# NOVOSIBIRSK STATE UNIVERSITY

**Profile: Computer science and System engineering**

**TERM PAPER**

**Statsenko Ilya Romanovich**

**Fedotkin Nikita Alekseevich**

**Ermakov Alexandr Vyacheslavovich**

**TANKS**

# Introduction

The project is a realisation of the game "Battle city". For its development Logisim was used, as well as the CDM-16 processor, which is responsible for controlling the enemy tank.

The rules of the game compared to the original version are simplified: the game lacks the player base and the number of lives or strength points of the tank. One hit on a tank means its destruction. The enemy is only one. He is located in the centre of the upper part of the map, and the player - in the centre of the lower part of the map.

A tank cannot shoot if there is another of its shells on the map. To win, it is necessary to destroy the enemy tank without getting under its shot.

The map has walls drawn with green pixels. The walls block movement and destroy projectiles on impact.

The tank is controlled using the "Keyboard" module in Logisim. To reset the game, press the "Reset" button next to the keyboard.

Tank Control:

* Forward - W (Ts)
* Back - S (S)
* Left - A (F)
* Right - D (B)
* Shot - SPACE

# Project structure and description of schemes

## Main scheme

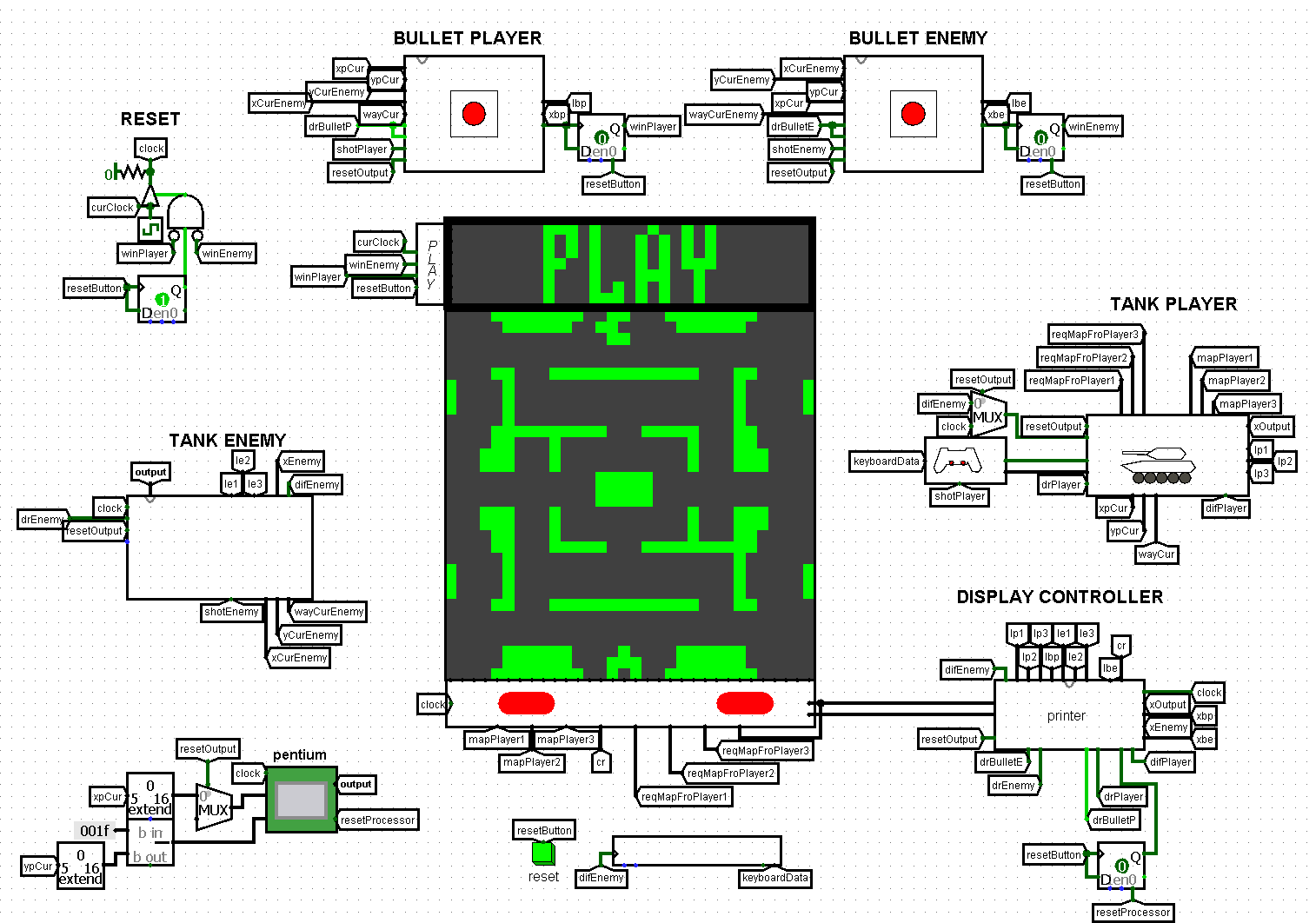
The scheme contains the main and auxiliary screens, schemes of tanks, shells, as well as rendering and screen control. Connections of schemes correspond to their description, except for some points, which are described below.

To prevent the game from starting instantly when the clock generator is switched on, a circuit labelled "reset" was built, which blocks the clock until the "reset" button is pressed. However, the scheme of rendering the additional screen does not depend on it, because it must work even if the game state changes. Also the scheme limits the clock cycles at the moment of victory of one of the tanks to avoid errors.

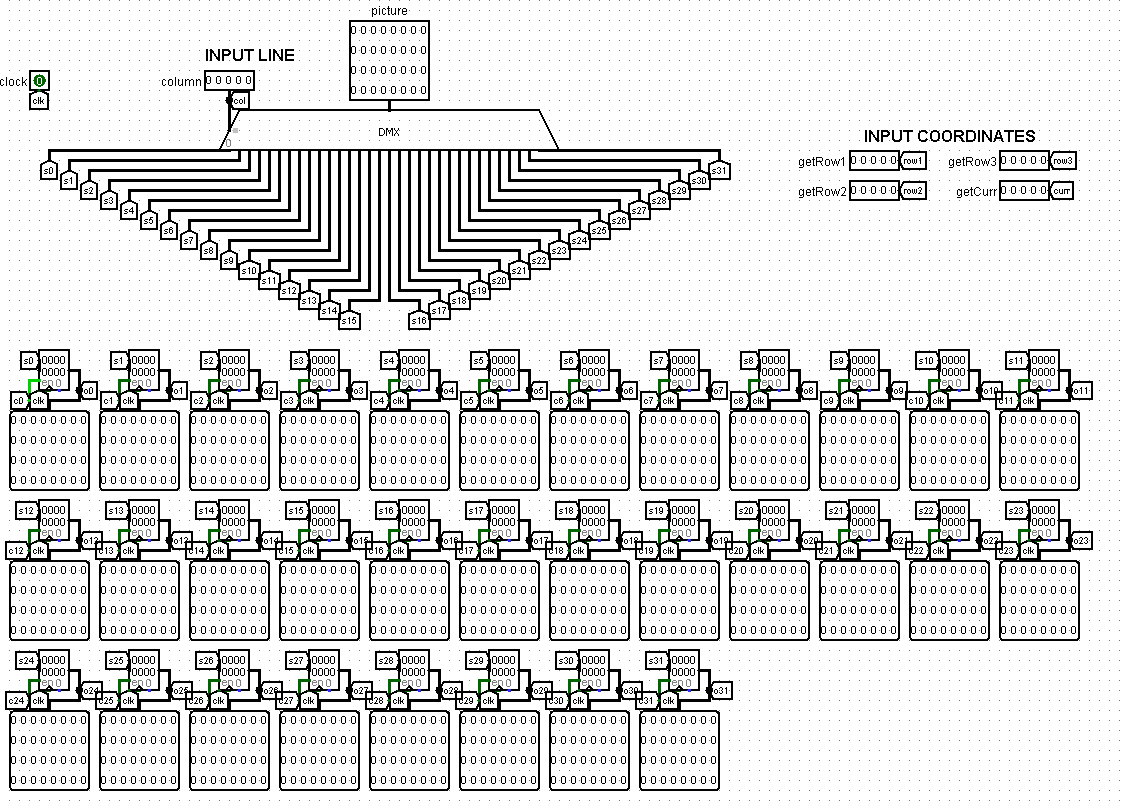
Instead of a clock, a contact was connected to the projectile circuits to update the state of the bullet. This was done to limit its speed, as the change of this signal is much slower than that of the clock.

A similar solution was applied to the player's tank, but we connected the player's tank state change to it. This was done to prevent the player from walking faster than the opponent. However, at the moment of state reset we connect the player's circuit to the clock generator, for correct reset of the circuit state.

The signal about the victory of tanks is recorded in the trigger and stored there until the user presses the "reset" button. This is done so that the user can restart the game at any convenient moment.



## Screen control scheme

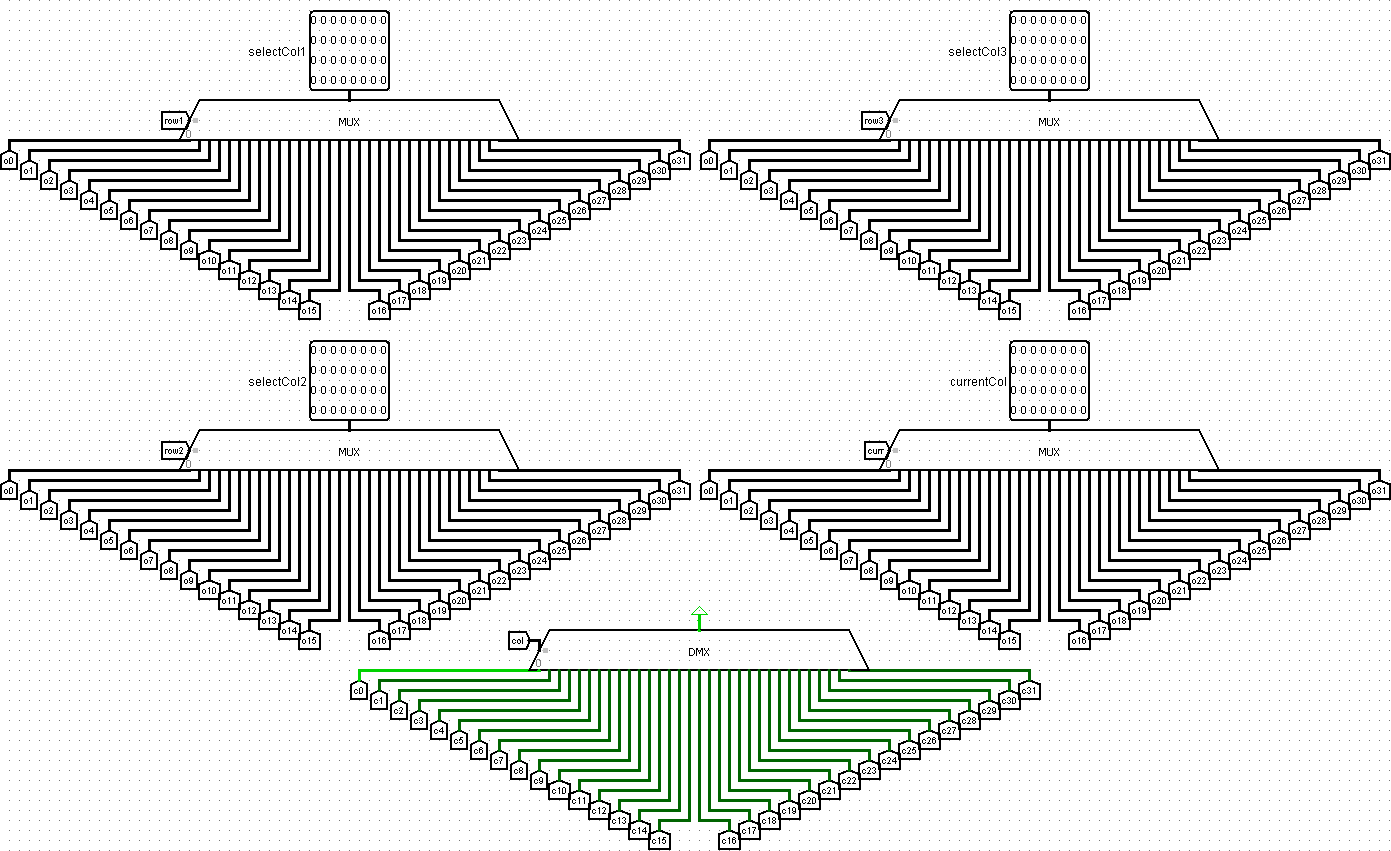


At the top of the circuit under the "INPUT LINE" label is a demultiplexer that sends the image from the "picture" input to a register indexed "column". This allows one column to be written per clock cycle.

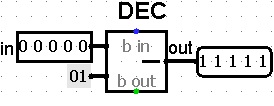
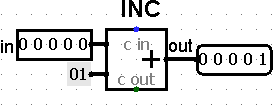
To the right are four five-bit inputs - these contain the column coordinates needed by the *player's tank scheme* (getRow1-getRow3), as well as the *screen rendering scheme* (getCur).

Below are 32 registers with the same scheme of connection of signals to them. For example, consider the register responsible for the zero column. The register is inputted with pin "s0", which contains the screen column data to be displayed. Pin "c0" allows the values to be written, preventing constant updating unnecessarily. The "clk" pin is connected to the clock. Contact "o0" is responsible for outputting the column data to pass it to other circuits or to display it on the screen.

Below that are four multiplexers that select the screen columns required by other circuits, and a demultiplexer that determines the register that needs updating.

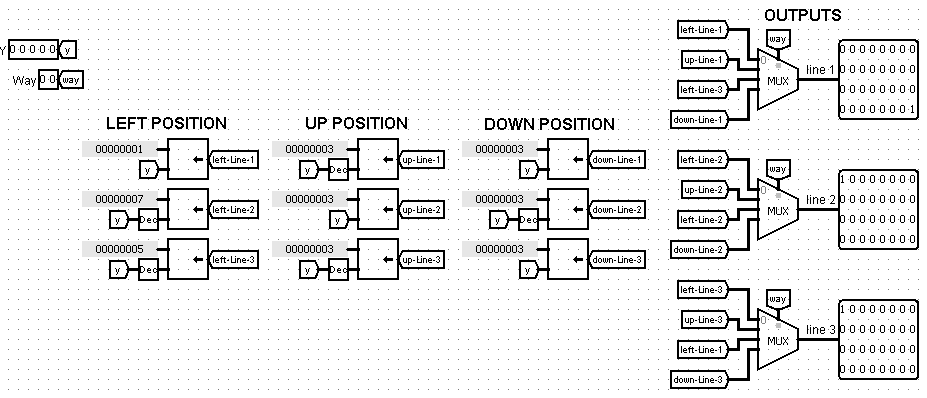


## Tank drawing scheme

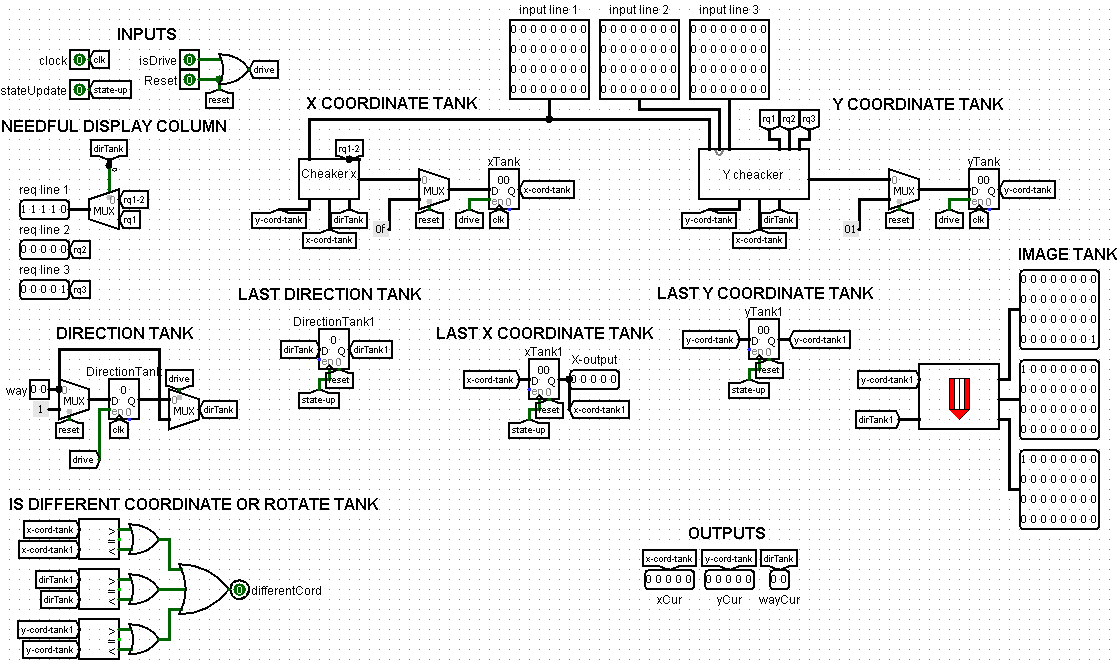


In the process of designing the scheme, *incrementer* and *decrementer* schemes were added. Above is the selected image of the tank in the "up" direction. At the top of the circuit are inputs containing the direction of the tank and the Y coordinate of the tank. The image is rendered through bit shifts of predetermined numbers.

Under the labels "left position", "up position" and "down position" there are schemes that draw the necessary image. The scheme for drawing the tank in the position "right" is not necessary, because it can be obtained from the image of the tank "left". At the output we select the appropriate image depending on the direction of the tank.



## Scheme player's tank



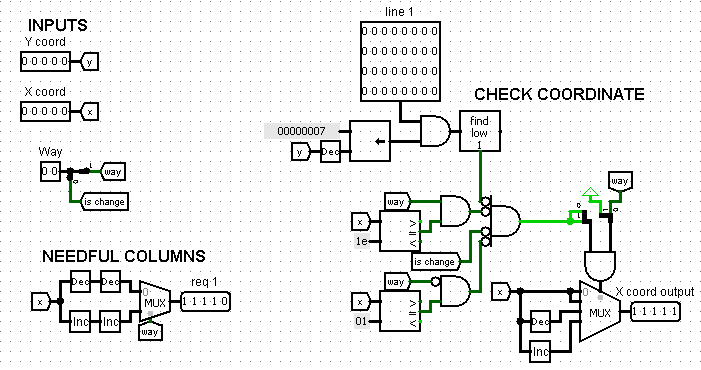
In the process of development we encountered a number of problems. The first of them arose after building a prototype circuit in which image rendering was performed through constant redrawing of objects. We noticed that Logisim was not processing the constantly updating image at a fast enough rate, causing flickering. To solve this problem, it was necessary to introduce several additional registers. One of them stores the previous value of the X coordinate to track the position of the tank before the update. Triggering these registers, occurs with the help of the signal "stateUpdate", which sends the *scheme draws the screen*. This signal indicates that the scheme erased the old image and is waiting for a new one. On the same principle work registers for storing the previous value of the direction and the previous Y coordinate.

*The "x cord checker"* and *"y cord checker"* circuits, which will be discussed next, are used to check the possibility of movement. They send to the "req line" outputs the index of the required screen lines, and afterwards receive them through the "input line" inputs. Since the circuit does not always need 3 screen columns, then one input is used to get the required information.

Before the registers containing the current directions and coordinates of the tank, there is a multiplexer with a selecting bit "reset", so that you can return the tank to the original coordinates. On these registers on the permissive input is "logical or" values "isDrive" and "reset", so that when resetting the state registers can be overwritten (because in the process of resetting the state of the tank is stationary).The output of the image of the tank is carried out using the *scheme of drawing the tank.*

Also outputs with current coordinates of the tank and its direction were added for further sending to the *projectile control scheme.* The second problem is that in the *screen rendering scheme it* is not possible to find out if the image of the tank has changed. To solve this problem, we send a bit of "differentCord" with information about the change of state. This bit is equal to one if the old and new position of the tank are different.

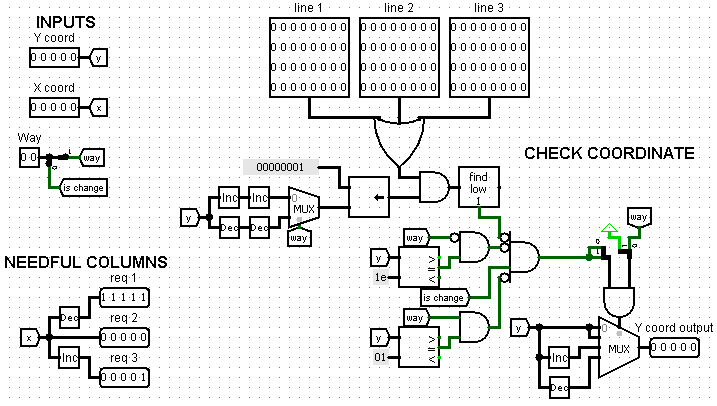
## Schemes for checking tank coordinates



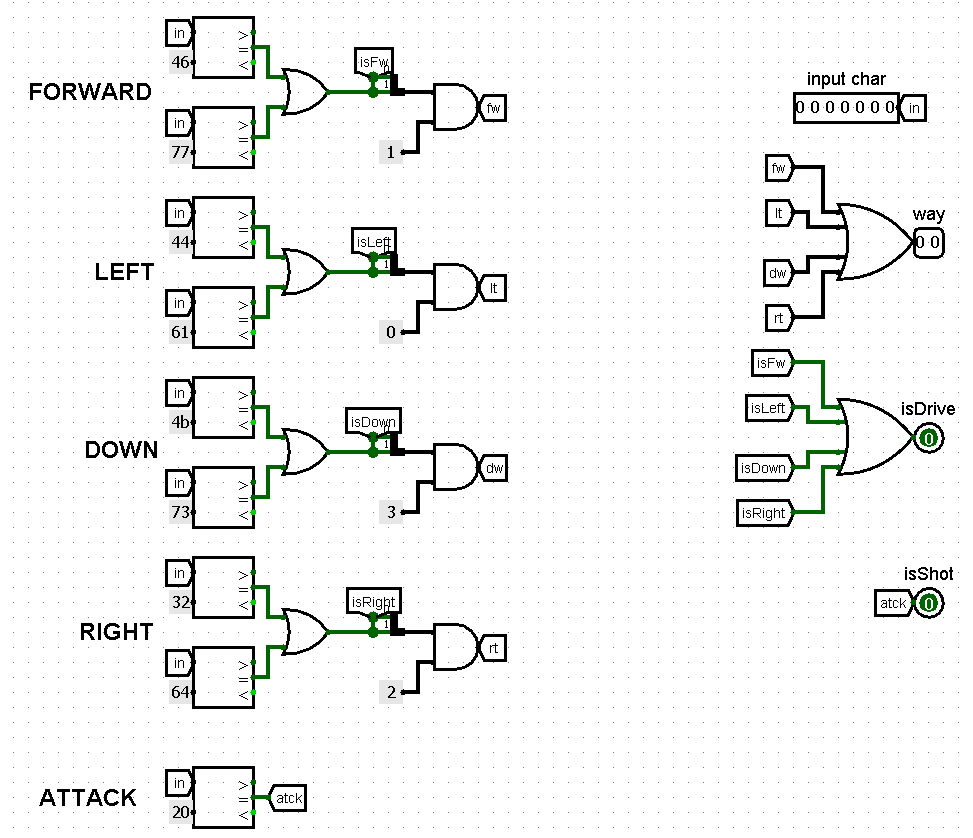
Schemes on the input are given X and Y coordinates of the tank, its direction and the necessary number of columns of the screen. At the output they send the indices of the required columns of the screen and the corrected coordinate. If the movement is possible, the coordinate is changed, if not - the original value is stored.

Let's consider the *scheme of checking the x coordinate*. The check is carried out by "logical and" between the hitbox of the tank and the column of the screen. If the result is not equal to zero, then in the direction of movement of the tank is a wall. We also look to see that the tank is not trying to go outside the map boundaries. For example, if it is at point (1, 1) it cannot go to the left. A zero bit of the "way" input with a value of 0 indicates that we have a direction of either 00 or 10 - that is, we are moving along the X coordinate. If the value of "logical and" between checking for boundary exit and no walls, and the inverted value of the zero bit of "way", then we can use the multiplexer to select the desired coordinate change using the first direction bit.

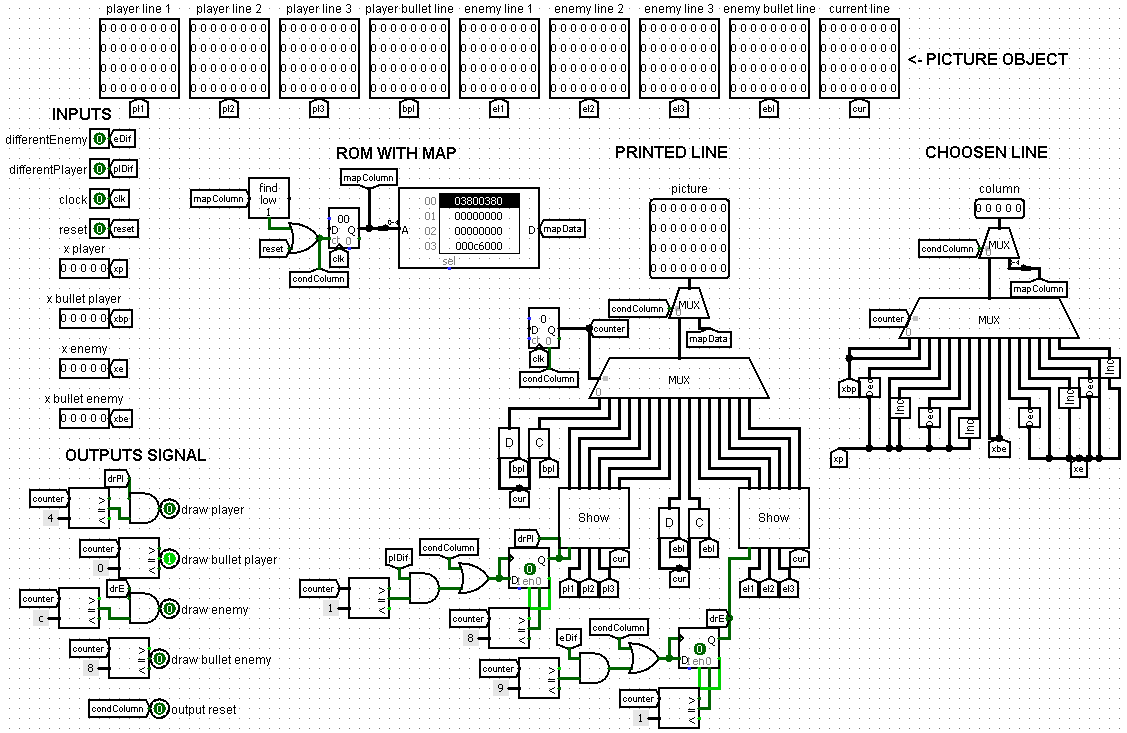
*The scheme for checking the Y coordinate* is constructed similarly, but with one difference. When analysing for obstacles, three columns of the screen are combined into one. This is necessary to make sure that there are no walls in the direction of tank movement.



## Tank control scheme

This circuit takes as input the ASCII code of the entered character, and then turns it into the corresponding control signals with the help of comparators. If, for example, the character code is the same as a movement to the right, the output of the "logical and" will be 2, and all other "logical and" will be 0. This is done so that the desired direction can be obtained at the output of the "logical or". On the bit "isDrive" is fed the result of "logical or" with all directions, so that you can determine whether the tank is moving or not. The signal "isShot" is transmitted directly from the comparator.

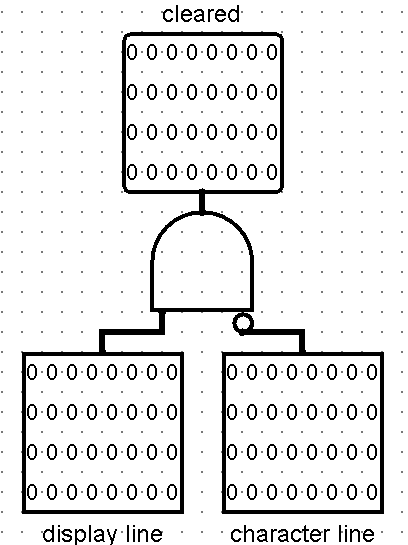
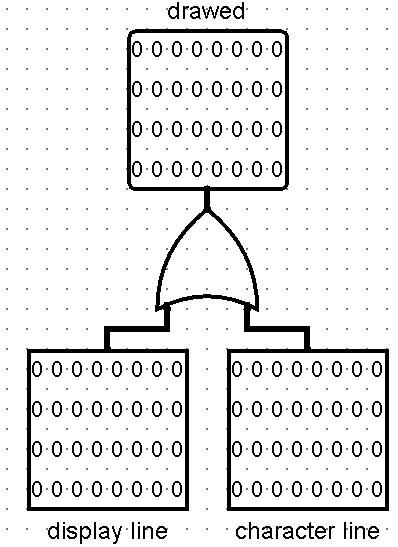
## Screen rendering scheme



To solve the problem related to screen flickering 2 additional inputs were added with information about changing images of the player's tank and enemy. Outputs for tanks and projectiles were also added, signalling that the scheme has finished erasing an object, now it is necessary to update data about it before it is drawn again.

Additional schemes were created for sketching the image *"clear line"* and for drawing *"draw line"*. They take as input the current screen column and the screen column to be converted. On the output they send the modified screen value. To draw the image, 2 multiplexers with four select bits are used. This is enough to update all the objects on the screen in 16 clock cycles. One multiplexer selects the number of the column to be updated, the second selects the data to which the column should be replaced.

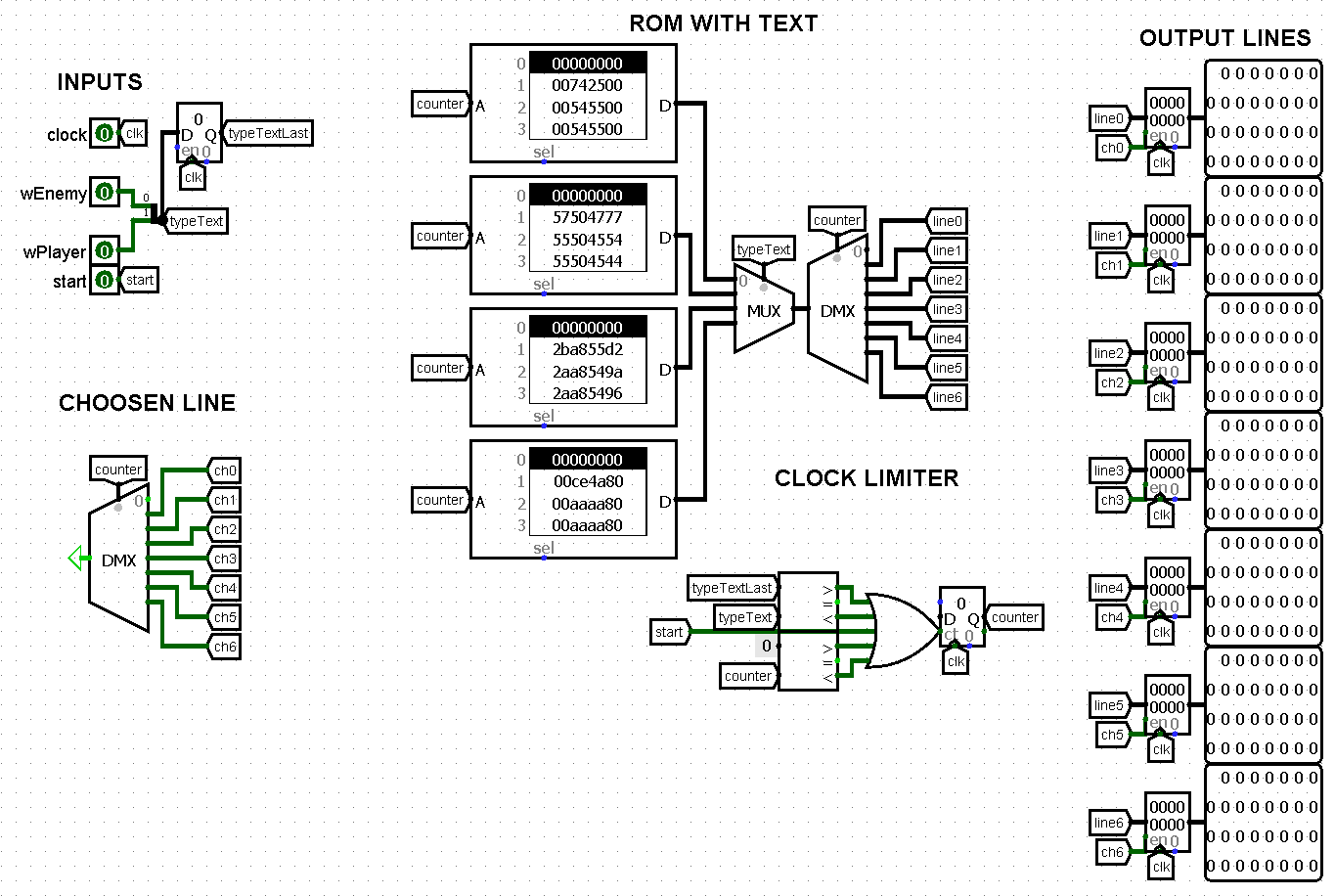
Auxiliary circuit "show" depending on the signal on the bit "diff" either sends the current value of the screen, or the image of the tank. A set of elements whose output is connected to this bit does the following: before drawing the image it checks whether the image has changed. If so, then allow the circuit for the necessary number of clock cycles to update the image. To prevent failures during processing, a trigger is used to save the signal value until the drawing is complete.

The map is drawn as follows: when the "reset" signal appears, we allow the counter to increment until the value is zero. This counter selects the screen column from ROM. For correct synchronisation with the processor the loading time of the map was increased, so that the tank does not remain inactive after the reset is completed.

## 

* 1. "show".

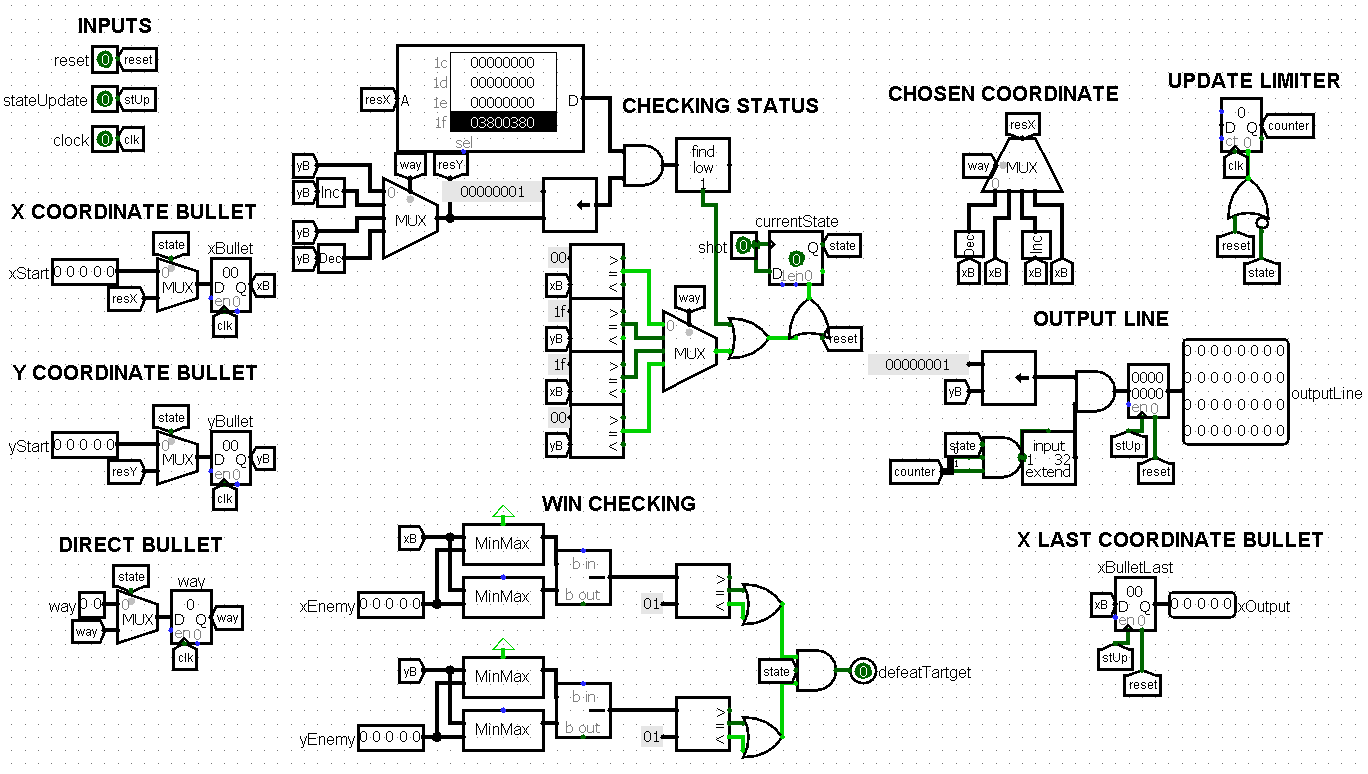
## Control scheme for the auxiliary screen with game status information



The scheme displays the game state on an additional screen. The screen is accessed line by line, and the lines themselves are stored in registers. The bits "winEnemy" and "winPlayer" are used to build a two-bit number, which is written to the register to store the game state. If the value at the input of the circuit and the output of the register are different, we allow the counter to increment until the value is zero. This counter selects the screen line from the ROM. We also added the possibility to start drawing at the moment of program start, as the register and inputs will be 0 at that moment, and the circuit will mistakenly think that it is not necessary to draw the text.

After building the prototype circuit, we encountered a problem: before pressing the "reset" button, the first row of the screen was drawn. This was due to the fact that the initial values of the circuit are equal to zero, so it seemed that the zero column was being selected. The solution to the problem was to shift the outputs in the demultiplexer by one. (if before we used only 0-6 outputs inclusive, now we use 1-7 inclusive)

## Projectile control scheme



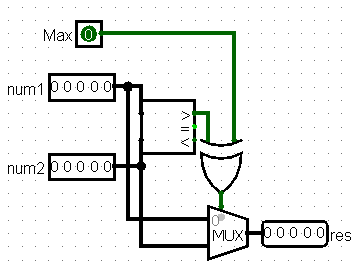
The scheme is responsible for the operation of the projectile for the tank. It is used both for the enemy tank and for the player. The signal "stateUpdate" comes from the *screen rendering circuit* and informs that the image was erased and now it is necessary to update its state. When the player has fired a shot, the signal is sent to the "shot" bit. The unit is written to the trigger and stored there until the projectile hits the target, flies off the map, or collides with a wall. This prevents the shot from being reactivated while the projectile still exists on the field.

Under the "checking status" label, the circuit checks for a projectile colliding with a wall or flying outside the map boundary. The map is taken from the ROM in the circuit itself. The Y coordinate of the projectile is calculated here.

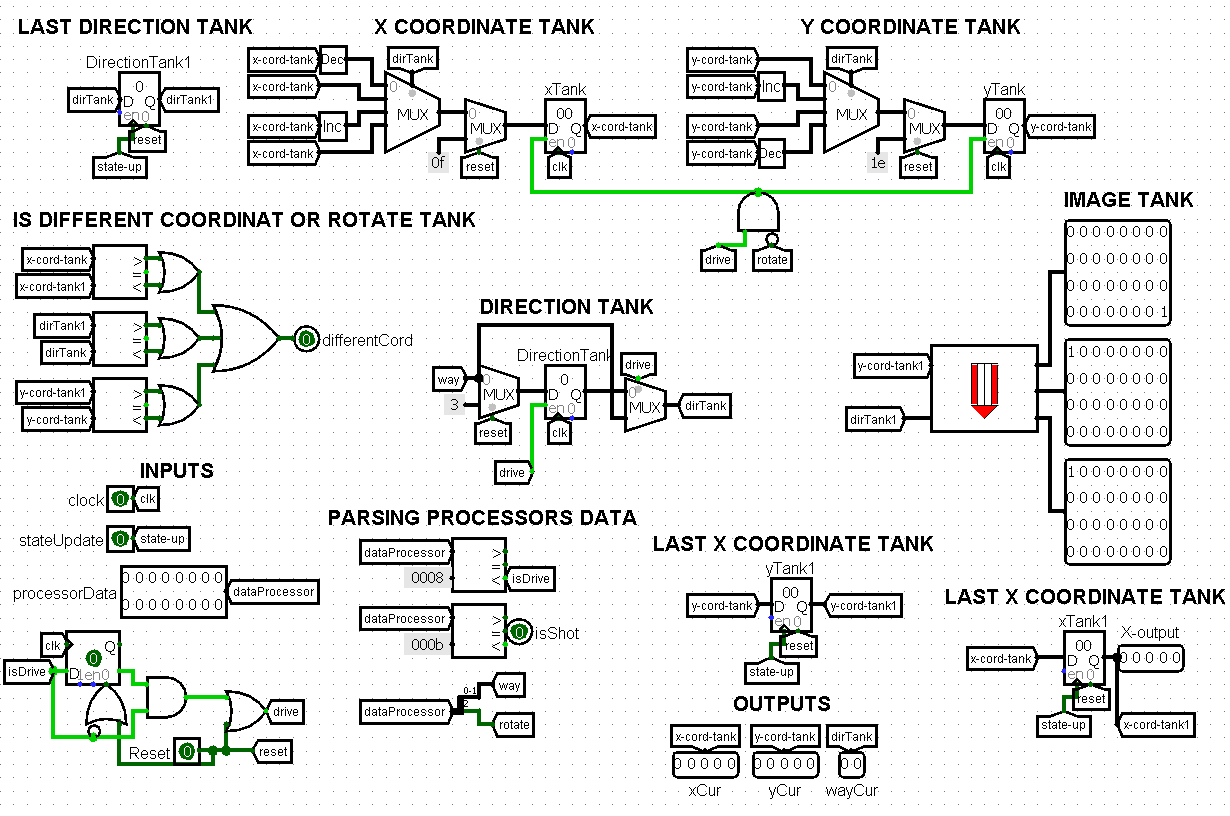
Under the mark "win checking" checks whether the projectile has hit the target or not. For this to happen the projectile needs to be at a distance of no more than one pixel from the centre of the tank. For calculation the auxiliary scheme "minMax" is used to find the maximum and minimum coordinate. When checking for victory, it is always checked that the projectile is active to avoid errors.

Under the labels "x and y coordinate bullets" the bullet coordinates are read and stored. Bullet is launched from the centre of the tank. In the process of implementing this scheme, we encountered the fact that the tank is drawn by the bullet when it is fired. The solution to the problem was to prohibit the projectile to be drawn until it goes beyond the boundaries of the tank. This work is performed by the counter under the label "update limiter".

Under the label "output line" we store the image of the bullet. Under the label "x last coordinate bullet" we store the old value of the X coordinate of the bullet.



## Scheme of the enemy tank



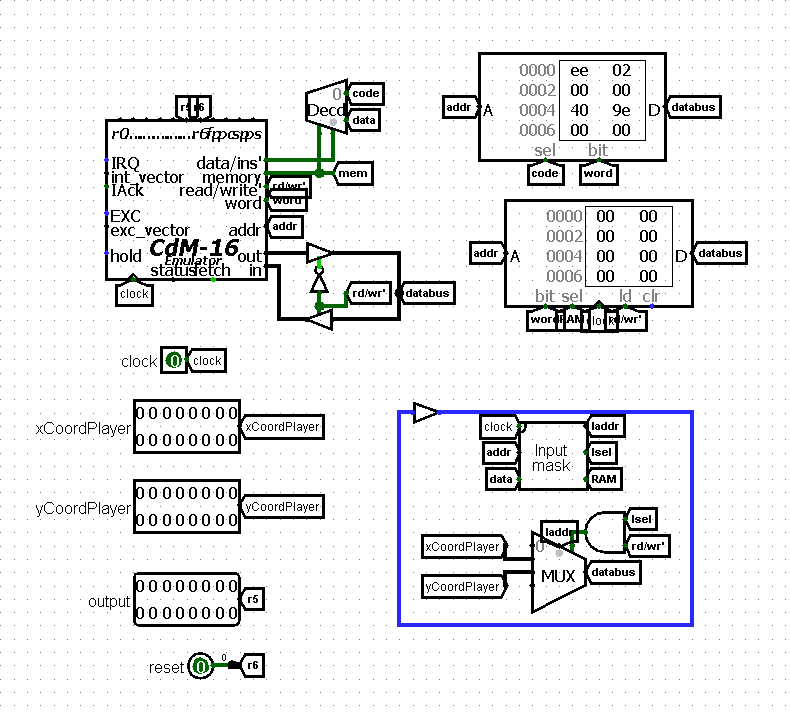
The scheme of the enemy tank is no different, except for the following points. It is fed to the input data from one of the registers of the processor. The circuit processes them in the following way:

* code 0-3 - driving and turning
* code 4-7 - just turn (so that the tank can shoot at the player, being in a narrow space)
* code A - do nothing
* code B - shot

If the code is 0-3, we allow writing to the registers that store X and Y coordinate of the tank. If the code is 4-7, then we can change only the direction of the tank.

During the development process we also encountered the problem that the value on the register is held not one but several clock cycles. In order to ensure that the signal is processed correctly, we built a small circuit containing a trigger that allows changing the position of the tank only for one clock cycle. It is reset when the input data is updated.

## Processor scheme

* 1. 

The enemy control scheme includes a Harvard-architecture processor that implements the bot's movement and firing logic based on the map and the player's position.

While testing the prototype programme, we encountered the problem that the programme to compute the shortest path to a point that uses BFS (graph traversal in width) takes too long to execute.

To eliminate this problem it was decided to use the Floyd-Worshall algorithm, which allows to calculate in advance the shortest paths between all pairs of cells on the map. This allows not to recalculate the route each time, but simply get the path from the path recovery matrix using the current coordinates of the bot and the player.

The next problem we faced in implementing the Floyd-Worshall algorithm was the amount of memory of about 2 MB that the path recovery matrix occupies.

To optimize memory usage it was decided to store in each element of the matrix not the coordinate to which it is necessary to move, but the direction of movement to it, it is encoded by two bits (encoding is similar to the player's tank). It was also decided to store only passable cells in the matrix, which also reduced the volume. To save memory and simplify route recovery, a two-bit number always means travelling in a particular direction rather than turning in place. By implementing this scheme, it was possible to reduce the amount of memory used to 32 KB.

To implement bot firing, it is necessary to accurately determine the moment when the player is in the kill zone. For this purpose, we used arrays of prefix sums calculated for each row and each column. They allow to determine in one operation whether there are walls in the direction of the intended shot.

Since the bot shoots in the direction of its movement, a turn mechanic was implemented: if the player is in a different direction, the bot first turns towards it and then fires a shot. However, movement commands not only resulted in turning, but also in moving the bot in the corresponding direction. Therefore, separate commands were added that are solely responsible for turning without movement.

Depending on the situation, the processor returns:

* Direction code
* Rotation code
* Shot code
* Inactivity code

Input to the processor is realised using a data intercept circuit. Output is realised via the CDM-16 general purpose register.