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): Software Engineering and Computer Science

TERM PAPER

Authors
Barsukova Darya Maksimovna
Bondar Egor Alexandrovich
Fesenko Roman Vladimirovich

Job topic:

"PINBALL GAME"

Novosibirsk, 2023

TABLE OF CONTENTS

Introduction

The use of software in conjunction with hardware is becoming increasingly popular these days. Video games are one example of the use of the technologies mentioned above. There are a large number of genres and trends. In our project we are dealing with arcade games. This is a genre of computer games, implying a short-time intensive gameplay. It is important to remember that the arcade is often combined with other types. Representatives of the genre are flying airplanes, Tetris, pinball and more.

In the classic Pinball game, the player needs to gain game points by manipulating one or more metal balls on the playing field covered with glass using flippers. The main goal of the game is to score as many game points as possible. The second most important goal is to maximize the duration of the game by using extra balls and keeping the ball on the playing field as long as possible to get a free game.

The aim of this project is to design and create a Pinball game on digital logic circuits using the CdM-8 processor and its assembly language.

In accordance with the purpose, it is necessary to solve the following tasks:

1. Explore analogues of the Pinball game;
2. Study and analyze information about the processor, its capabilities, commands and instructions;
3. Develop the hardware for the operation of the game;
4. Define functional requirements;
5. Develop the software part and implement it into the digital logic circuits.

1. Problem statement

1.1. Concept

The Pinball game is a field on which walls and other various obstacles are placed. A ball is launched onto the existing field, which actively bounces off the walls under the influence of gravity. The player's task is to control the doors at the bottom of the field, and keep the ball on the field as long as possible. The player is awarded points for each hit of the ball against the wall.

There are only 4 types of walls in our implementation of the game: horizontal, vertical and 2 types of diagonal at an angle of 45 degrees. The doors controlled by the player do not move, but turn on and off. They are controlled by 2 separate buttons, so only one door can be closed at a time.

1.2. Functional requirements

The purpose of this project is to develop and create a Pinball game based on an electrical circuit with a CdM-8 processor included in it.

Below are the functional requirements:

1. Controlled door position (on/off);
2. Uncontrolled movement of the ball under the influence of gravity;
3. The ball appears at the top of the field;
4. The ball cannot cross the wall;
5. The ball can move in all directions;
6. The score counter should increase by one when the ball hits the wall;
7. The score counter should reset to zero when the ball is reloaded.

2. Analogues

During the creation of the project, we explored games similar to Pinball. Let's consider some of them:

1. A British inventor named Montague Redgrave created a manufactory for the production of bagatelle tables in 1869. The inventor received the American patent "Improvements In Bagatelle" in 1871: the holes to get into were replaced with springs, and the cue at the end of the table was replaced with a plunger. The player launched the ball onto the tilted playing field using a plunger, which became an integral part of all pinball machines. This innovation has made the game more convenient. The size of the table has become smaller, and a score counter is installed on the top panel. Innovations are recognized as the official birth of pinball.

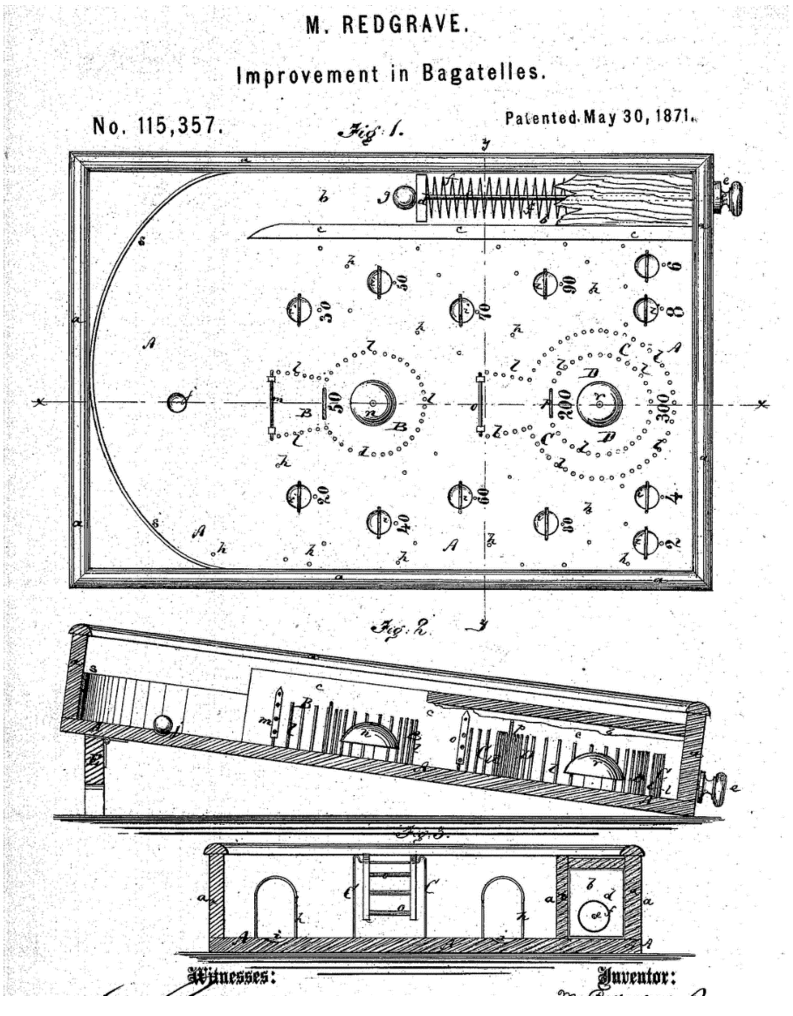


Figure 1 – mechanics of a pinball machine

2. Full Tilt! Pinball is a computer game developed by Cinematronics and published by Maxis Software in 1995, a simulator of the pinball board game for the personal

computer platform. During the game, the player controls the left and right flippers, with which it is necessary to hit the ball, hitting various targets. At the beginning of the game, three balls are given, as the player progresses, he must accumulate balls. There are a total of 9 levels in the game, which correspond to the ranks, increasing which the player gets access to more complex missions.

In addition, the game has the options "2 people", "3 people" and "4 people" — by choosing one of these options, you can compete with friends who will score more points. The game ends when the player loses all the balls. The maximum number of points that can be scored in this game is 999999999.

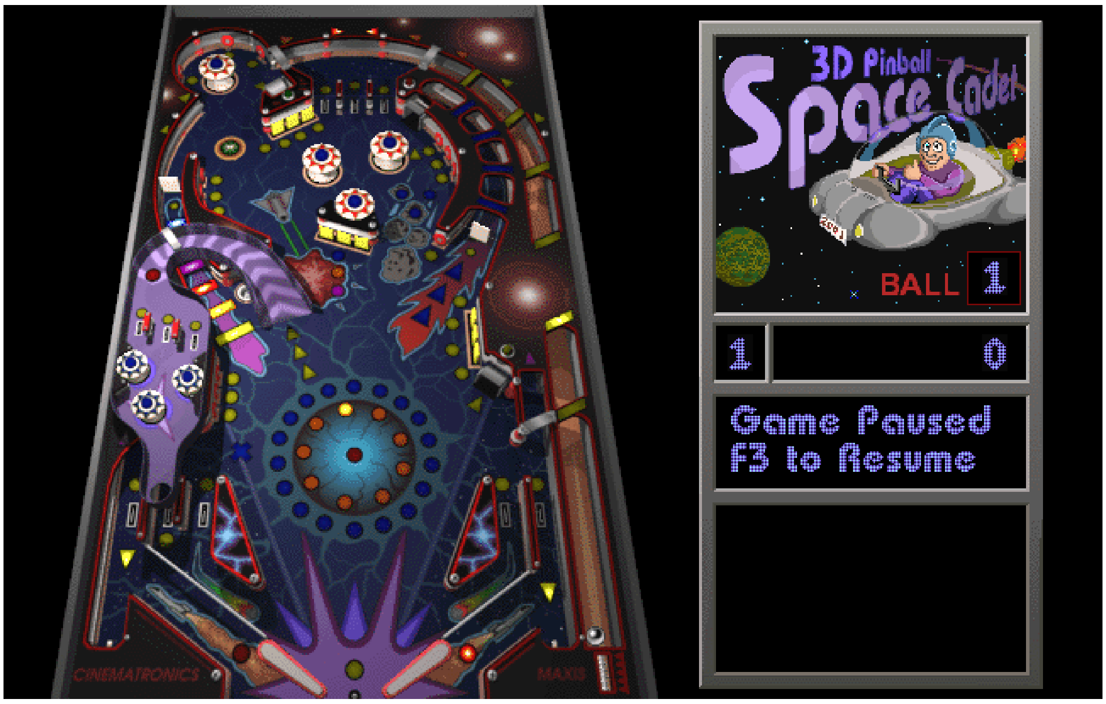


Figure 2 – Pinball game interface

After studying and analyzing these examples, our team decided to create a project based on the design and content of the Pinball game. Due to limited resources, a not very complicated option was required, which could be implemented using CdM-8 and our knowledge in the field of circuit design.

1. Hardware

The hardware part of our project consists of digital logic circuits created in the Logisim program, in which we modeled and edited them using an accessible and intuitive graphical interface. Let's look at our developments.

This part of our game is divided into the following blocks: The `Ball` chip is responsible for all the movements of the ball and its physics and the `Graphics Main` chip is for the graphics part.

We also use a small but important chip to calculate the player's current points and save the maximum result.

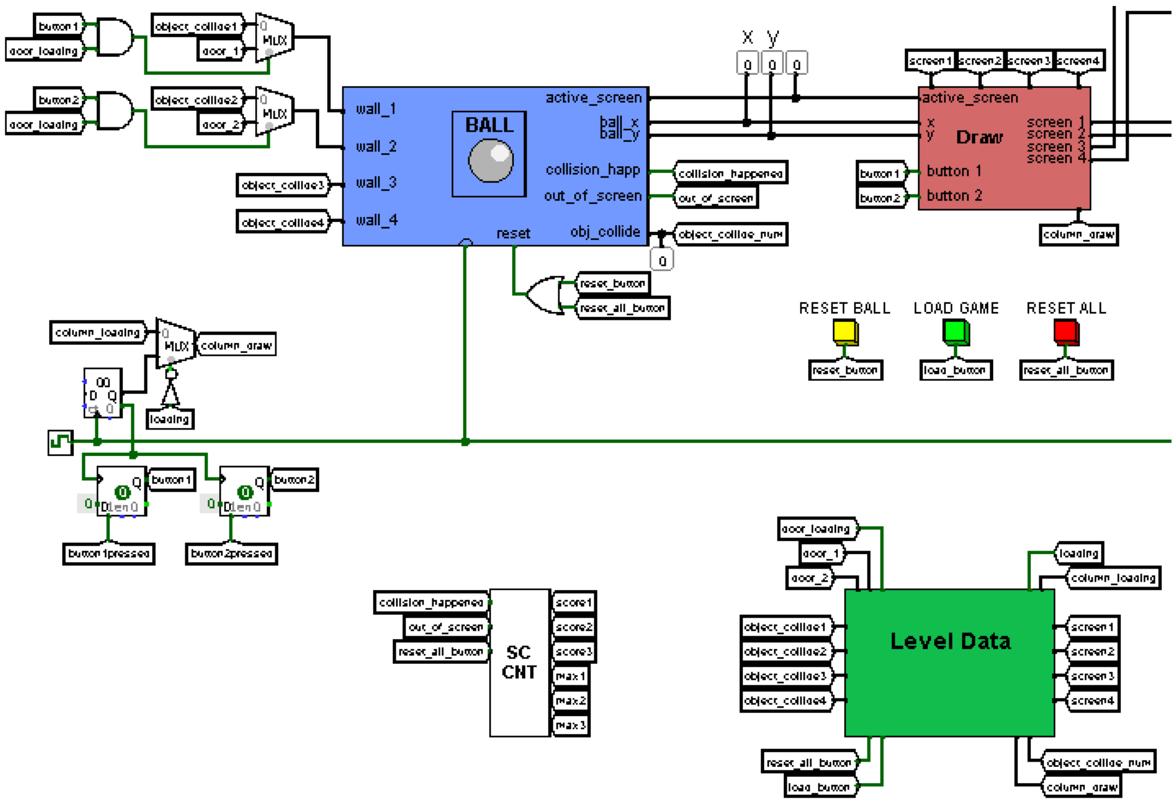


Figure 3 – the screenshot of the Main circuit

1.1. Main interface

Consider the interface of the game. The `graphic man` circuit is connected to the matrix and is controlled as follows: by the `reset ball`, `load game`, `reset all`, `left`, `right` buttons. We used a 64*64 pixel extended screen to make the game look beautiful. We have assembled our screen from 4 standard ones. `Memory` blocks are presented to each of the screens for optimal output to all 128 columns. The blocks update only 1 of the rows each clock cycle, leaving the remaining rows in the previous saved state. Thus, the full rendering of the screen takes 32 cycles. Figure 4 shows a screenshot of the main interface circuit.

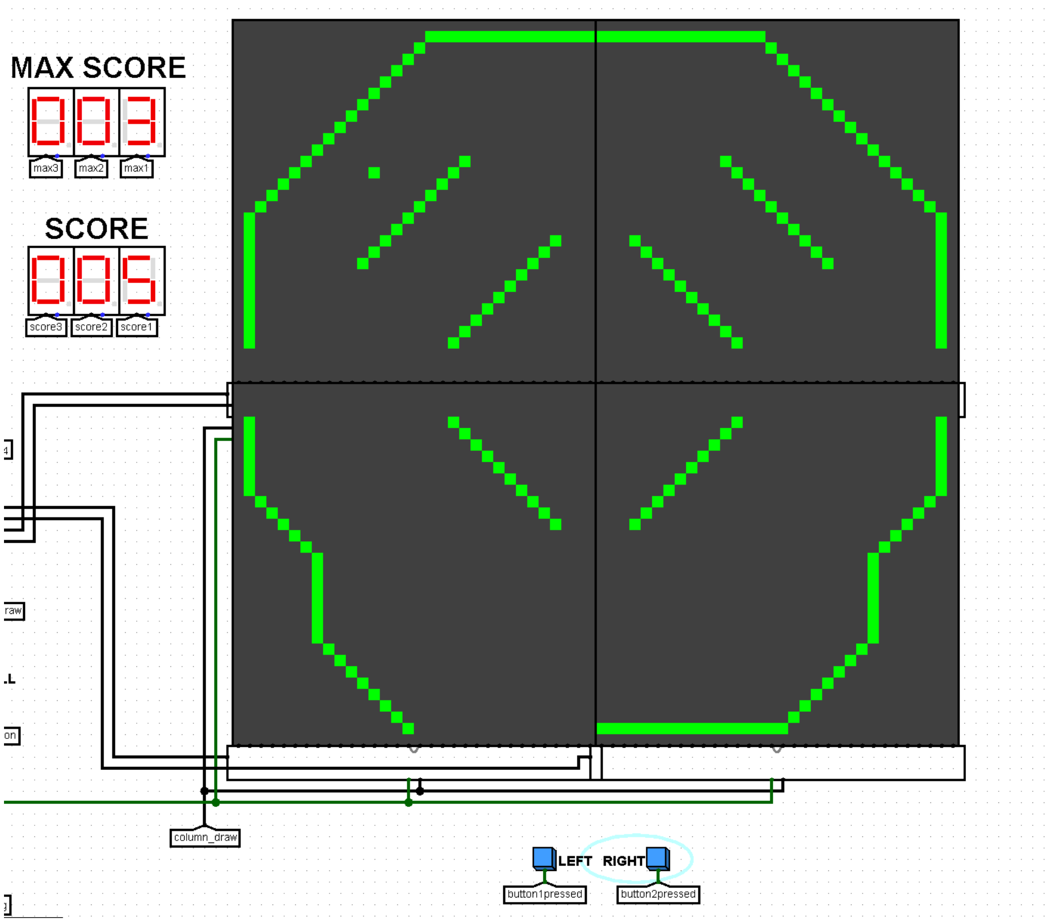


Figure 4 – game interface

1.2. Graphic part

Initially, we made a block that converts 1 wall into a pixel image and overlays the resulting images on top of each other. This algorithm draws 1 wall in 32 clock cycles. We analyzed that it is too resource-intensive to simultaneously support 128 32-bit numbers changing every frame. This noticeably slows down the clock speed of the game due to the complexity of calculations and enormously increases the size of the circuit. Therefore, due to the fact that all the walls in the game do not move, we can run the algorithm 1 time, remember the resulting state of the screens, and then load it from RAM.

The graphical part of our project consists of 2 main parts: a loader, a handler and a storage device. We will consider the device and the principle of operation of the `level loader` in the description of the software part, and now let's look at the last two parts in more detail.

1.2.1. The handler

The `Draw` block accepts several parameters as input: the ball, the state of the doors (on/off), the number of the column to draw and one 32-bit number for each screen. This number `base` is responsible for how the walls look in the current column of the screen. It is calculated in advance using the `level loader` block, which will be described in the following parts. The principle of operation of the `Draw` block is that it iterates over the columns and, if necessary, draws the ball and the doors to the resulting `base`. After that, it transmits the result to the screen. It is worth noting that this part works simultaneously with 4 screens.

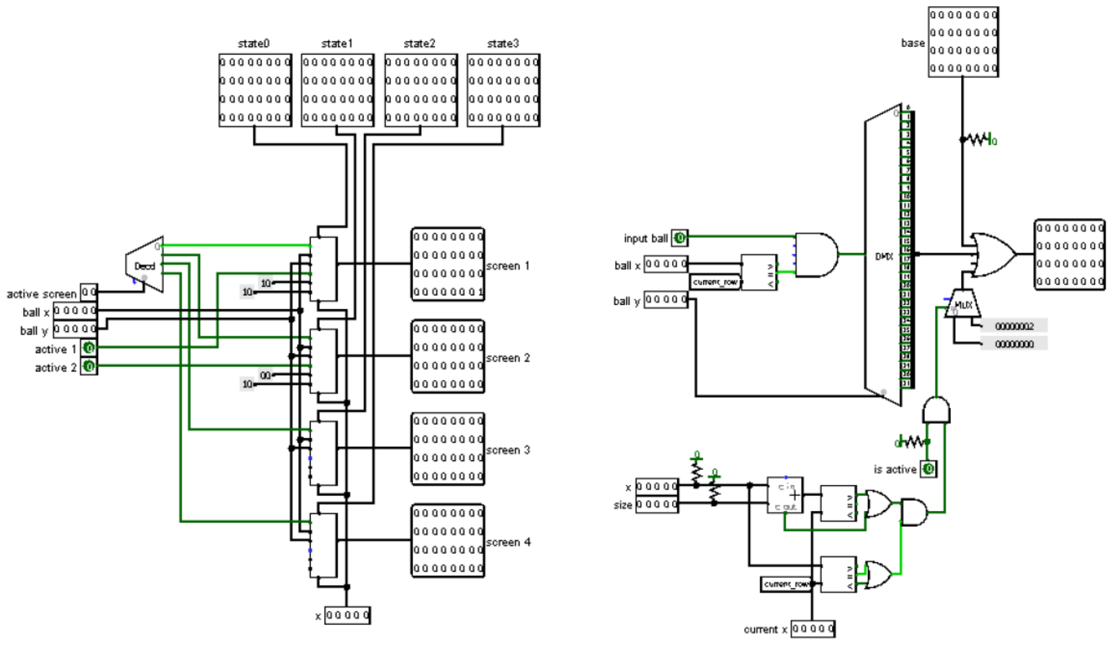


Figure 5 – Draw block

1.2.2. The storage device

`Screen Memory` block stores thirty-two 32-bit numbers, each of which represents columns for each screen. In other words, it consists of 32 registers and accepts the updated column and its number as input. This part works exactly like RAM, however it has 32 simultaneously active data outputs to maintain the image on the screens. Thus, we update the columns of the screens in turn, while keeping the old values in this block.

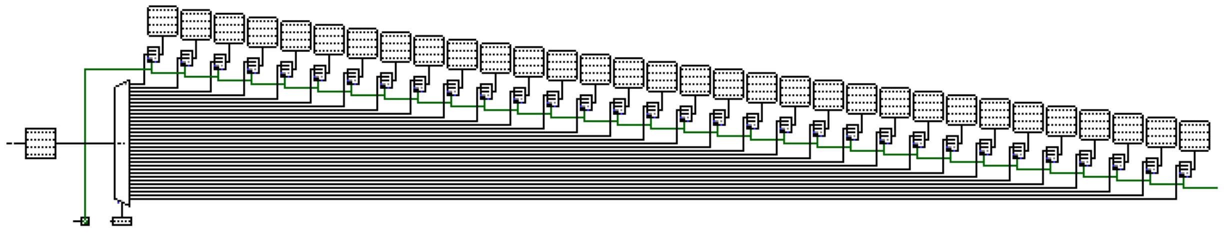


Figure 6 – Screen Memory block

Initially, all walls are stored in 4 ROMs, each of which can have up to 8 walls. The doors are stored in 2 separate constants. Each wall is 26 bits.

Presentation format: `TT SSSSSSSS YYYYYYYY XXXXXXXX`, where the first 2 bits are for type, the next 8 bits are for size and another 8 bits are for Y-axis and X-axis coordinates. Moreover, the coordinates are always calculated from the upper left corner of the wall.

The length of the wall rotated by 45 degrees seems to the user $\sqrt{2}$ times longer, because the program stores exactly the number of its pixels.

In fact, all the coordinates and lengths of the walls are no more than 5 bits, because they exist inside the screens, but for convenience we store them this way.

The coordinates of the ball are presented in 8-bit format: `SXXXXXFF`, where `S` is a screen bit, `F` is the floating part, which we need for greater accuracy of the ball movement.

The speed of the ball is also presented in 8-bit format: `XXXXFFFF`. The speed has a floating part of 5 bits.

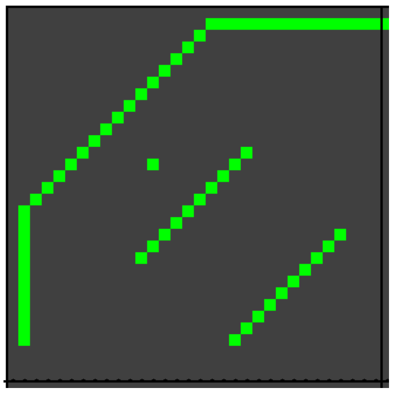


Figure 7 – illustration of all types of walls

We often need to add two 2’s complement numbers, and it is quite simple, but in our calculations there is a floating part that makes everything much more difficult. We found a solution in creating a scheme as in Figure 8. We invert the whole part of the number to get the inverse number, and then we do not add the numbers, but subtract the inverted one from the first number (in this case, `y_b` is always positive, so we only need to work with `vel_y`).

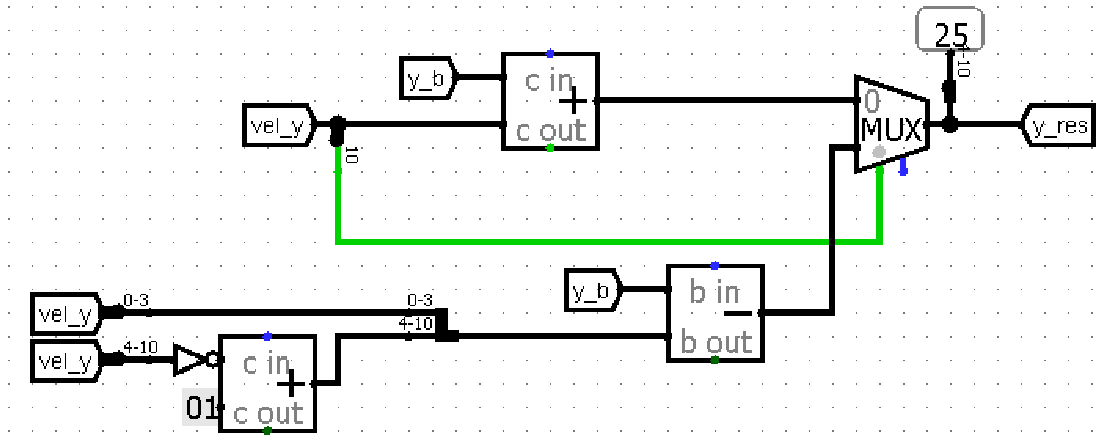


Figure 8 – converting the floating part

1.3. Ball Logic

The main logic of the game is implemented in the `Ball` chip. This is a complete sequential logic circuit without any code, since CdM-8 is too slow for fast recalculations of coordinates, velocities and collisions of the ball. We have achieved a real-time gameplay using pure logic.

Firstly, we will overview combinational logic, then the details of the `Collision` chips and the chips that are responsible for moving the ball. Secondly, we will describe how we use sequential logic to put everything into action.

1.3.1. Combinational Logic

The combinational logic of the `Ball` chip takes as input data:

- wall from the currently active screen
- `ball_x`, `ball_y`
- `speed_x`, `speed_y`

The combinational logic of the `Ball` chip returns 4 signals as output data:

- `new_ball_data` and `move_ball_data` data buses
- `collision_happened` signal
- `out_of_screen` signal

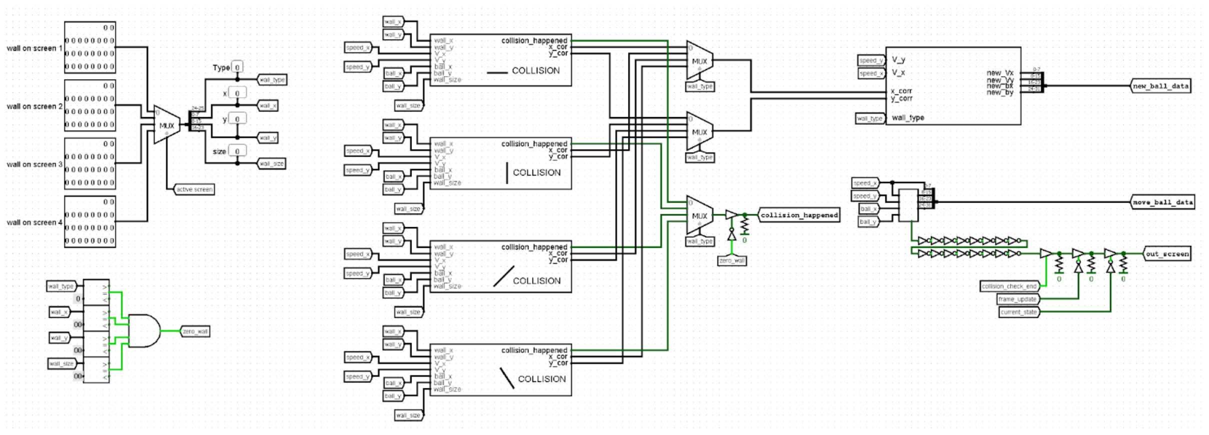


Figure 9 – the general view of combinational logic

The first part takes the 4 nth walls from 4 screens and selects the correct one using a MUX with `active_screen` as the selected signal, and then splits the wall into `wall_type`, `wall_x`, `wall_y` and `wall_size`. It also checks whether the wall consists of all zeros. If this is the case, then the wall simply does not exist. Then the `zero_wall` signal is generated.

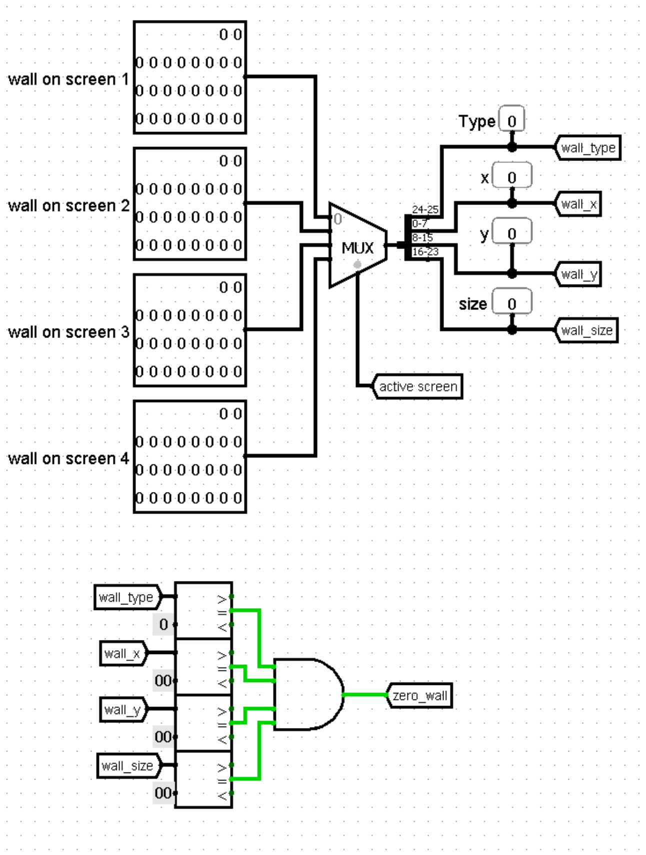


Figure 10 – the first part of combinatorial logic

The second part consists of 4 `Collision` blocks, each for one type of wall. Each `Collision` block checks the collision of the ball with the wall and outputs a collision_happened signal, x_corr and y_corr, which are the corrected coordinates of the ball after the collision.

If the ball is far enough away and at the same time has a high speed when approaching the wall, then we cannot immediately change its speed. In this case, it is also necessary to adjust the coordinates so as not to give the impression that the ball pushed off the wall before reaching it.

We will discuss Collision block in detail later.

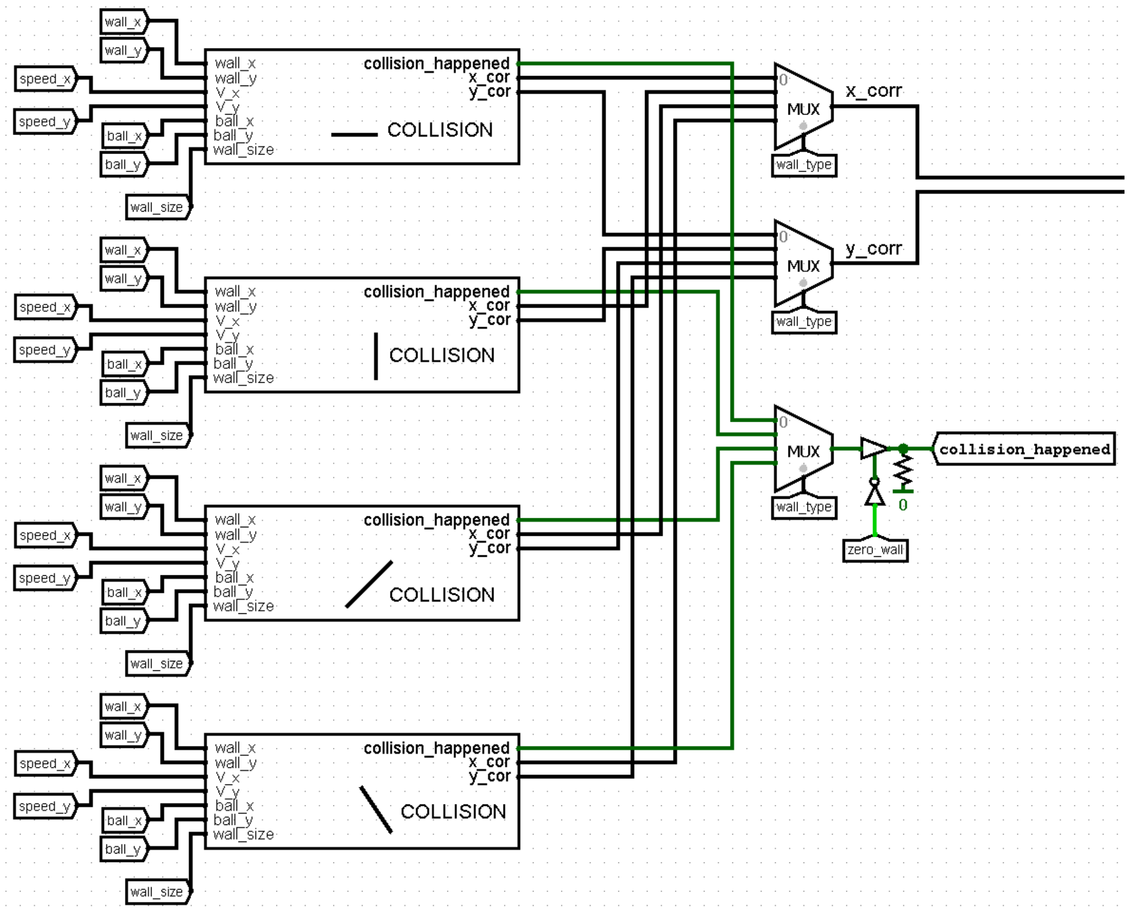


Figure 11 – Collision block

The third part consists of two chips.

The upper chip takes the current speed of the ball, the adjusted coordinates and the type of wall as input. It outputs the new speed and coordinates, and then collects them in the new_ball_data bus (it will be used and loaded into the ball register if a collision occurs).

The lower part takes the current speed and coordinates of the ball as input and outputs the new speed, coordinates and out_screen signal, which informs that the ball moved off the screen. For this purpose, we have created a Move chip that basically just moves the ball using its velocity and updates the Y-axis velocity by simulating gravity. It also collects all the data about the ball in the move_ball_data bus (it will be loaded into the register if there is no collision).

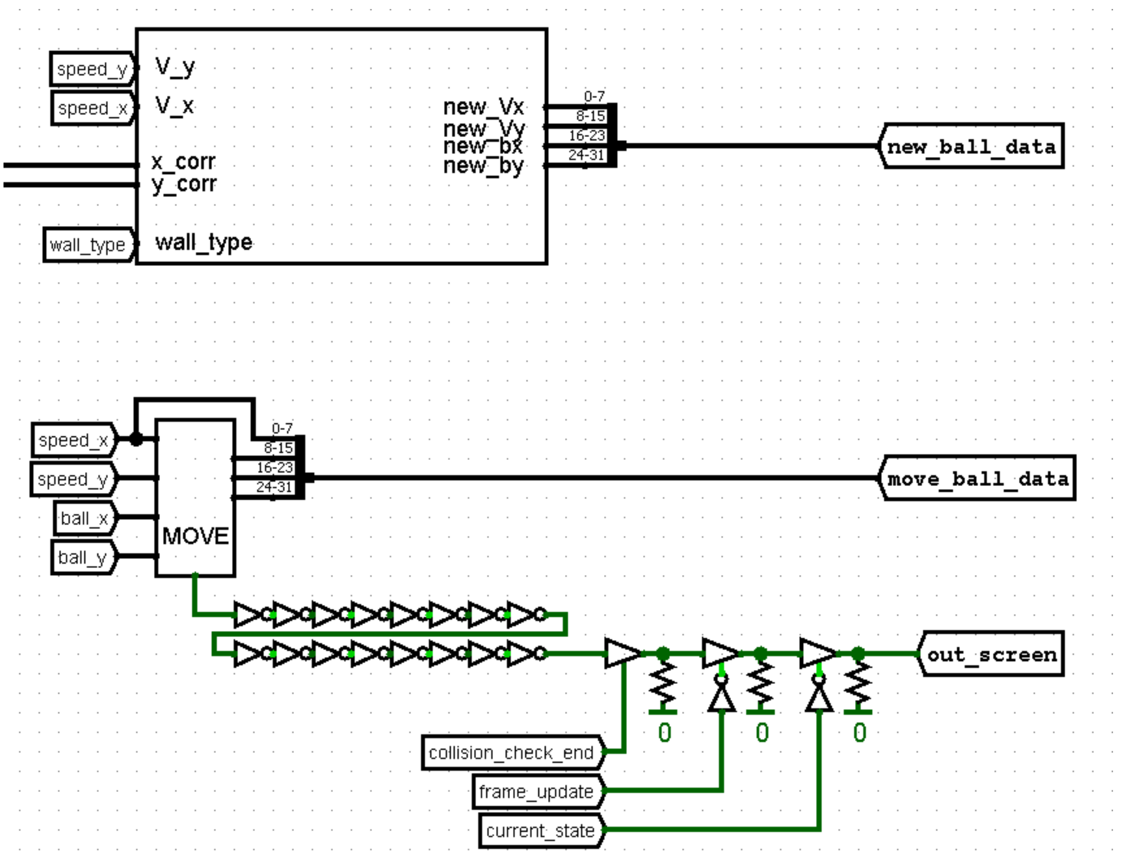


Figure 12 – the third part of combinatorial logic

`Update_ball_param` chip just updates the speed of the ball depending on the type of wall it collides with.

If wall type is 0 (_) → y = -y

If wall type is 1 (|) → x = -x

If wall type is 2 (/) → x = y and y = x

If wall type is 3 (\) → x = -y and y = -x

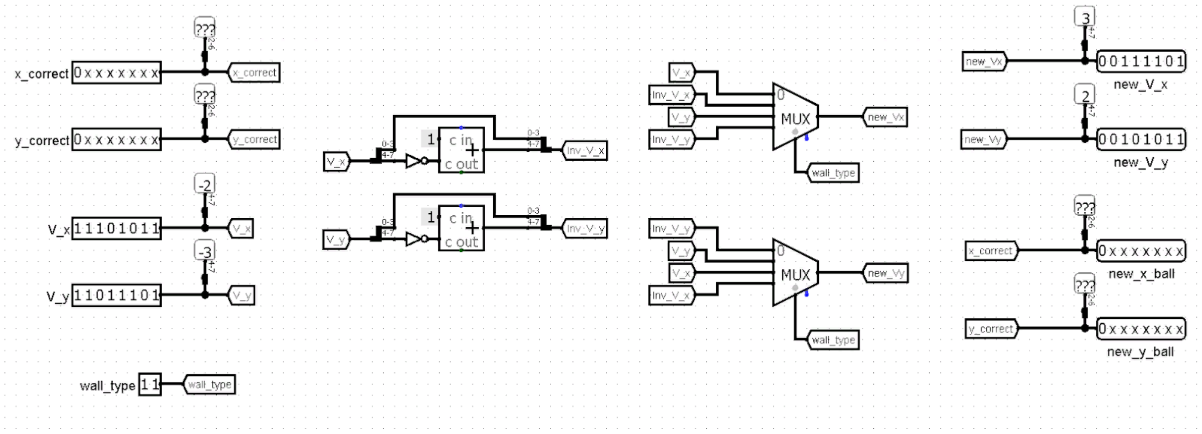


Figure 13 – update_ball_param chip

The input block of the `Move` chip takes all ball data buses and extends them to an 11-bit format (`SXXXXXXFFFF`).

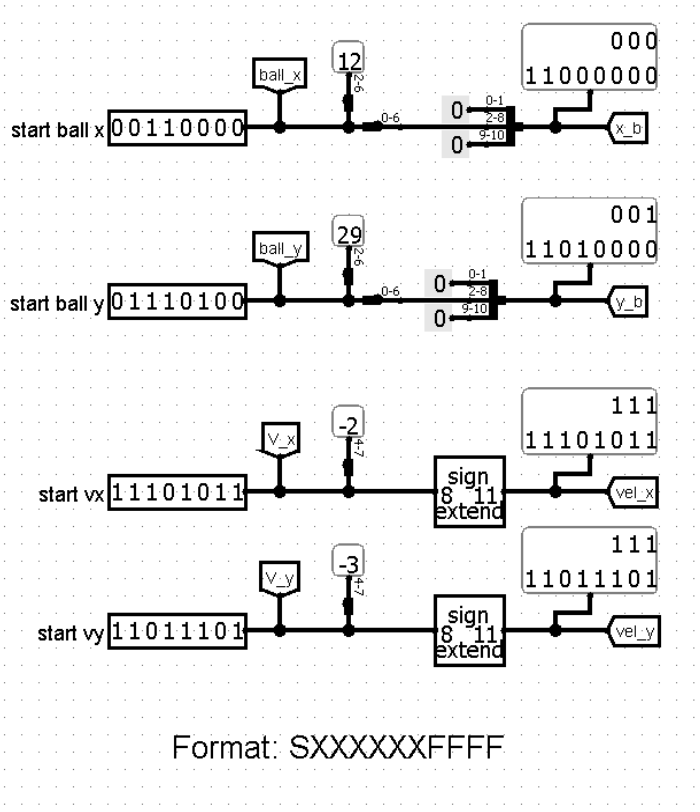


Figure 14 – the input block of the Move chip

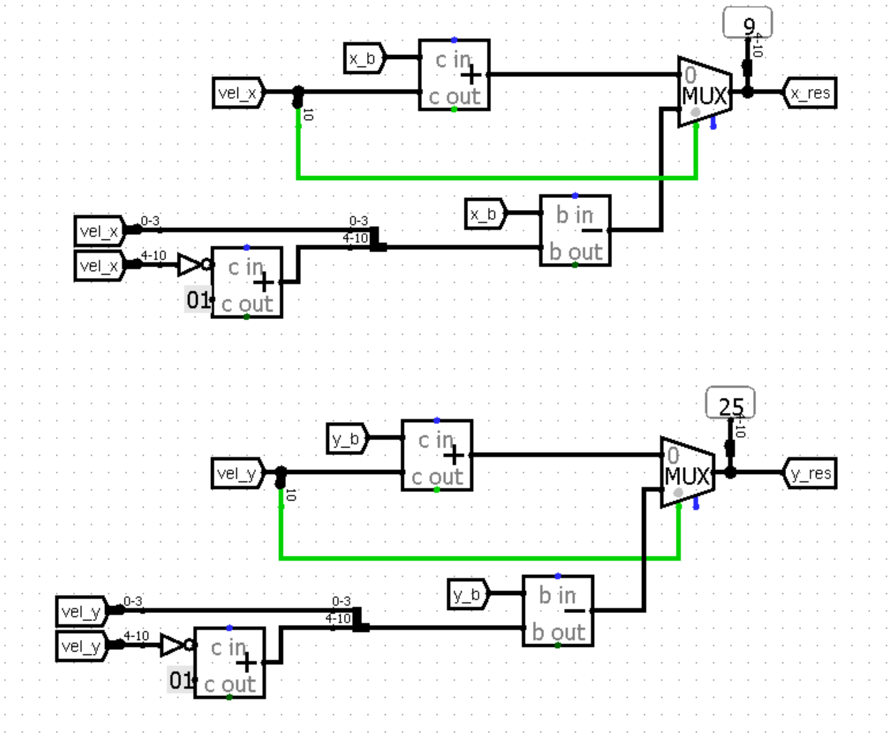


Figure 15 – calculating the resulting coordinates of the ball

This logic calculates the resulting `V_y` value by subtracting a little gravitational constant from it.

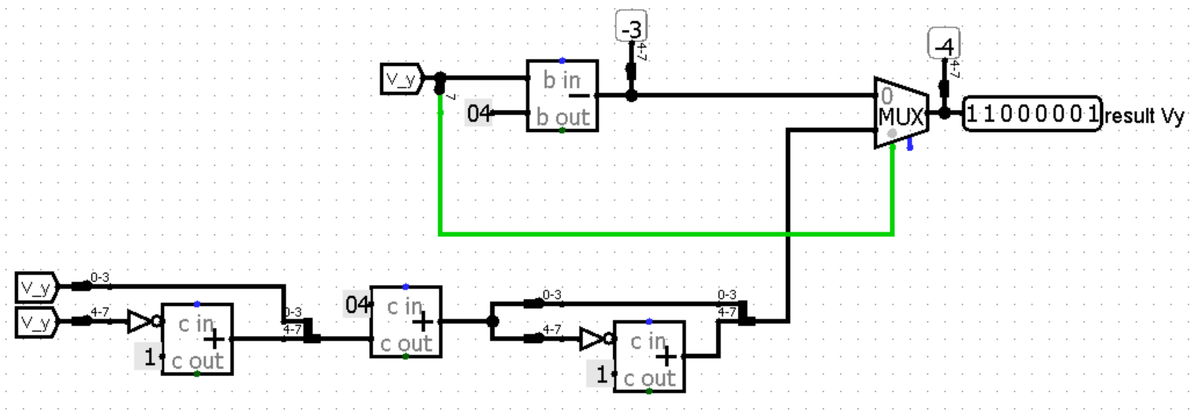


Figure 16 – calculating the resulting V_y value

This is the main part in Figure 17 that outputs the resulting coordinates (for X and another similar part for Y). It uses a fairly simple logic to determine whether the ball flies from one of the 4 screens to another or it flies out the screen.

If the screen bit of `ball_x` is 0, then it is on screen 0 and can fly to screen 1 in the positive direction or fly out of the screen in the opposite direction.

So, if the ball flies off the screen, then we will not calculate its coordinates. However, if it flies to screen 1, then we need to execute (`x_res` – 32) to get the correct coordinates on screen 1.

We know that it flies out of its starting screen (out or to 1) by signal: (`x_res` < 0 OR `x_res` >= 32).

If the ball remains on the same screen and has changed the coordinate value, then we send the signal as selected to the second MUX on the right side, which selects only from the `x_res` value.

If the screen bit `ball_x` is 1 and the ball goes to screen 0, then we change its coordinate: 32 - (inv(`x_res`)FFFF). We can't just add 32 to `x_res` because of the floating part of the coordinate.

The lower part checks whether the ball flies out of the game screen. These are 2 AND gates:

(`x_res` >= 32 AND screen bit `ball_x` = 1 AND `vel_x` > 0)

(`x_res` < 0 AND screen bit `ball_x` = 0 AND `vel_x` < 0)

The first gate determines whether the ball flies out from screen 1 to the right. The second gate determines whether the ball flies from the screen 0 to the left. We have the same logic for the Y coordinates.

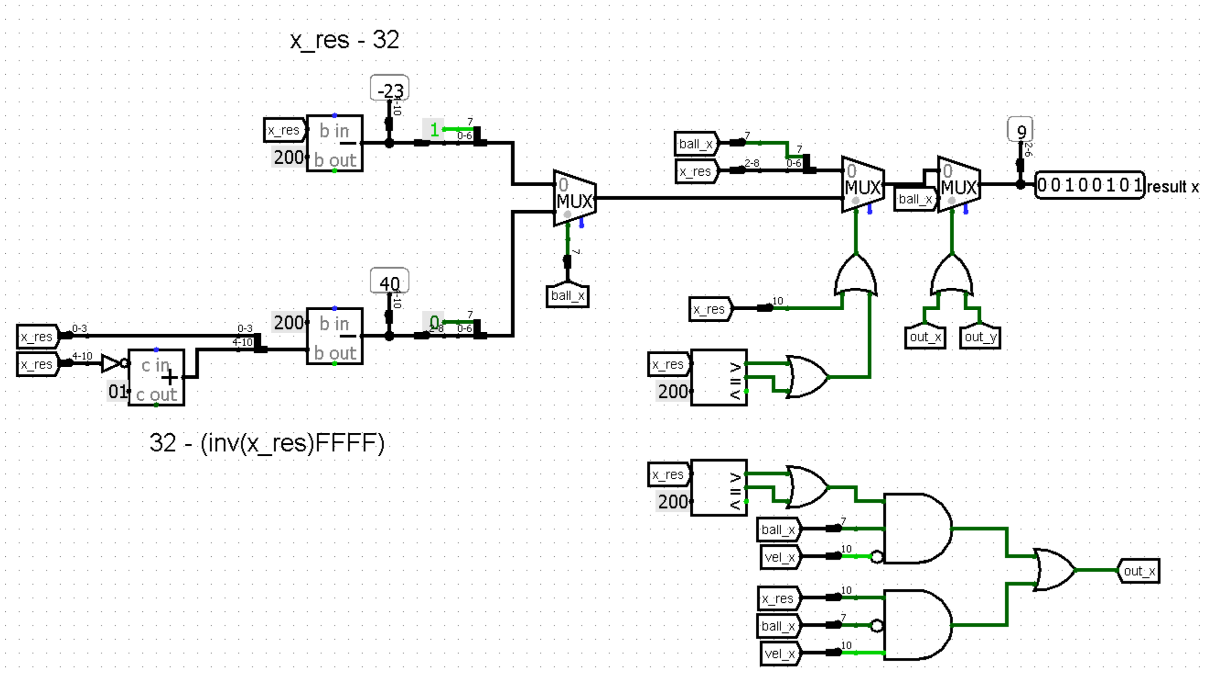
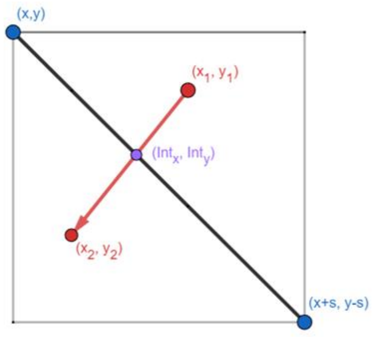


Figure 17 – the main part of combinatorial logic

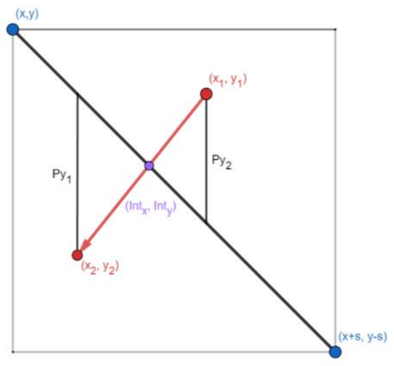
1.3.2. Collision Block

Let's look at the general principle that we use to determine a collision:

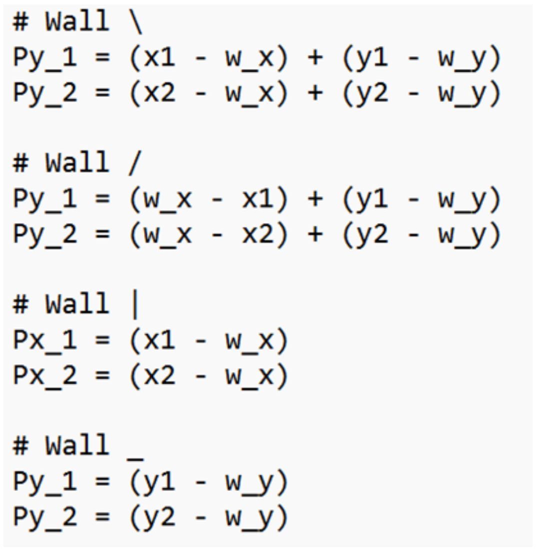
1. We calculate the intersection point of the wall and the straight line given by 2 points: the starting and ending position of the ball. For this purpose, we use 4 `Intersection` chips, each for 1 type of walls. This will be described in detail later. Also, the `Intersection` chip generates an `intersection_impossible` signal, which means that division by zero occurs.
2. If this point fits into a square (our wall is a diagonal of this square), a collision is possible.



3. Then we calculate the vertical distance from the start and end points to the wall: `Py_1` and `Py_2`.



We use fairly simple geometric formulas for these calculations:



```
# Wall \
Py_1 = (x1 - w_x) + (y1 - w_y)
Py_2 = (x2 - w_x) + (y2 - w_y)

# Wall /
Py_1 = (w_x - x1) + (y1 - w_y)
Py_2 = (w_x - x2) + (y2 - w_y)

# Wall |
Px_1 = (x1 - w_x)
Px_2 = (x2 - w_x)

# Wall _
Py_1 = (y1 - w_y)
Py_2 = (y2 - w_y)
```

Figure 18 – intersection_point_border_condition signal generation

4. If `Py_1` and `Py_2` have different signs, then the ball will pass through the wall. This option is also possible if one of the values is zero and other is not. It generates a `wall_crossing` signal.
5. Then we combine 3 signals to get the signal that will determine the collision.

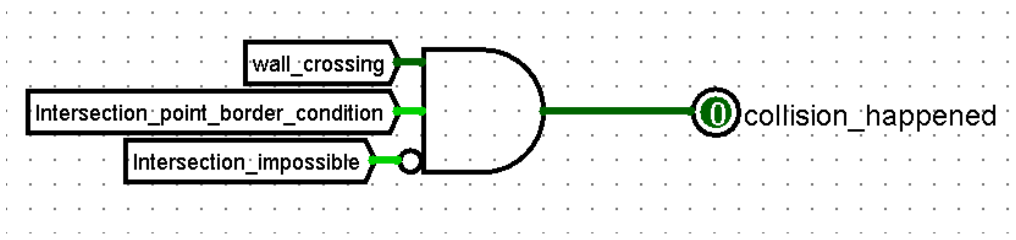


Figure 19 – combining 3 signals

The `Collision` chip is also responsible for the output of corrected coordinates. So, we take the coordinates of the intersection point and slightly change them depending on the type of wall and the direction from which the ball came. We need to do it because we use a pixel screen, and the intersection point can be on the other side of the wall. Let's pay attention to Figure 20. We need to change the coordinate int point +1 on each axis. After modification, we check whether this point fits into our screen (0 <= x/y and < 32). If the point does not satisfy the condition, then we output only the standard coordinates of the intersection point.

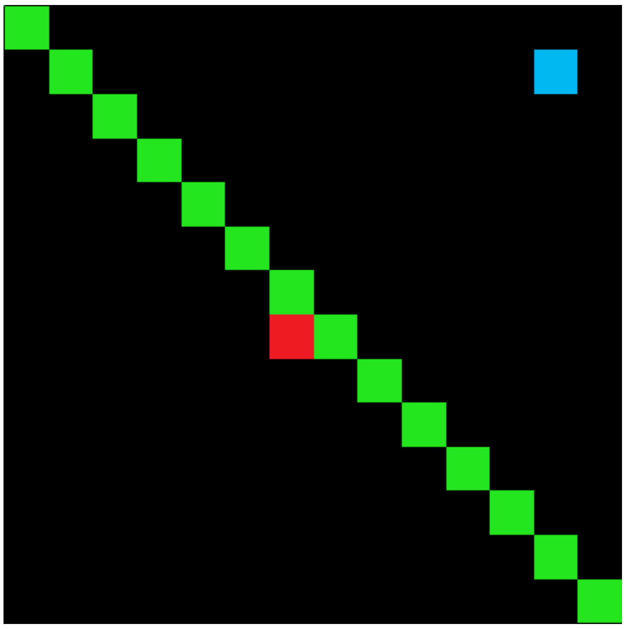


Figure 20 – blue – starting coordinate of the ball, red – intersection point

Now we will look at the `Collision` algorithm using the example of the 4th block, which checks for a collision with the 3rd type of wall (\):

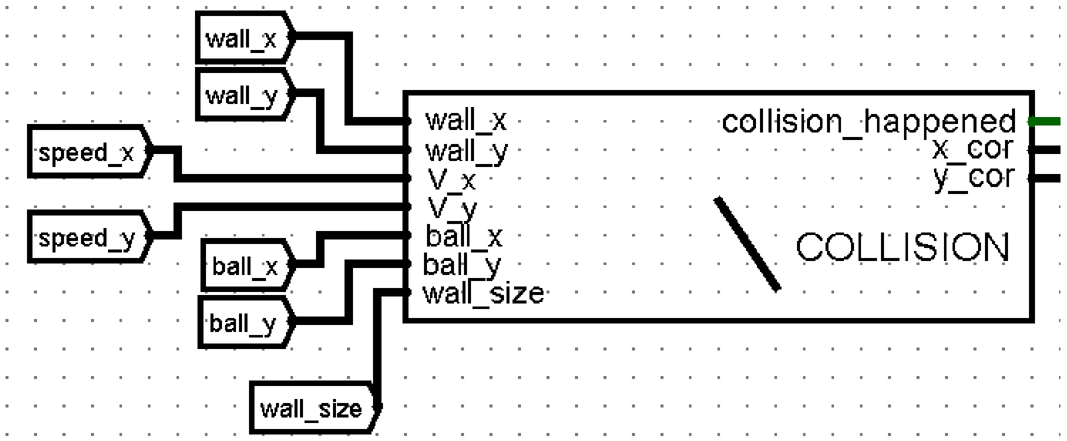


Figure 21 – a collision with the 3rd type of wall (\)

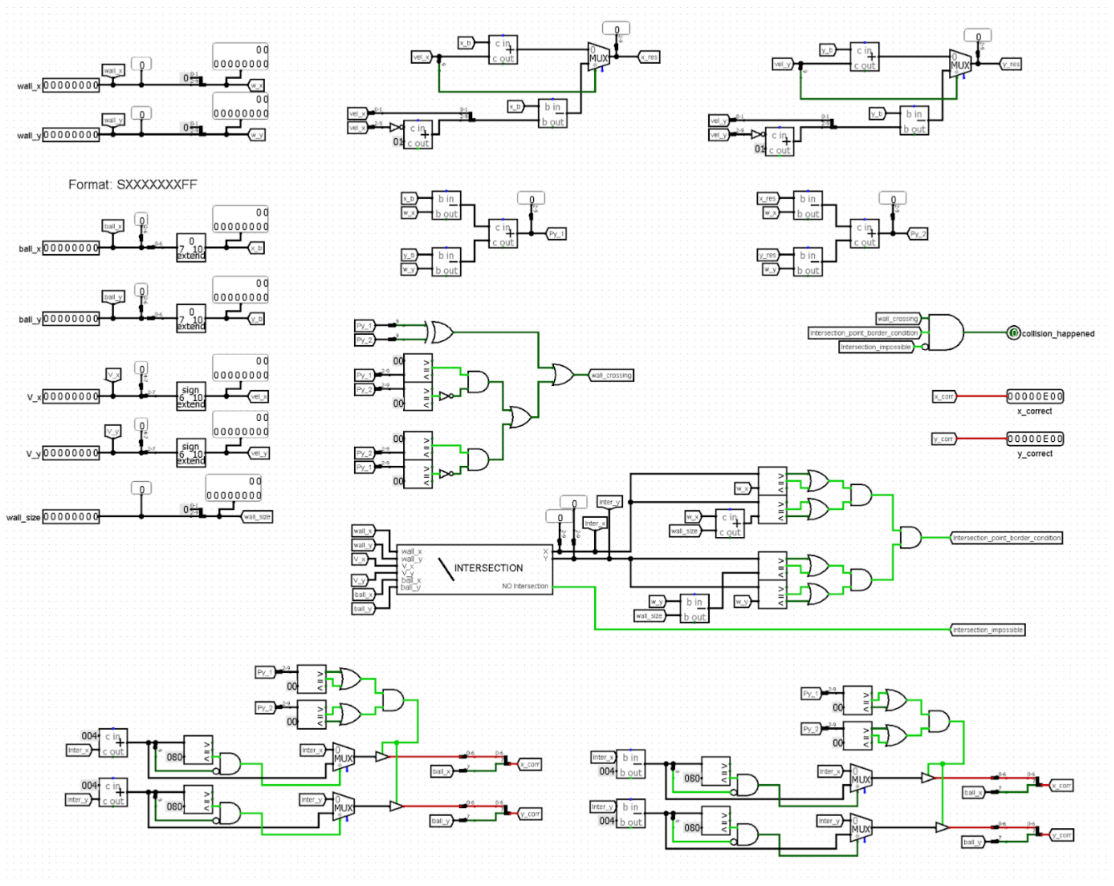


Figure 22 – image of the entire logic of the Collision chip (\)

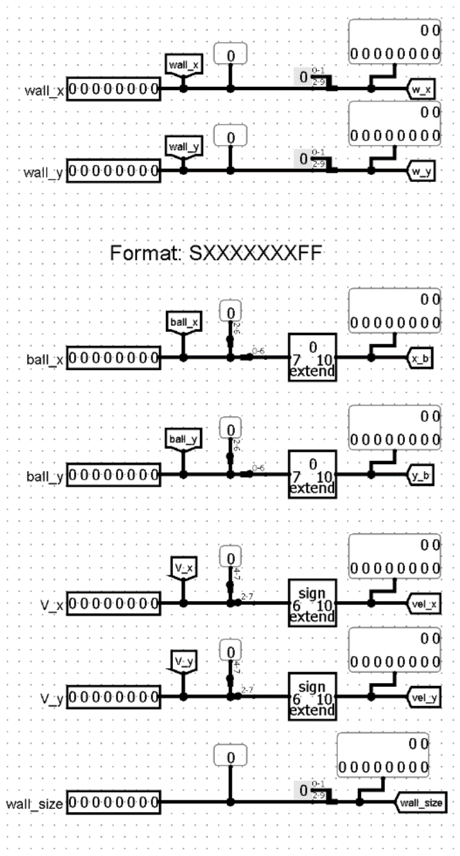


Figure 23 – the input block

We accept input signals and extend them to the format: `SXXXXXXXFF`, where `SXXXXXXX` – 2's complement 8-bit number, `FF` is the floating part.

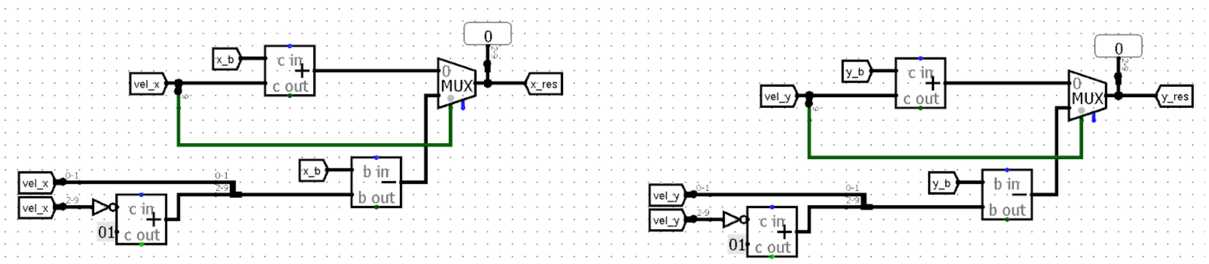


Figure 24 – calculating the x_res and y_res coordinates

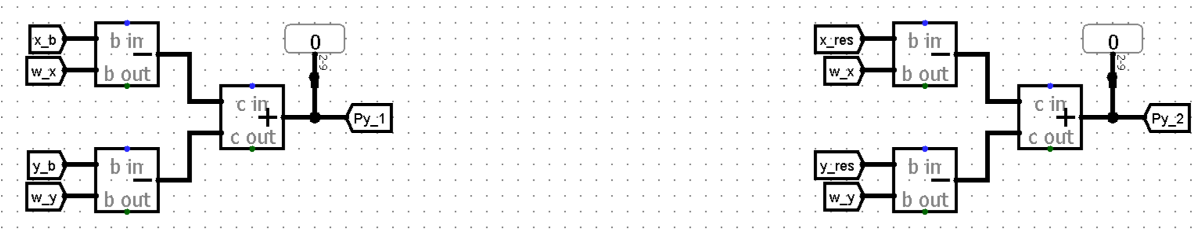


Figure 25 – calculating Py_1 and Py_2

The block in Figure 26 checks the condition of crossing the wall defined earlier.

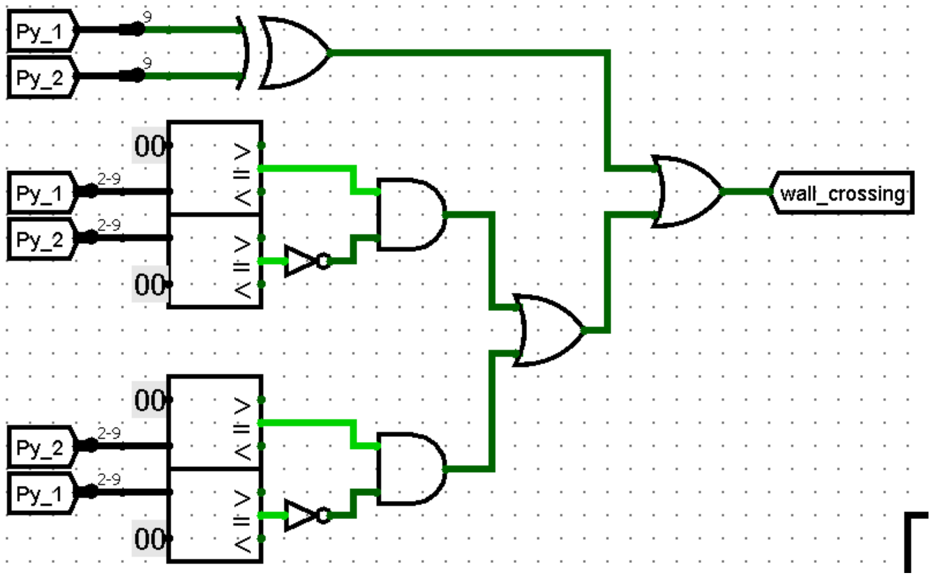


Figure 26 – checking the intersection

The next part uses the `Intersection` chip to calculate the intersection point and then checks if it fits into the square. For this case, these are:

(`inter_x` >= `w_x` and `inter_x` <= `w_x` + `wall_size`)

(`inter_y` >= `w_y` – `wall_size` and `inter_y` <= `w_y`)

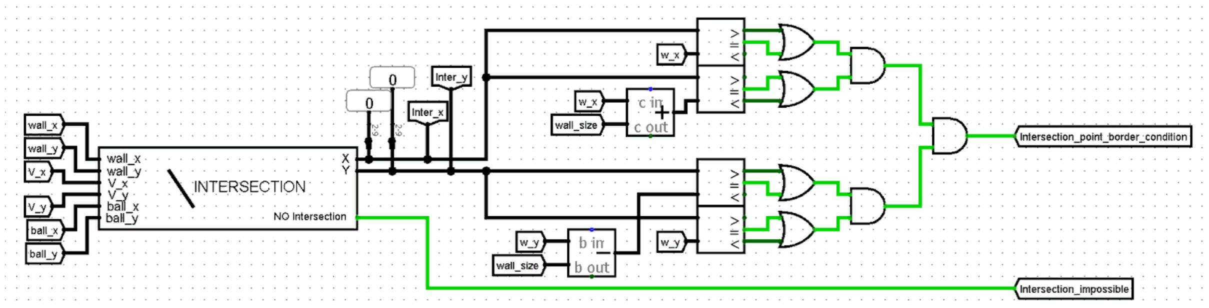


Figure 27 – verification of condition fulfillment

The next two blocks change the coordinates of the intersection points to get the corrected coordinates, and also check whether they correspond to the screen. Two comparators in the upper part determine which side the ball came from and transmit the correct coordinates.

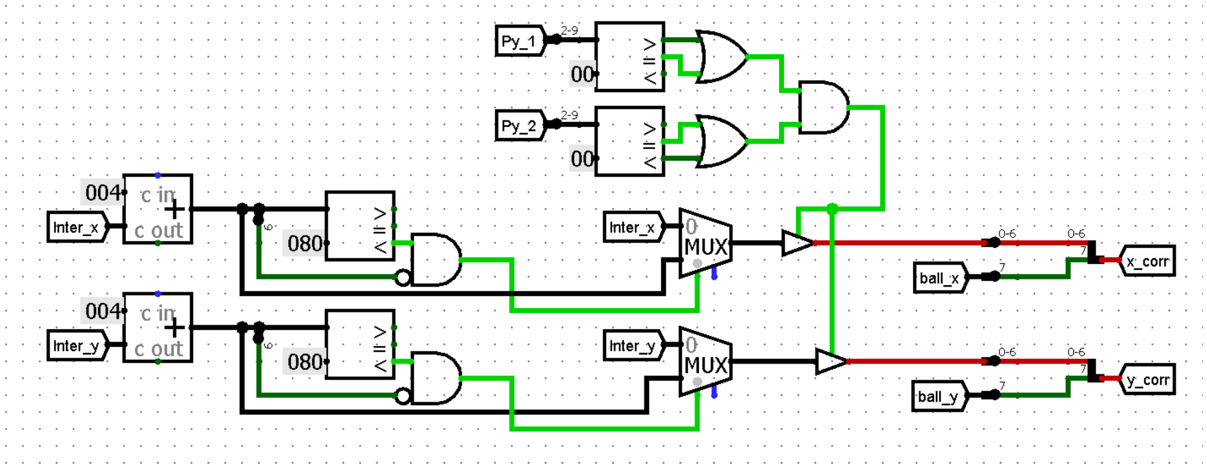


Figure 28 – changing coordinates

1.3.3. Intersection Block

We use the following formulas to determine the intersection point:

Wall (\): $C_x = \frac{V_x(x+y)+x_0V_y-y_0V_x}{V_x+V_y}$; $C_y = \frac{V_y(x+y)-x_0V_y-y_0V_x}{V_x+V_y}$;

Wall (/): $C_x = \frac{V_x(y-x)+x_0V_y-y_0V_x}{V_y-V_x}$; $C_y = \frac{V_y(y-x)+x_0V_y-y_0V_x}{V_y-V_x}$;

Wall (|): $C_x = x$; $C_y = \frac{xV_y+x_0V_X-x_0V_y}{V_x}$;

Wall (–): $C_x = \frac{yV_x+x_0V_y-y_0V_x}{V_y}$; $C_y = y$;

In these calculations, $C_{x,\,y}$ is the coordinates of the intersection point, and x_0, y_0 is the coordinates of the ball.

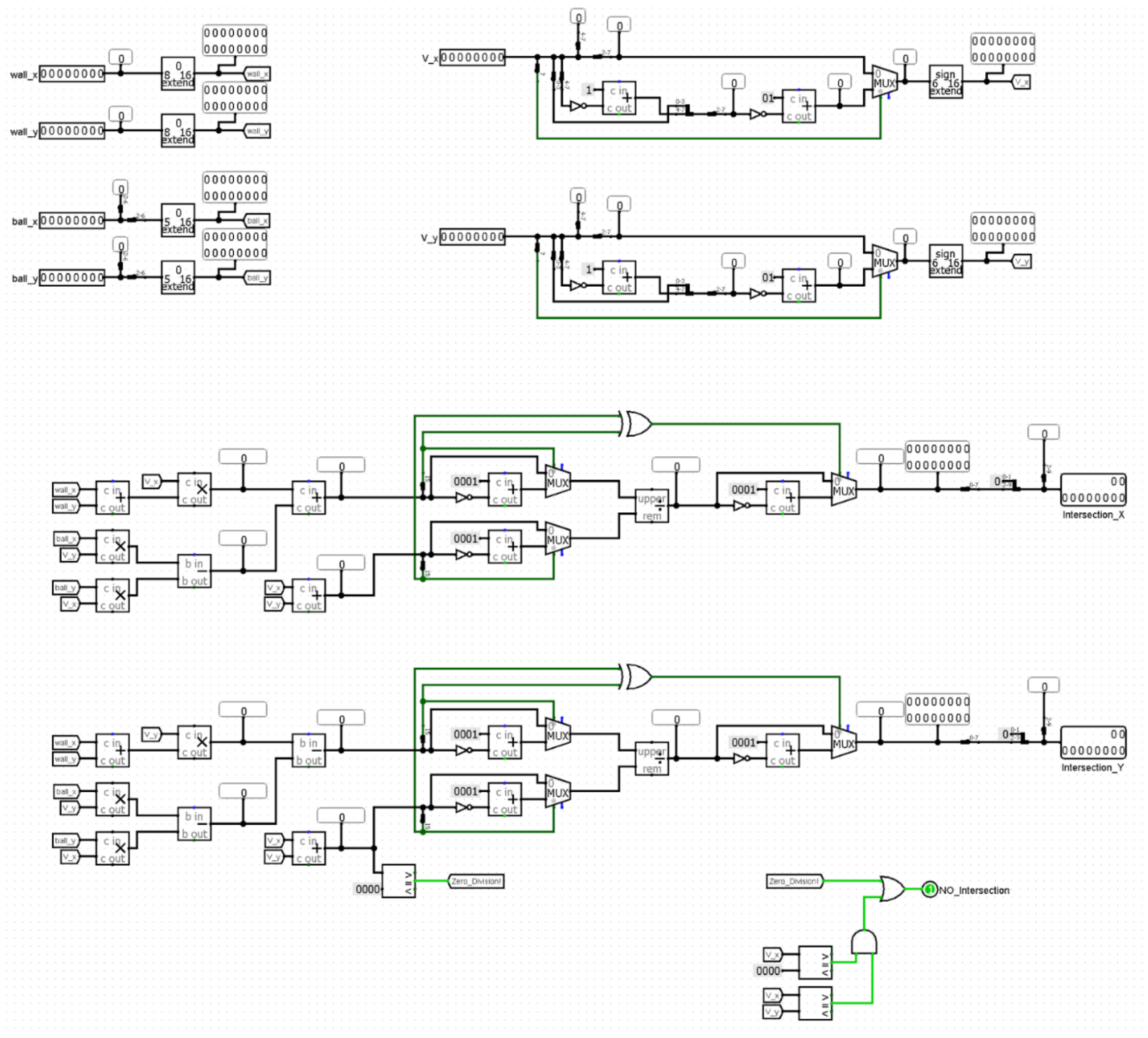


Figure 29 – image of the Intersection block (\)

The next part is responsible for extending the input coordinates to 16 bits in order to avoid overflow.

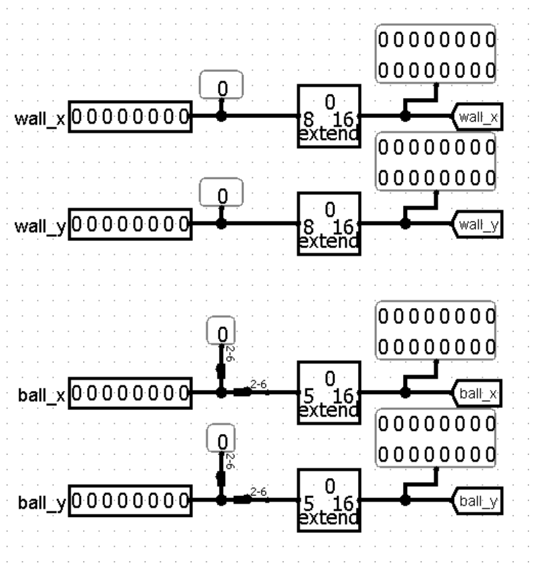


Figure 30 – coordinate extension

The block in Figure 31 accepts input velocity values (in `XXXXFFFF` format) and extends them to 16 bits. But it saves two `FF` bits that implement multiplication by 2. After all, if the speed is very low, then we still want to find out the intersection point and extend the velocity vectors.

If the velocity is positive, we just use the `XXXXFF` bits. Otherwise, we invert the 2's complement number and perform multiplication by 2, and then invert it again to get the correct result.

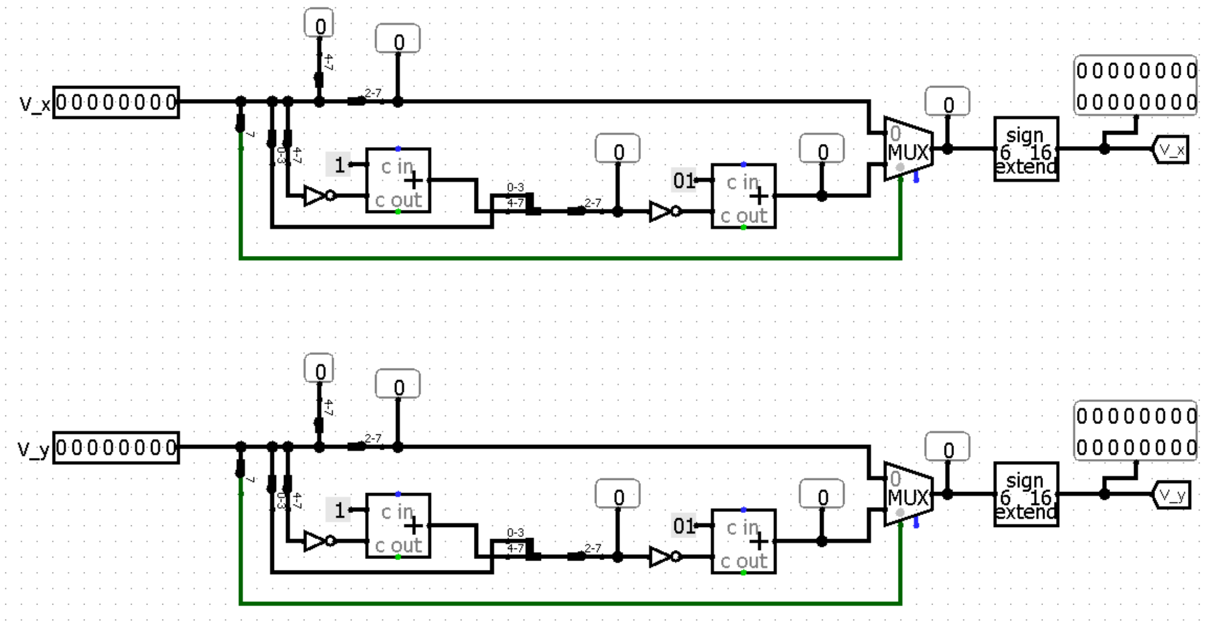


Figure 31 – data extension

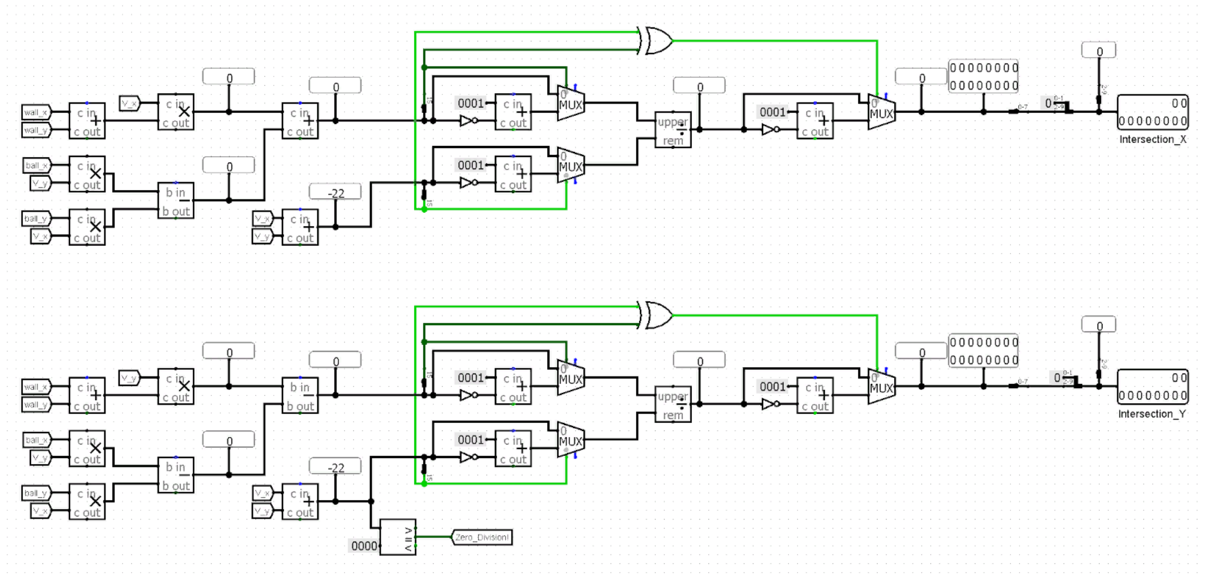


Figure 32 – implementation of the mathematical formula for the intersection point

The `Divisor` block can only work with unsigned numbers. If the input numbers are negative, then we invert them and the division result if it is also negative. We determine it by taking the sign bits from the 2 signals and XOR'ing them.

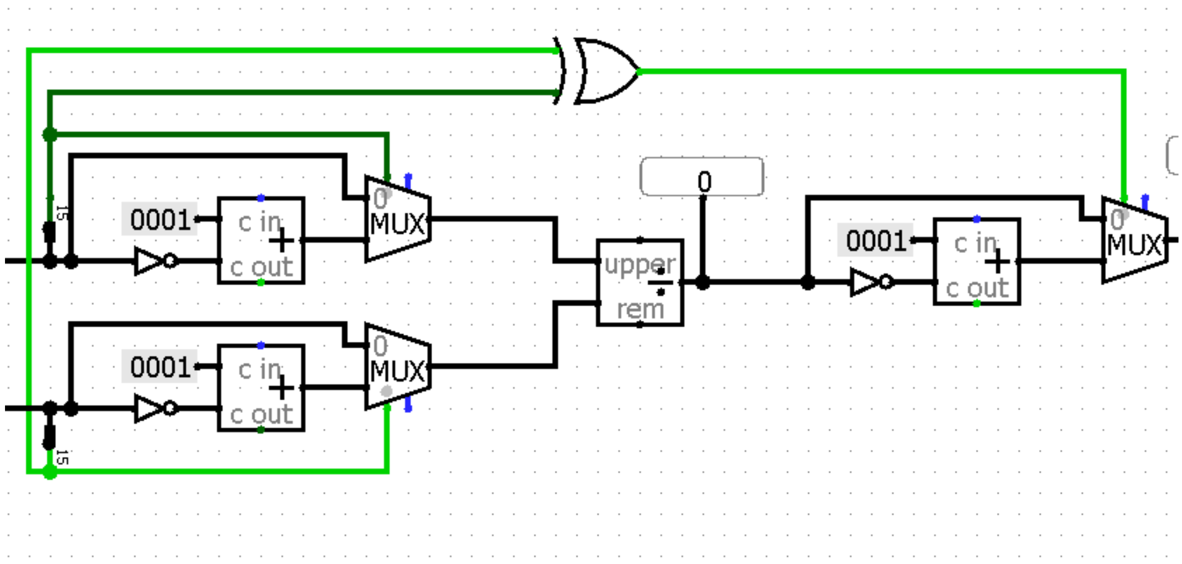


Figure 33 – the Divisor block

1.3.4. Sequential Logic

The screen is completely updated every 32 clock cycles, so we use a `Frame` counter in sequential logic. It counts clock cycles, updates the register of the main ball on the 31st clock cycle and resets all combinational logic on the 32nd.

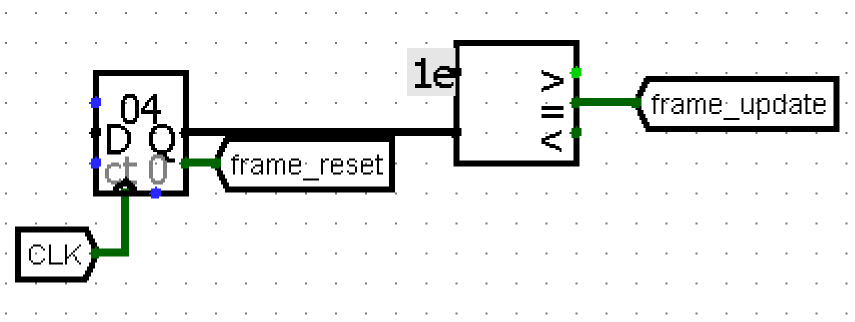


Figure 34 – the Frame counter

We also have a 1-bit register that stores information about the collision that occurred before the reset or the end of the frame.

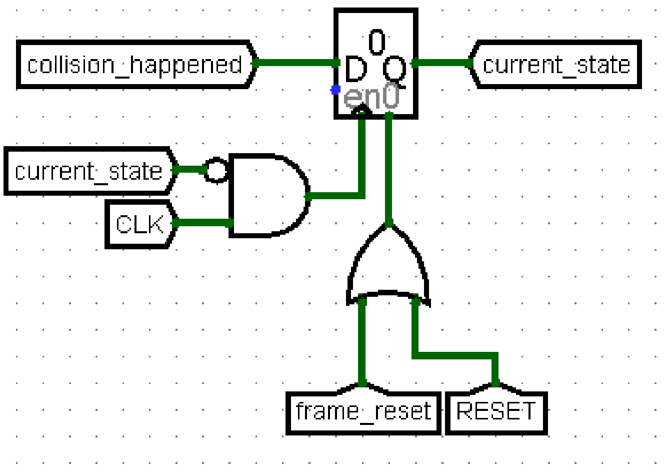


Figure 35 – storing information about a collision

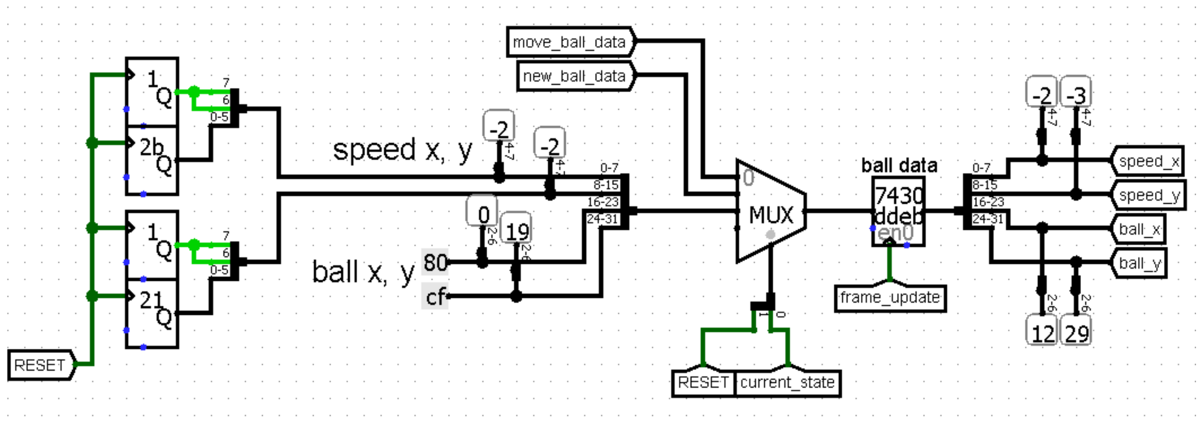


Figure 36 – a ball data register that contains all the information about the ball in 32 bits (8 bits for the X coordinate, 8 bits for Y and 8/8 bits for the X/Y velocity)

Before that, we have a MUX that selects which data bus to load into the register.

If the selected signal is 0, then there was no collision. Then we load `move_ball_data` into the register when updating the frame.

If the signal is 1, then there was a collision. We load `new_ball_data`.

If the signal is 2, then we reset the game and load the default X/Y coordinates and the random speed generated by 4 random blocks.

Also an important part is the `Wall Sequencer`, which processes from 0 to 7 walls on each screen, then sends this signal to the `Data Level` chip, which returns 4 walls to the `Ball` chip (one from each screen).

The next register contains the `collision_check_end` signal, which is needed to keep the `out_screen` signal from the `Move` chip from exiting and resetting the game until we check all the walls.

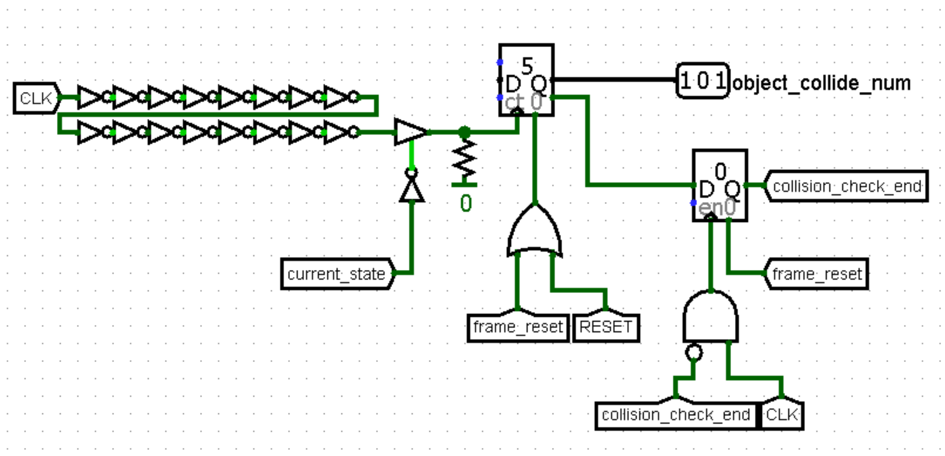


Figure 37 – the Wall Sequencer

1.4. Score counter

We have created a score counter to make the gameplay more interesting. This scheme checks for a collision of the ball with the wall. If the condition is met, the current score counter is increased by 1. If the ball leaves the field, the current counter is reset. Also at this stage, the status of the counter is updated if a greater result is obtained in the current game than in the previous one.

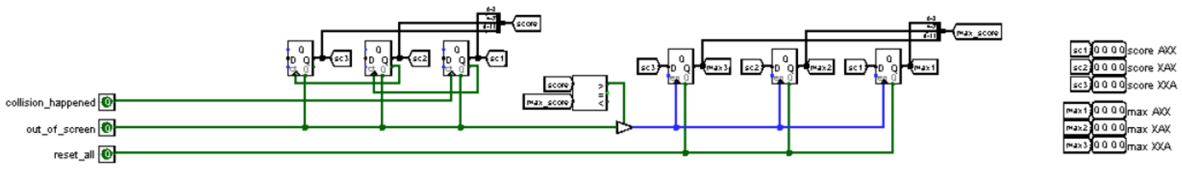


Figure 38 – the score counter

2. Software

2.1. The principle of operation of the Level Loader block

The software part of our project is implemented using the assembly language of the CdM-8-mark5 processor in the CocoIDE integrated development environment, created specifically for the development of code executed by this processor. We created the `Level Data` chip for loading the level from ROM and storing information about the walls in RAM. Let's take a closer look at how it works.

We have already described in the hardware part how we store the walls. This method of storing information about walls is not initially suitable for their rapid rendering on the screen. The `Level loader` block is needed to automatically get an image of the walls on the screen according to the available data. To solve this problem, we iterate through all the screens from 1 to 4, then all the walls inside each screen. The wall data is transmitted to the CdM-8, which is waiting for them as input. After that, the processor begins to output the coordinates of the pixels of the wall in turn. The assembler code runs a loop for a number of steps equal to the length of the wall. The starting coordinates of the wall are output at the first step, then the output coordinates change depending on the type of wall with each subsequent iteration of the cycle.

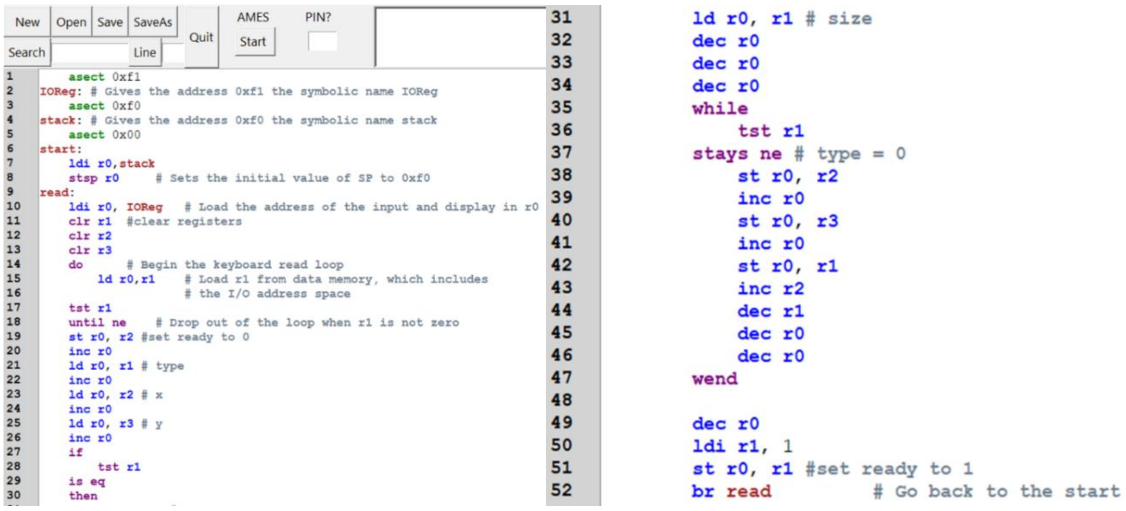
the wall type is 0 → only the variable X increases

the wall type is 1 → only the variable Y increases

the wall type is 2 → both variables X and Y increase

the wall type is 3 → the variable X increases and Y decreases

At the end of the loop, CdM-8 changes the value of the finished flag to 1 and waits for a new wall.



```
    asect 0xf1
IOReg: # Gives the address 0xf1 the symbolic name IOReg
    asect 0xf0
stack: # Gives the address 0xf0 the symbolic name stack
    asect 0x00
start:
    ldi r0,stack
    stsp r0     # Sets the initial value of SP to 0xf0
read:
    ldi r0, IOReg   # Load the address of the input and display in r0
    clr r1  #clear registers
    clr r2
    clr r3
    do      # Begin the keyboard read loop
        ld r0,r1    # Load r1 from data memory, which includes
                    # the I/O address space
    tst r1
    until ne    # Drop out of the loop when r1 is not zero
    st r0, r2 #set ready to 0
    inc r0
    ld r0, r1 # type
    inc r0
    ld r0, r2 # x
    inc r0
    ld r0, r3 # y
    inc r0
    if
        tst r1
    is eq
    then
```

```
ld r0, r1 # size
dec r0
dec r0
dec r0
while
    tst r1
stays ne # type = 0
    st r0, r2
    inc r0
    st r0, r3
    inc r0
    st r0, r1
    inc r2
    dec r1
    dec r0
    dec r0
wend

dec r0
ldi r1, 1
st r0, r1 #set ready to 1
br read        # Go back to the start
```

Figure 39 – code implementation

2.2. Description of the interaction of software and hardware parts

Logisim accepts the entire processor output and performs a bitwise operation OR with the desired column from the `Level Loader` RAM block responsible for the screen. Logisim moves to the next wall when the finished flag is detected and to the next screen when there are no other walls. The entire field is loaded when there are no other screens left to process. Then the `Level Loader` completes its work.

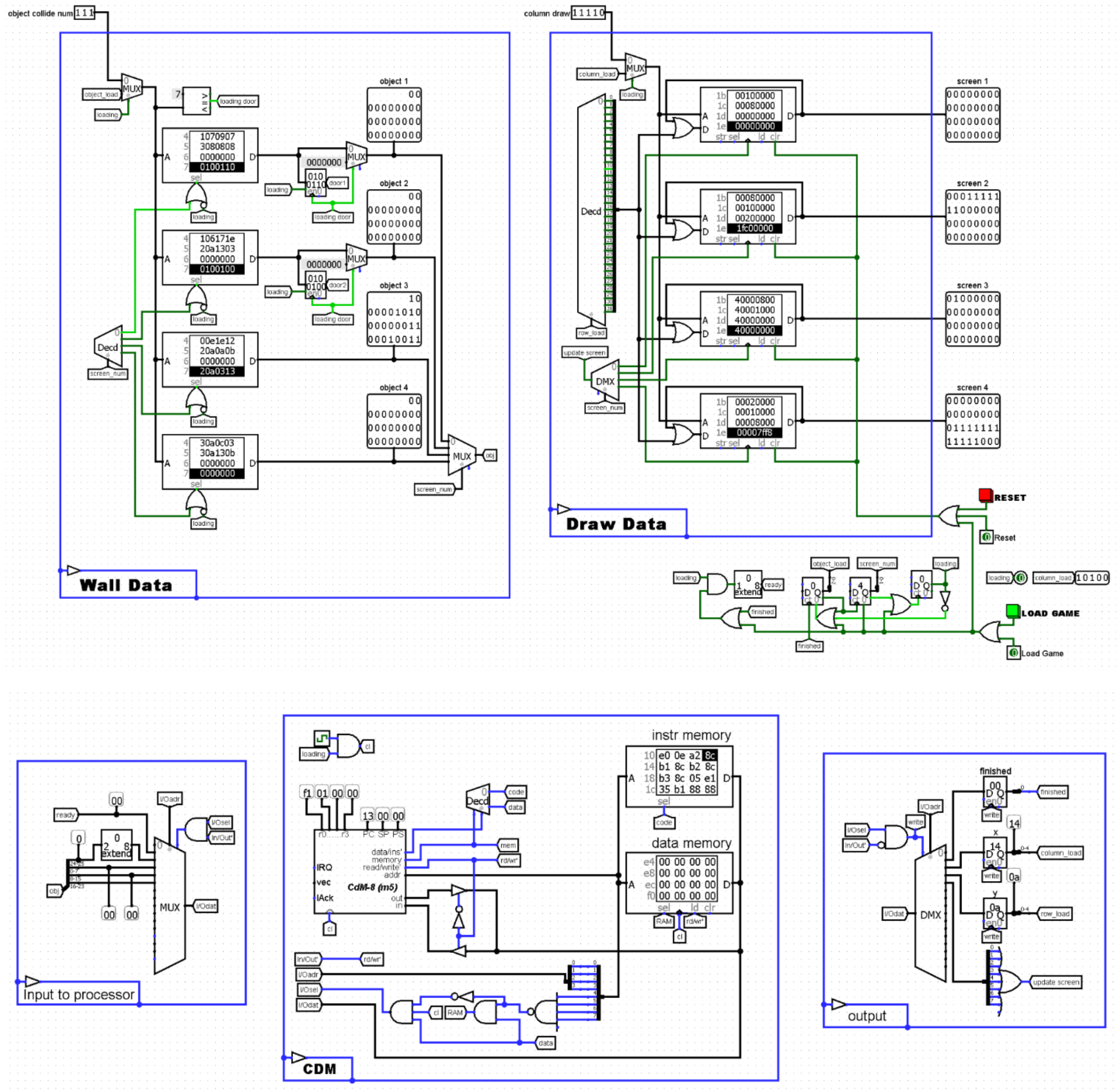


Figure 40 – CdM-8 usage

The advantage of `Level loader` is that with its help you can easily change the configuration of the field and even make several different levels for the game.

3. User manual

The main components of pinball are the playing field with obstacles (walls), the ball and the flippers (doors). There are only 4 types of walls: horizontal, vertical and 2 types of diagonal at an angle of 45 degrees. It is assumed that the playing field has a slight slope, so the force of gravity acts on the ball during the game, which is why it moves from top to bottom.

Before starting the game, you need to press the "LOAD GAME" button, then the "reset ball" button to launch the ball onto the playing field. Wait until the ball approaches the doors at the bottom of the field, then control the doors using the "left" and "right" buttons.

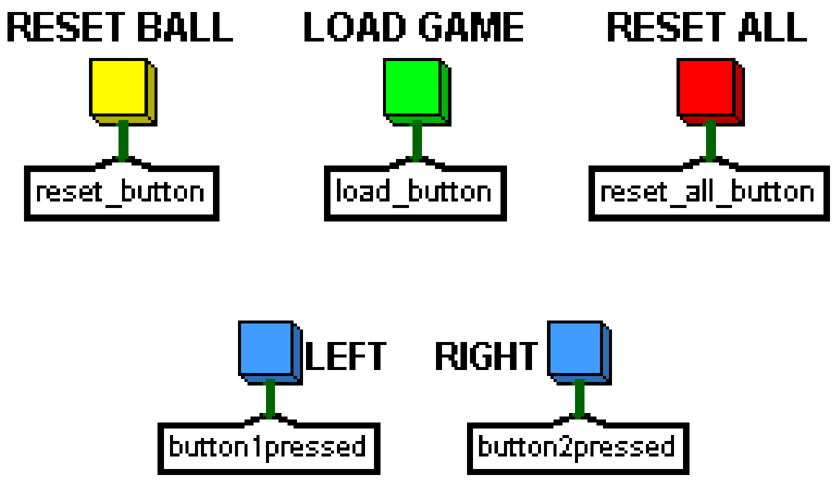


Figure 41 – game control buttons

The game has 2 flippers to control the ball. They are located at the bottom of the playing field, directly above the area of the ball loss. The doors perform 2 functions. Firstly, they do not allow the ball to fall into an area that is unacceptable for the game. Secondly, the player uses them to hit the ball towards the walls. The player is awarded points for colliding with walls. The flippers are controlled by two buttons that are located under the game area. The "left" button controls the left door, and the "right" button controls the right one.

The game continues as long as the ball is on the playing field. One point is awarded for each collision with the wall. You need to score as many points as possible. If the ball falls into an unacceptable area, then the game ends (the player ceases to earn points).

Conclusion

As a result of the work done, we managed to implement the Pinball game. We used digital logic circuits and the CdM-8 processor that executes the code we have written. The tasks we set at the beginning of the project were successfully solved. We have implemented all the specified functional requirements: the position of the door controlled by the player (on/ off), the uncontrolled movement of the ball under the influence of gravity, the player receiving points for each hit of the ball against the wall, and so on. Creating the project, we gained knowledge in the field of creating digital logic circuits, working with the processor and its programming, writing project documentation, and also gained valuable experience working in a team.