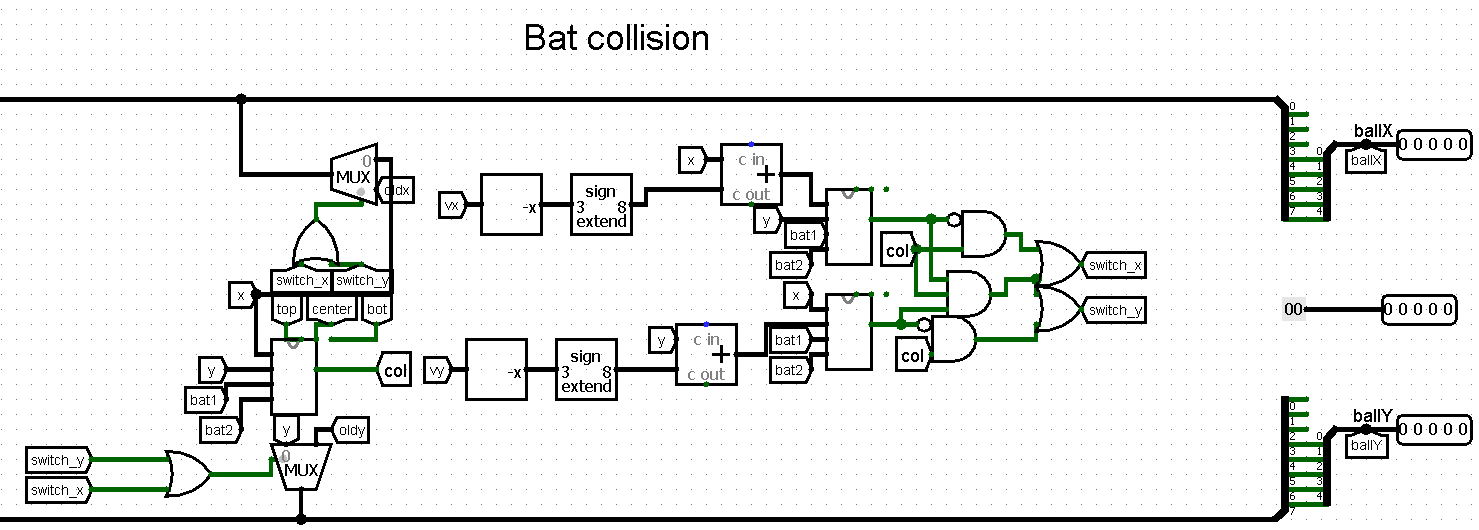
Next, let’s consider the **reflect** circuit, which is part of the **motion** circuit.

**Reflect**

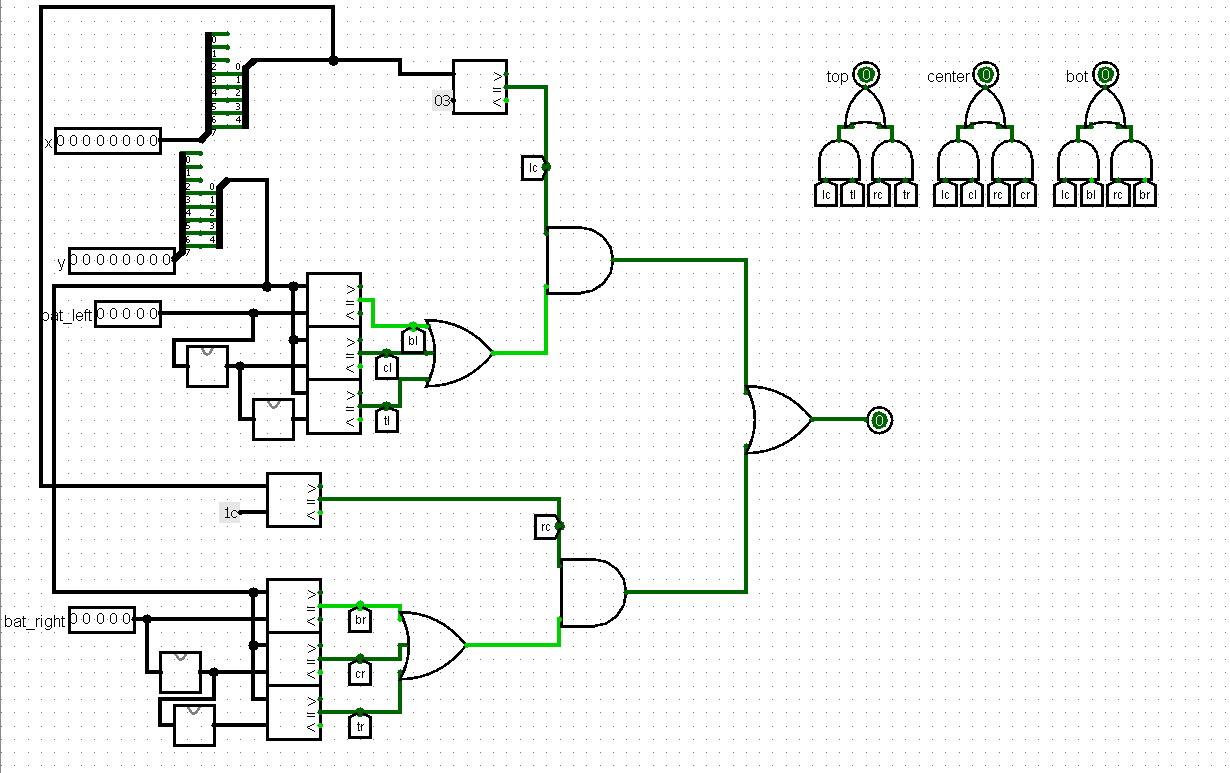
****

In this circuit, two values are input: the ball’s speed and the carry bit from the adder that sums the speed and the ball’s coordinate **motion** circuit. We use the XOR gate to determine if an overflow has occurred. The outputs are the updated speed, a bit indicating that an overflow happened, and two bits indicating which player scored a goal.

Next, we handle the possible collision between the bat and the ball.



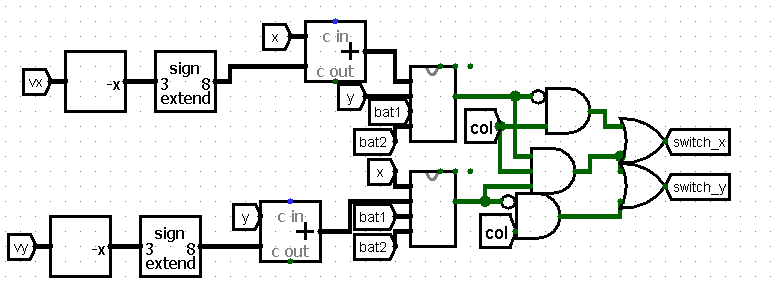
First, we send the data to the **check collision** circuit.  
  
**Сheck collision**

****This circuit is responsible for detecting collisions of the ball with the bats during the game. It analyzes the current ball’s coordinates and compares them with the positions of the left and right bats.

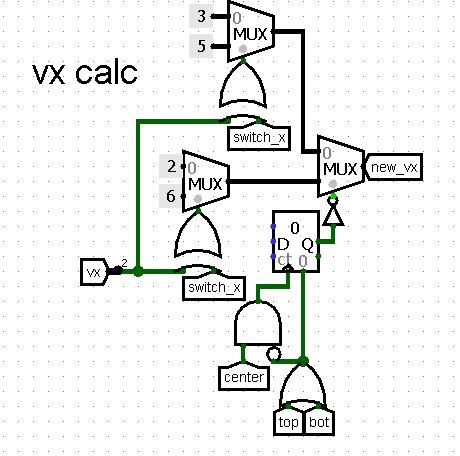
First, we check whether the ball is in the column with the bat: column 4 for the left (in the circuit x == 3) and column 29 for the right (x == 28). If the coordinate matches, the signal lc (left column) or rc (right column) signal is activated.

Since each bat occupies 3 pixels vertically, we compare the ball’s vertical y coordinate with three values: the coordinate of the bottom edge of the bat, this coordinate +1 and +2. We perform such checks for both bats, resulting in signals tl, cl, bl (top/center/bottom left) and tr, cr, br (top/center/bottom right), indicating where the ball hit.

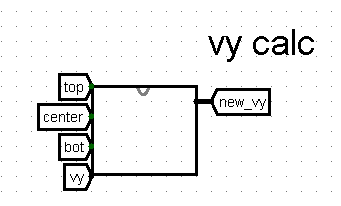
For each bat, we use a logical AND between the horizontal collision flag (lc or rc) and the logical OR from all three vertical collision flags. This ensures that a collision is detected only if the ball is in the correct column and at a height that matches the bat’s area. Then we combine the signals from the left and right bats using OR, and form the final signal.



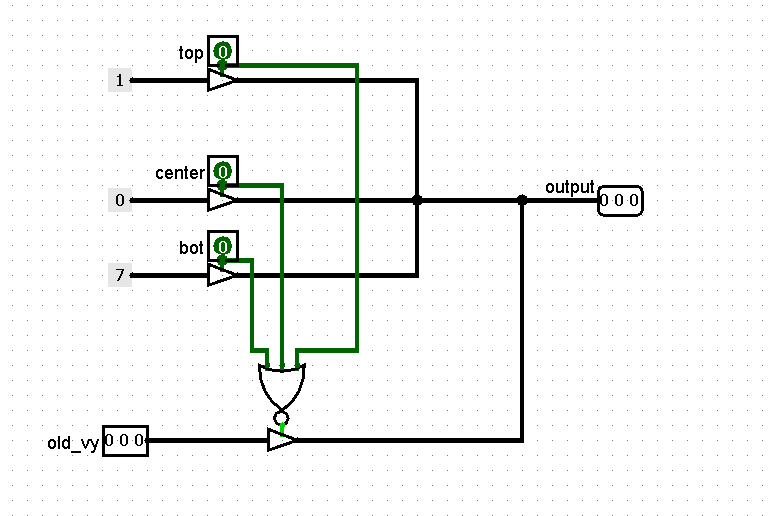
Next, we get data on which coordinate caused the collision.



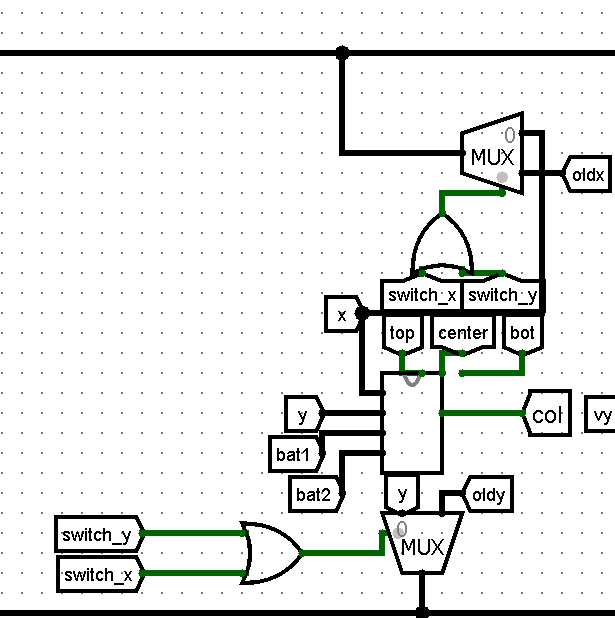
After that, we analyze the collision data, determining a new vx value from the range of -3, -2, 2, 3. If the ball hits the edge of the bat, the speed is simply inverted. If it is in the center, it gets the maximum speed (3 or -3) and keeps it until it hits the edge.



Changing the vy speed.



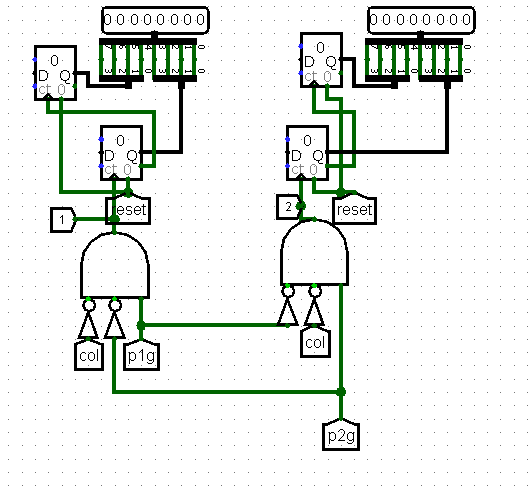
This circuit determines in which part of the bat the ball hit, depending on this, gives the output speed. If the ball hits the top of the bat, then vy = -1, center - vy = 0, bottom - vy = -1.



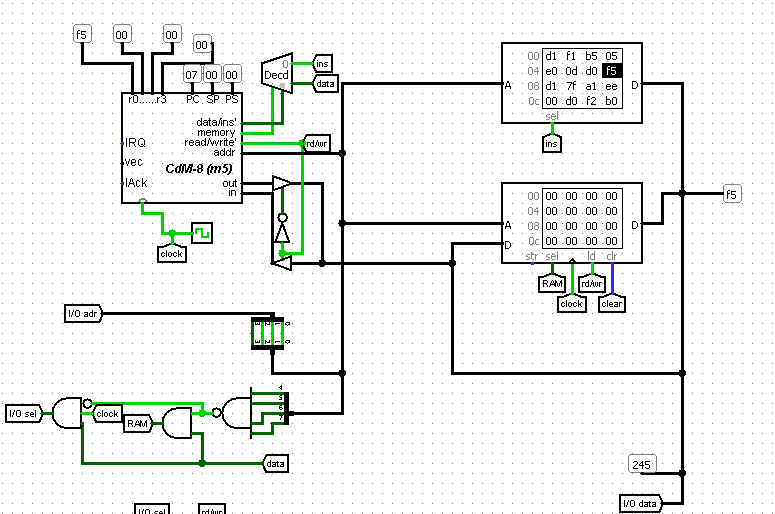
Next, we load the updated ball coordinate data. If the ball collided with the bat, we load the coordinates from before the collision, but with the updated speed.

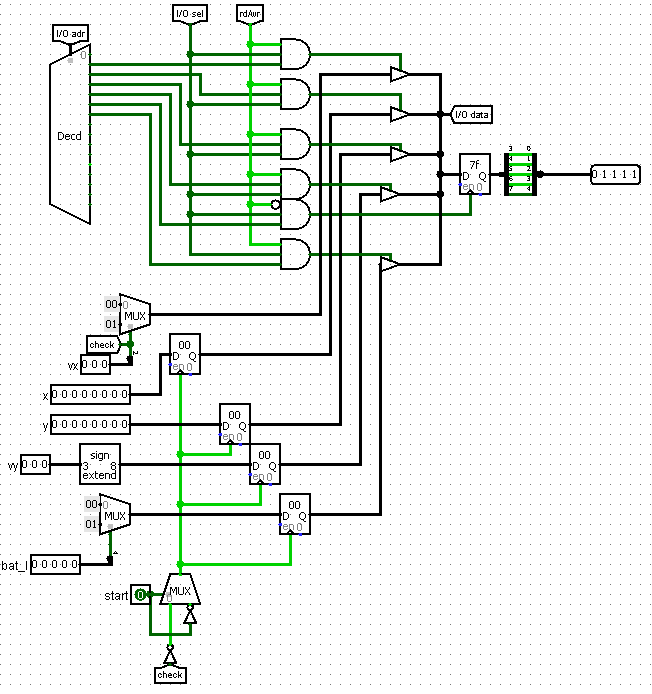
As a result, this collision logic ensures that the ball correctly bounces off the bat — it doesn't pass through it but reflects properly. Additionally, it accurately handles cases when the ball hits a corner or the top edge of the bat: in such situations, only the vertical speed vy changes, making the ball's behavior more realistic.

**Score Increment**

****

While handling collisions, we count the points scored using a Counter, representing the score in decimal form. We account for possible false goal detections during bat collision and avoid this. Since decimal values cannot be directly displayed on a Hex Digit Display, we implemented logic using two counters, the lower 4 bits represent the units, and the higher 4 bits represent the tens.

**Integration CdM-8 with Logisim**



By integrating CdM-8 with Logisim, we input 6 values and get one output - the 5-bit coordinate of the bottom cell of the bat. To process the data in software, we load them into registers, whose trigger is activated in two cases: changing the speed from negative to positive or starting the game.

**Software**

The artificial intelligence calculates the Y-coordinate of the bottom pixel of the right bat. The following values are input: 0xF1 is the most significant bit of the ball's X speed, 0xF2 is the ball's X coordinate, 0xF3 is the ball's Y coordinate, 0xF4 is the most significant bit of the ball's Y speed, 0xF6 is the Y coordinate of the bottom edge of the left bat.

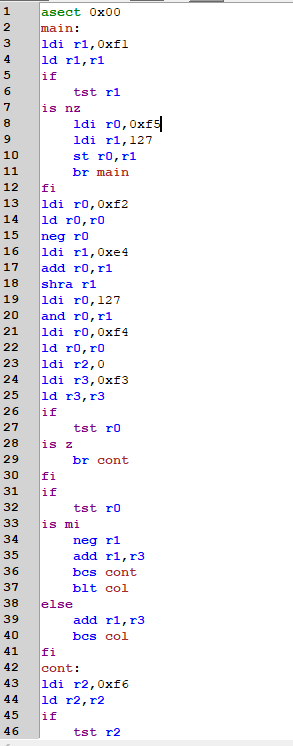
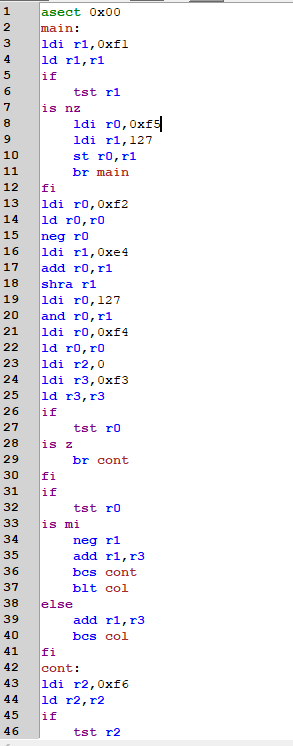
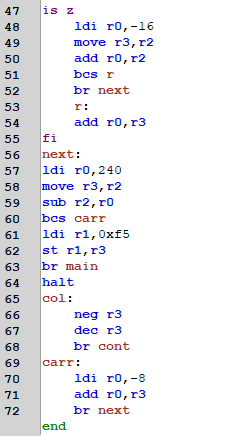
First, we check the most significant bit of the ball's X speed. If it is non-zero(meaning the ball is moving toward the left bat), the bat's position is set to the center of the field.

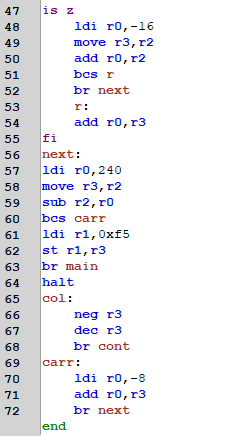
If the ball is moving to the right, the program proceeds to calculate the point where it can hit. We load the ball's X coordinate and calculate the distance to the right bat, which is located at position X = 224. We divide this distance by two to get the absolute change of the ball in Y.

Next, using the direction of the Y speed, we calculate the new Y position, working with overflows, that is, collisions of the ball with the upper or lower boundary during the calculations.

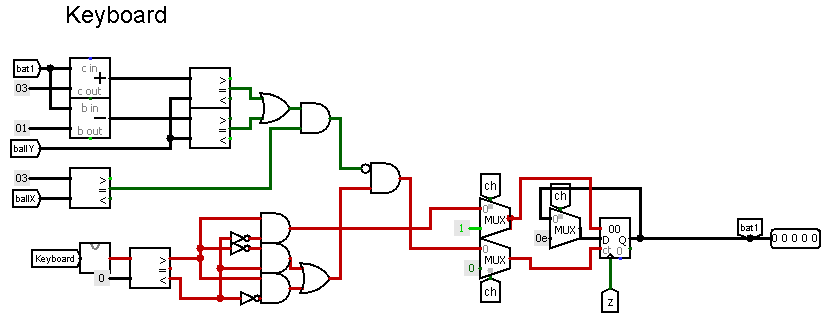
Then, we load the coordinate of the left bat. If it is located in the lower half of the field, adjust the final coordinate so that the ball hits the top of the bat, again taking into account possible overflows.

We also handle the case where the right bat itself goes beyond the top boundary and decrease the coordinate until the bat fits within the game field.

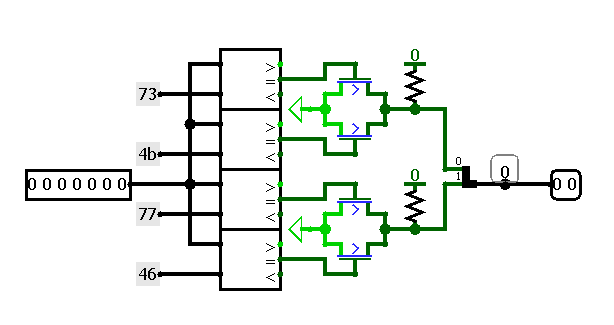




**Control (input).**

****

The game is controlled using the Keyboard circuit, with its output connected to the controller. In two-player game mode, the ‘W’ and ‘S’ keys are used to control the left bat, and the ‘O’ and ‘L’ keys are used for the right bat. The input is processed using the following circuit.



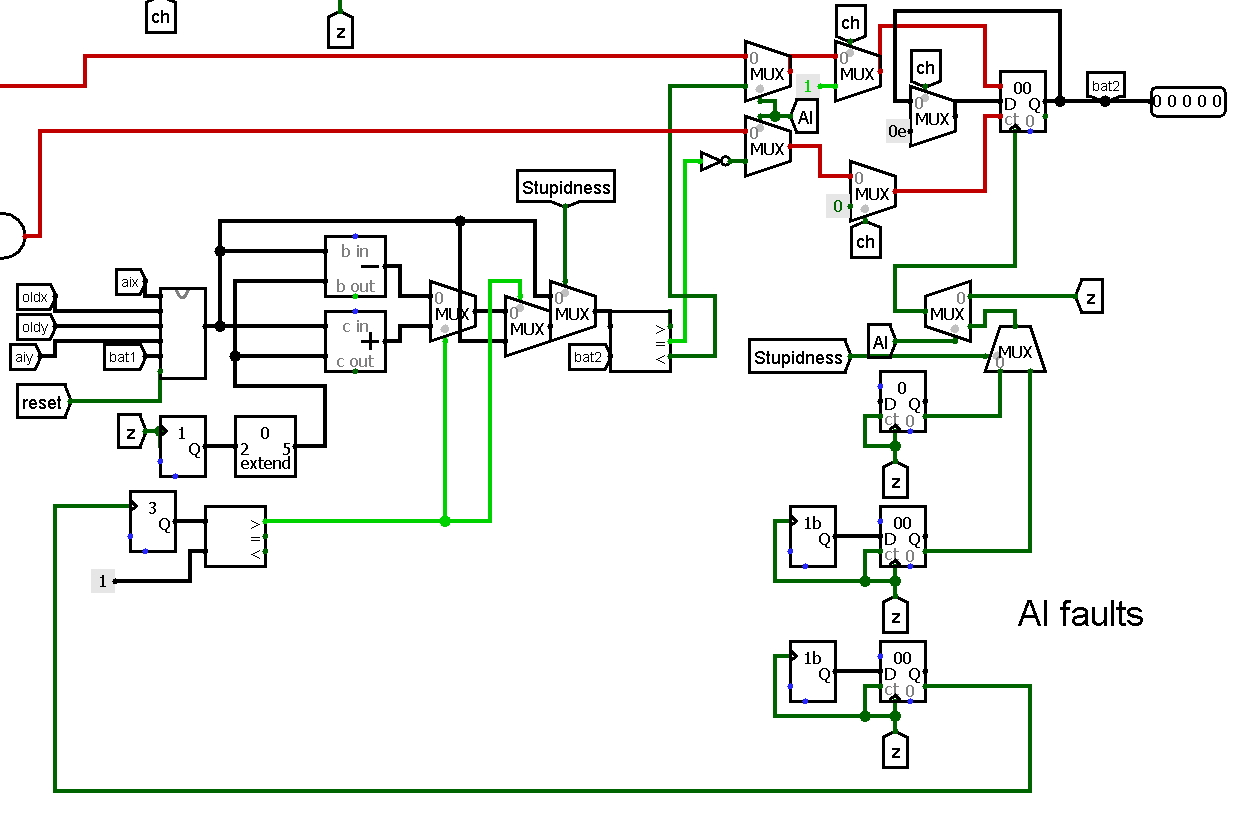
Here, if the input is ‘W’ or ‘Ц’, then the circuit outputы the 2-bit value 01, if ‘S’ or ‘Ы’ then 10, if neither is 00.

Returning to the main controller circuit : after processing the keyboard input, we compare the output value with 0 to determine the direction of movement — up or down. Then we pass this value to the counter, which accumulates the final bat coordinate in 5-bit format. Overflow does not occur, because the counter specifies the mode to stay at value.

Additionally, the controller includes a protection mechanism to prevent the ball from getting "stuck" inside the bat. Previously, if the ball was directly above or under the bat and it was moving towards him, the ball could be blocked inside. Now, a block in the upper part of the controller circuit that compares the ball coordinate with the bat coordinate. If the ball is located directly above or under the bat, movement in that direction is blocked, preventing the ball from getting stuck in the bat. Going off the screen and overflowing are not a problem, since the coordinates of the ball are already processed taking into account the boundaries of the field.

In AI game mode, the output for the right bat is blocked and replaced by the output from the AI circuit.

**AI faults**



When playing against the AI, there are two modes: genious and stupid, and the AI input and stepping circuit, the stupid mode is implemented. If stupid is enabled, the AI makes mistakes in each game tick by randomly determining the delay until its next move from 1 tick to 32 (in genious mode, the delay is fixed and equal to 1 step for every 16 ticks). Also, at a random moment unrelated to its steps, the AI makes a mistake in determining the desired position by ±3 bits including 0, so even when idle, it constantly recalculates its target and moves around it slightly.

**AI handler**

The AI component outputs only a number, which is the estimated position of the ball at the moment of contact with the right bat. In the genious mode, the input bypasses the random transformation handlers and passes to counter, which begins to accumulate the current value incrementally or decrementally to the introductory value once for every 16 ticks (implemented as follows, in another counter, each tick accumulates one value for 4 bits, when the overflow bit creates 1 clock cycle, which triggers accumulation for the value of the right bat controlled by AI).