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

“Arkanoid”

Digital platforms project

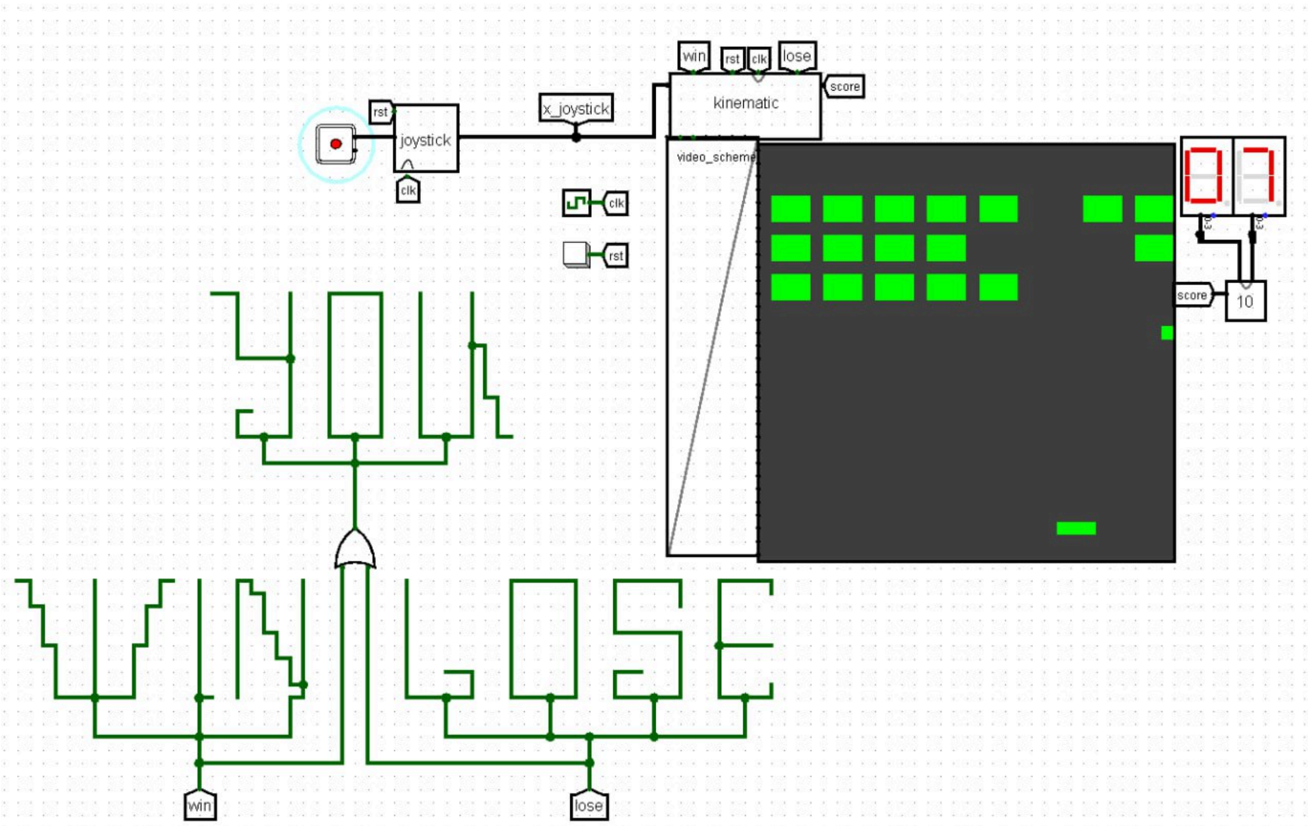
Anisimov Kirill
Lebedev Nikolay
Romankin Daniil

Novosibirsk State University
2024

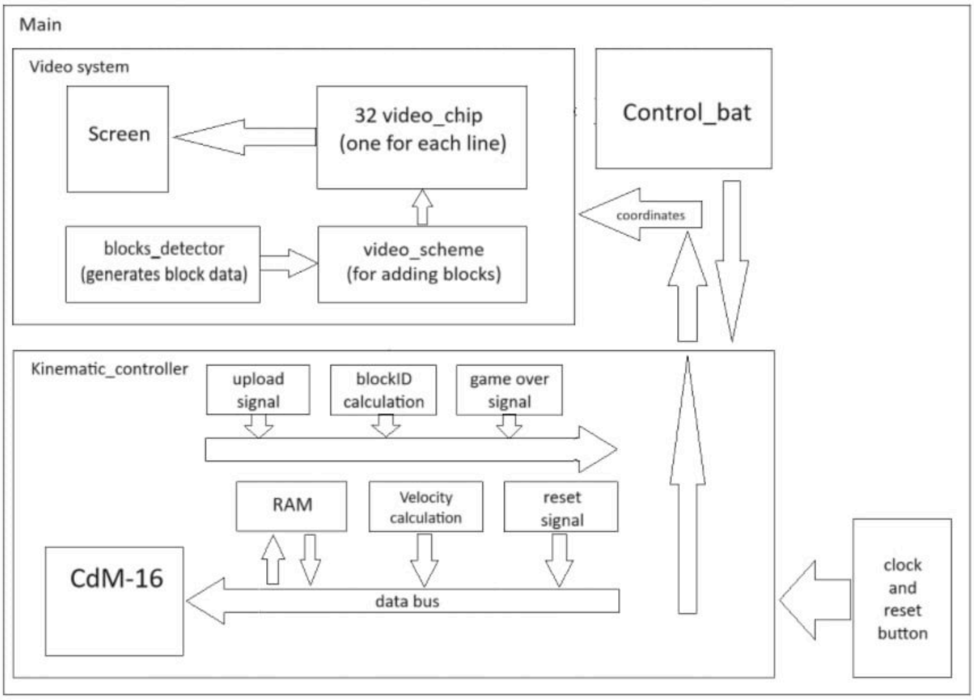
Contents

Introduction

Arkanoid is a classic arcade game in which the player controls a platform, hitting a ball and destroying blocks on the screen. The goal is to clear all the blocks from the field without letting the ball fall. The game ends when a player reaches 72 points (win) or when the ball reaches the bottom of the screen (lose). Our version of Arkanoid is a simplified variant of the original game. It offers one block placement option and one difficulty level. Additionally, the game provides different ball bounce settings and angles. To play, we have used a video display with a resolution of 32 x 32 pixels and a kinematic controller chip to move the bat and ball on the display.



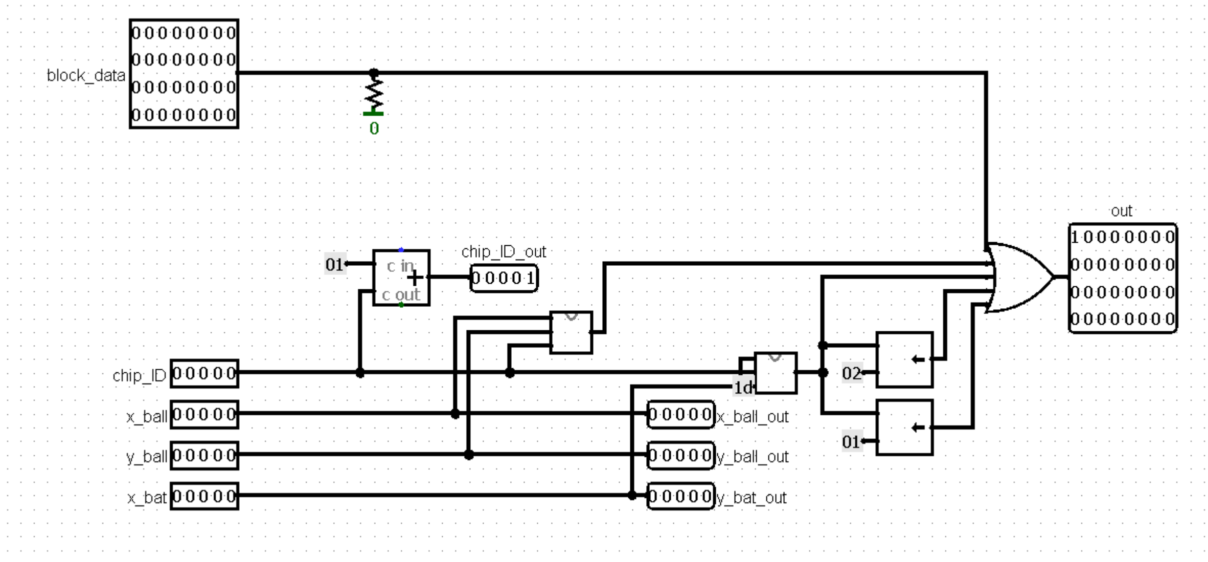
1 Hardware



1.1 Video system

The video system has a display with a resolution of 1024 pixels by 32. Each line of the display is composed of 32 individual pixels, and the lines are numbered sequentially from 0 to 31 on the left side of the screen. The pixels within each line are also numbered consecutively from the top of the screen to the bottom, beginning with 0 and ending at 31. The video chip and video scheme are located within the subsystem. This circuit is responsible for drawing one line at a time on the screen, following the specified numbering pattern.

1.1.1 Video_chip

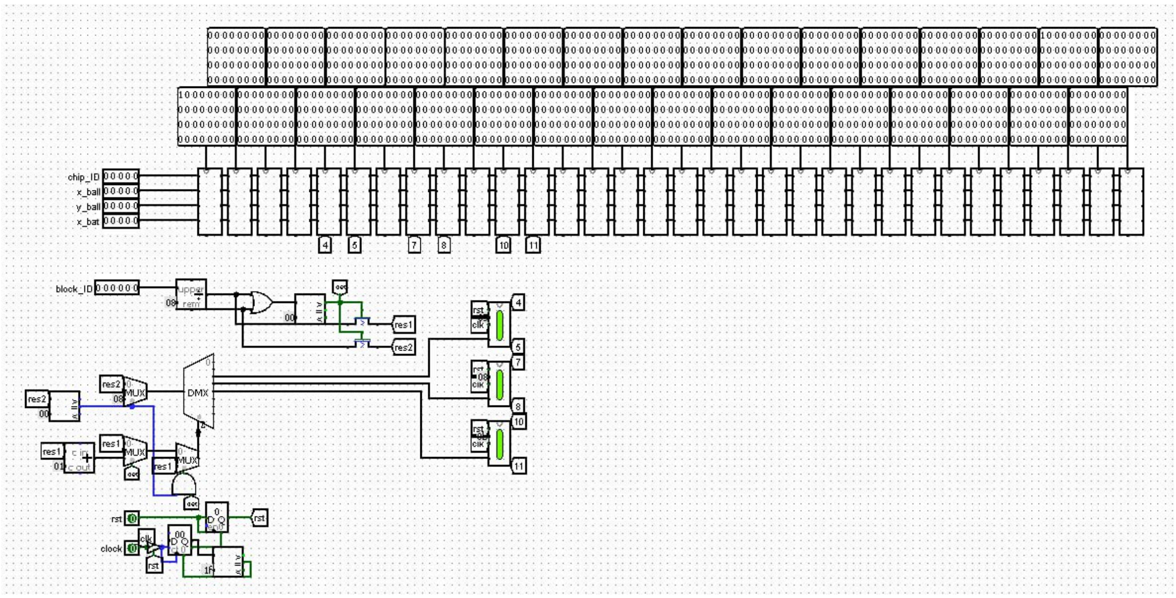


The circuit draws a line on the display. The input data includes the *chip_ID* from 0 to 31, the coordinates of the ball (*x_ball* and *y_ball*), the position of the right edge of the bat (*x_bat*), and information about blocks in the line.

The *chip_ID* sends a signal indicating whether drawing or data transfer is required. The bat is displayed in chip 29 and the ball in the same chip as its y-coordinate. The rocket is created by combining three 32-bit values: the original value at the bat's location, and two additional values shifted left by 1 and 2 bits.

Output is transmitted as a string that will appear on the display, including the coordinates of the ball and bat (*x_ball_out, y_ball_out, x_bat_out*) and a *chip_ID* value incremented by 1.

1.1.2 Video_scheme

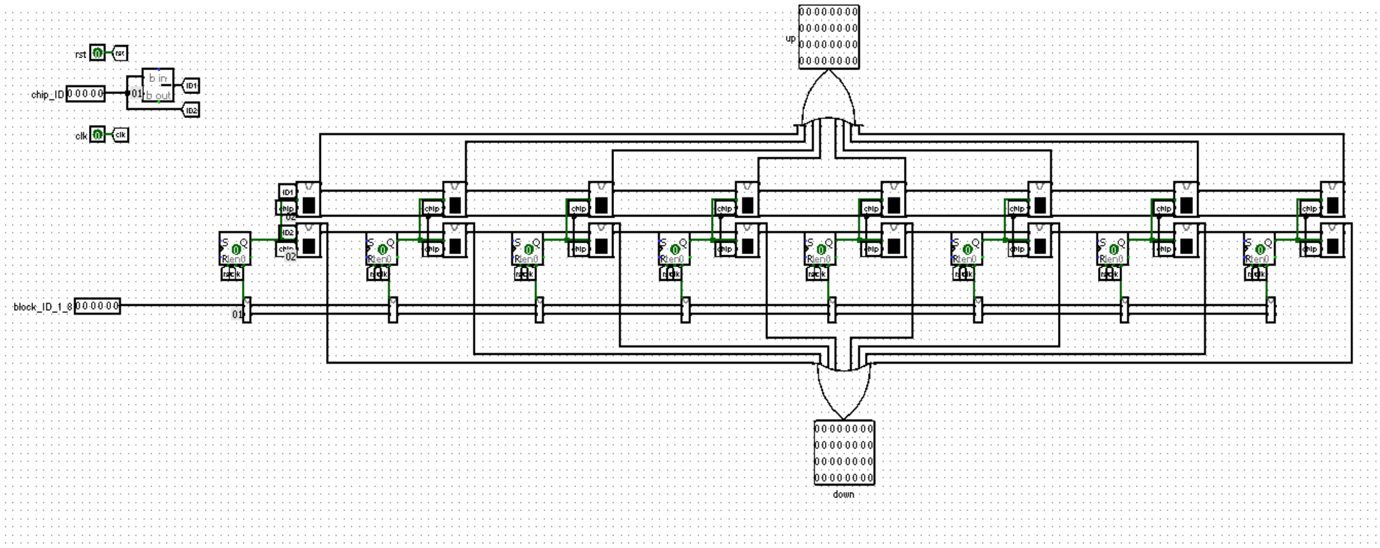


The circuit consists of 32 video chips and 3 detector block circuits. It establishes a connection between the video chips and the display, as well as converting the *block_ID* into a signal that removes the desired block from the display when the ball hits it.

The *block_ID* is a unique identifier for each block on the display, ranging from 1 to 32, from top left to bottom right. When the ID is divided by eight, the resulting remainder ranges from 1 to 8 and is fed into a specific detector block circuit based on the quotient. An 32-bit signal is sent to the input of the selected video chip.

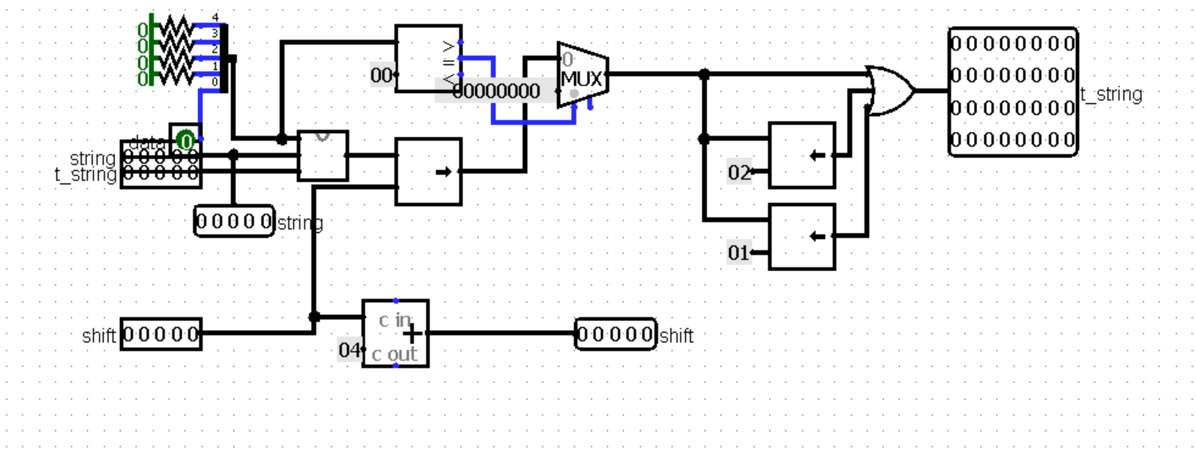
Input to the circuit includes the x-coordinate of the bat (*x_ball* and *y_ball*), *chip ID*, *block ID*, a *rst* signal (which will be described later), and a *clock* signal (provided by a clock generator).

1.1.3 Block_detector



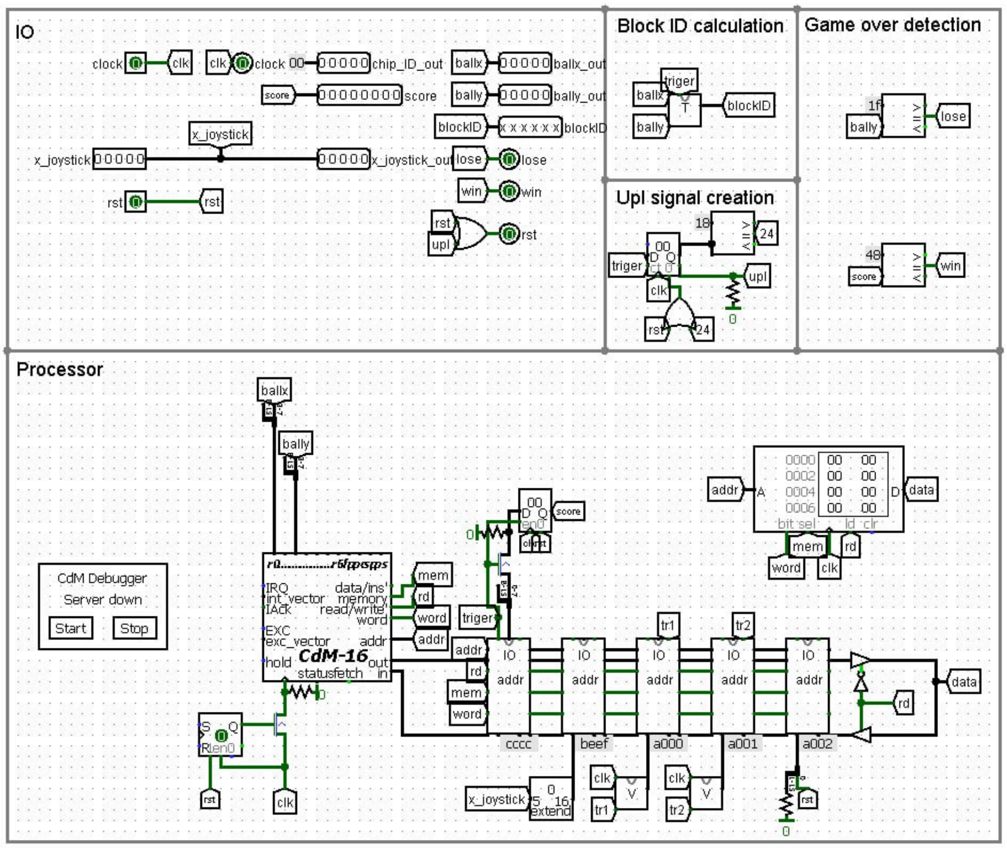
The circuit is responsible for generating the data required to draw blocks. The inputs to the circuit are the *rst*, *clock*, *chip_ID*, and *block_ID*. Based on the *block_ID*, it determines which S trigger should send a signal to erase the block (asynchronously load zero). Using two **block_paint** schemes for each of the eight CP triggers, the block is then drawn in the desired location.

1.1.4 Block_paint



In circuit, data for drawing blocks is generated. The input consists of the *data* (trigger CP value), a *string*, and *t_string* for comparison, as well as a *shift* parameter (determining how much the line should be shifted to the right to draw the block at the desired position). If the *data* is zero, a string of 32 zeroes is sent to the *t_string* output. The *shift* value is increased by 4 and passed to the next **block_paint** block.

1.2 Kinematic_controller



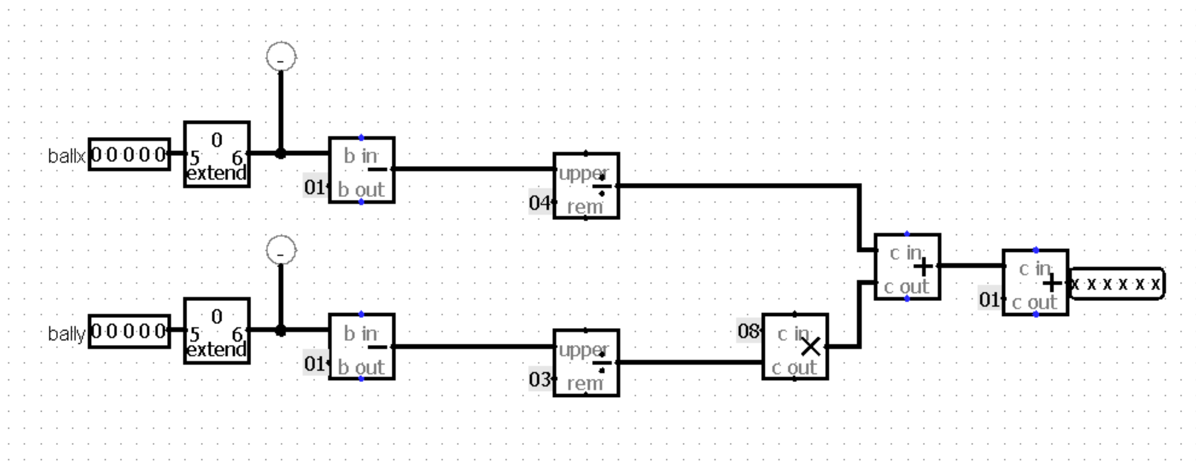
In scheme, interaction with the processor takes place, the *block_ID* is calculated and a signal indicating victory or defeat is generated.
The CDM-16 processor in the Von Neumann configuration was selected for its ability to handle a large number of memory locations and accelerate

command execution. This allowed for the implementation of all collision detection and resolution functionality within the device.

The entire assembler code for the game is stored in RAM (random access memory). To enhance the gameplay experience, a system for randomly generating velocities after each collision was implemented, making the trajectory of the ball unpredictable. The new velocity is calculated and communicated to the processor for processing.

1.2.1 BlockID calculation

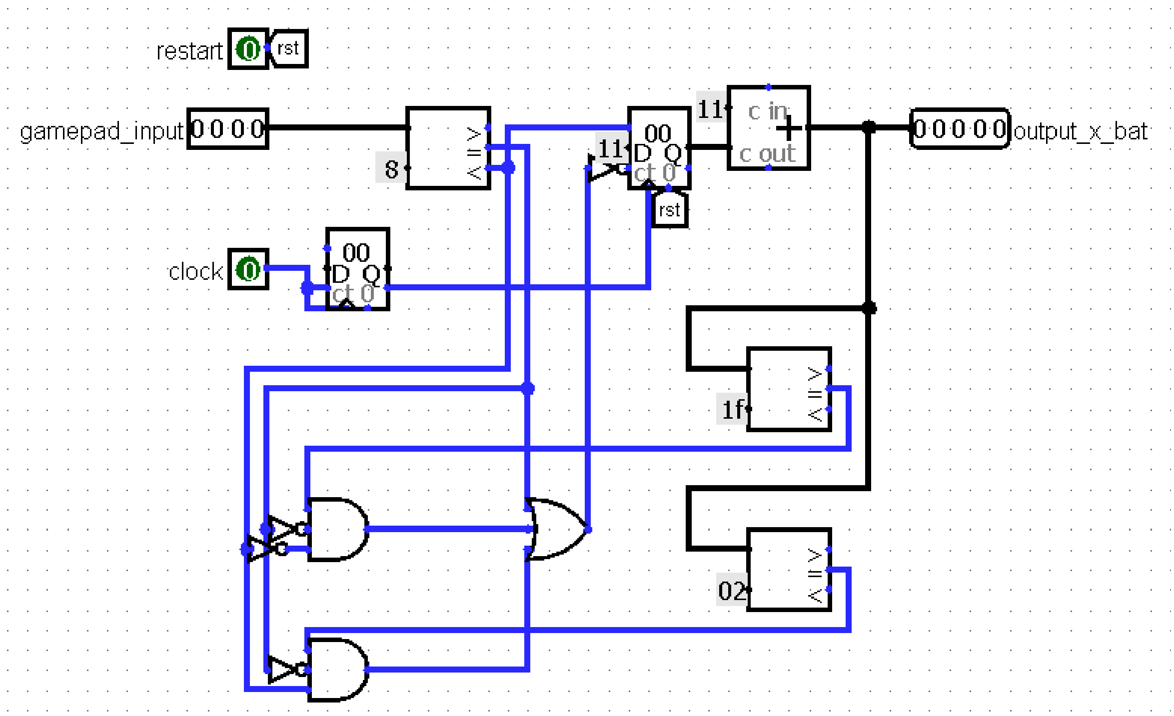
OBJ



In scheme, the *block_ID* is calculated using the formula

$$((bally - 1) // 3) * 8 + ((ballx - 1) // 4) + 1$$

1.2.2 Bat_control



The circuit is responsible for translating the input value from the joystick into the x-coordinate of the rightmost pixel of the bat (*output_x_bat*), which ranges from 2 to 31. The input is a number (*gamepad_input*), which can range from 1 to 16. If this number is less than 8, the coordinate will be decreased by 1; if it is greater than or equal to 8, it will be increased by 1. If the number is equal to 8, nothing will happen. Additionally, restrictions have been implemented to prevent the bat from moving to the left of the screen when moving right, and to prevent it from hitting the right side of the screen when moving left. An artificial delay in the rate at which the bat's coordinates change has been implemented due to the addition of an extra counter, which is controlled by the clock frequency.

2 Cdm-16 software

The CDM-16 assembler program performs several important functions, including:

- Calculating ball physics
- Storing the game score at the specified address (**0xcccc**)
- Storing and updating block states
- Handling collisions
- Resetting the game state (reset signal stored at address **0xa002**)
- Reading the x-coordinate (5-bit) of the right pixel of the bat from the Logisim circuit at address **0xbeef** and random velocities stored at addresses **0xa000** and **0xa001**

In our program, we have a game field with a size of 256 x 256, and 8-bit values for coordinates and velocity, which range from 5 to 8 and are generated by a random number generator in Logisim. We use the most significant 5 bits of the coordinates for accurate display in Logisim.

All the blocks are stored in six arrays of zeros and ones, where each array corresponds to a specific Logisim display string (4, 5, 7, 8, 10, or 11).

```
# arrays of blocks, every array corresponds to logisim display string
bricks4: dc 0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0
bricks5: dc 0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0

bricks7: dc 0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0
bricks8: dc 0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0

bricks10: dc 0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0
bricks11: dc 0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0
```

We appended one zero to each array in order to facilitate the handling of collisions. The program begins by setting the initial values:

```
main>
#we use:
#r0 -  x-coordinate
#r1 -  x-velocity
#r2 -  y-coordinate
#r3 -  y-velocity
#r4,r5,r6 - for temporary calculations
ei
ldi r0,128 #x-coordinate
ldi r1,0xa000 #random x-velocity
ldb r1,r1
ldi r2,224 #y-coordinate
ldi r3,0xa001 #random y-velocity
ldb r3,r3
neg r3
```

Then, we initiate a while loop with a condition "while the y-coordinate is less than 248" (i.e., when the ball reaches the bottom).

```
while
    ldi r6,248  #comparing y-coordinate
    cmp r2,r6
stays lt
```

After that, we verify for a reset signal, and if it is equal to 1, we reset the score to zero and proceed to the restart subroutine.

```
ldi r5,0xa002
ldb r5,r5
if
    cmp r5,1   #checking for reset
is eq
    ldi r6,0xcccc
    ldi r5,0
    st r6,r5
    ldi r6,0xb001
    stb r6,r5
    ldi r6,0xb002
    stb r6,r5
    jsr restart # if reset signal is 1, jump to restart subroutine
fi
```

The restart subroutine is called when the level ends or when the player presses the reset button.

```
restart: #subroutine that called if reset or level ends
    ldi r6,0xcccc
    ld r6,r6
    if
        ldi r5,24
        cmp r6,r5
    is eq
        ldi r5,0xb001 #setting the flag, that means that restart procedure happened to this score value
        ld r5,r4
        inc r4
        stb r5,r4
    fi
    if
        ldi r5,48
        cmp r6,r5
    is eq
        ldi r5,0xb002
        ld r5,r4
        inc r4
        stb r5,r4
    fi
    if
        ldi r5,72
        cmp r6,r5 #end of game
    is eq
        halt
    fi
```

It sets the flags to different values to prevent re-entry into the subroutine on each new loop iteration, and the game ends at 72 points.

Then, we update the values in the block arrays to the starting values.

```
    ldi r6,0
    ldi r5,bricks4
    ldi r4, bricks5 #rewriting the arrays of blocks
    while
        cmp r6,8
    stays lt
        ldi r3,0
        stb r5,r3
        stb r4,r3
        ldi r3,1
        add r5,2
        add r4,2
        stb r5,r3
        stb r4,r3
        add r5,2
        add r4,2
        stb r5,r3
        stb r4,r3
        add r5,2
        add r4,2
        stb r5,r3
        stb r4,r3
        add r5,2
        add r4,2
        inc r6
    wend
```

We use the "reset" instruction to jump back to the beginning of the program.

```
wend
reset #reseting the programm
rts
```

After checking for reset signals and score values, the physics calculation section begins.

We use *r4* as a temporary register to precalculate the x-coordinate, and copy the value from *r4* into the actual register *r0* for accurate display of the ball. At this point, we read the random x-velocity from Logisim.

```
add r1,r0,r4 #we use r4 register to pre-calculate x-coordinate for correct display of the ball
#checking wall collisions by x-coordinate
if
    ldi r6, 255
    cmp r4,r6
is gt
    neg r1
    add r1,r4,r4
    ldi r6,0xa000 # loading random Vx
    ldb r6,r6
    if
        cmp r1,0
    is lt
        neg r6
        move r6,r1
    else
        move r6,r1
    fi
fi
```

We check for collisions with blocks by examining the y-coordinate. If the y-coordinate is equal to the block row coordinate, we load the address of the necessary array, extract the 5 most significant bits from the coordinate, and add this value to our address twice (because the values in the array are stored as 16-bit quantities, so we must advance two bytes). If the value at the resulting address is equal, a collision has occurred and we must update the score. Next, we need to determine whether the left or right pixel was struck in order to correctly update the memory values. We can do this by adding two to our address: if the value at this new address is 1, the left pixel has been struck; otherwise, the right pixel.

```
ldi r5, bricks11 #loading adress of necessary array
move r0,r4 #copying x-coordinate
shr r4 #taking 5 most significant bits from 8-bit value of x-coordinate
shr r4
shr r4
add r5,r4,r5 #adding x-coordinate to the adress of array
add r5,r4,r5
push r1
ld r5,r1
if
    cmp r1, 1 #checking whether a collision occured
is eq # we need to know if this left or right pixel
    push r5
    ldi r6,0xcccc #updating the score, which is stored at 0xcccc adress
    ld r6,r5
    inc r5
    stb r6,r5
    pop r5
    add r5,2    # if we increase adress by 1 x-coordinate, and value by that adress equals 1
                # we are adding 2 to r5 because we are working with 16-bit values
                #it is left pixel, else it is right pixel

    ld r5,r1
```

We are updating memory values and adjusting random access speed.

```
dec r1      # updating values in current row
sub r5,2
st r5,r1
add r5,2
st r5,r1
add r5,2
st r5,r1
ldi r5, bricks10 # updating values in adjacent row
add r5,r4,r5
add r5,r4,r5
st r5,r1
add r5,2
st r5,r1
add r5,2
st r5,r1
pop r1
neg r1
ldi r6,0xa000 #loading random Vx
ldb r6,r6
if
    cmp r1,0
is lt
    neg r6
    move r6,r1
else
    move r6,r1
fi
```

We are performing this procedure on other block arrays. Following that, we check for y-axis collisions. Wall collisions are checked in the same manner as with the x-axis.

We must check for collisions with the bat. First, we check the collision with the right pixel, then the middle, and finally the left (note: if the ball hits an edge pixel, the Vx and Vy values are negated; if the ball hits the middle pixel, only the Vy value is negated).

```
    then
        ldi r5,0xbeef
        ld r5,r4 #now it has coordinate of bat
        move r0,r6
        shr r6
        shr r6
        shr r6 #taking 5 most significant bits of 8-bit x-coordinate
        if
            cmp r6,r4   #checking whether a collision occurred with right pixel of the bat
        is eq
            neg r3  # if ball hits edge pixels of bat, vx and vy velocities are being negated,
                    # if ball hits central pixel - only vy
            neg r1
            ldi r6,0xa001 #loading random Vy
            ldb r6,r6
            if
                cmp r3,0
            is lt
                neg r6
                move r6,r3
            else
                move r6,r3
            fi

        else
            dec r4 #checking whether a collision occurred with middle pixel of the bat
            if
                cmp r6,r4
            is eq
                neg r3
                ldi r6,0xa001   #loading random Vy
                ldb r6,r6
                if
                    cmp r3,0
                is lt
                    neg r6
                    move r6,r3
                else
                    move r6,r3
                fi

            else
                dec r4 #checking whether a collision occurred with left pixel of the bat
                if
                    cmp r6,r4
                is eq
                    neg r3
                    neg r1
                    ldi r6,0xa001   #loading random Vy
                    ldb r6,r6
                    if
                        cmp r3,0
                    is lt
                        neg r6
                        move r6,r3
                    else
                        move r6,r3
                    fi

                else

                fi
            fi
```

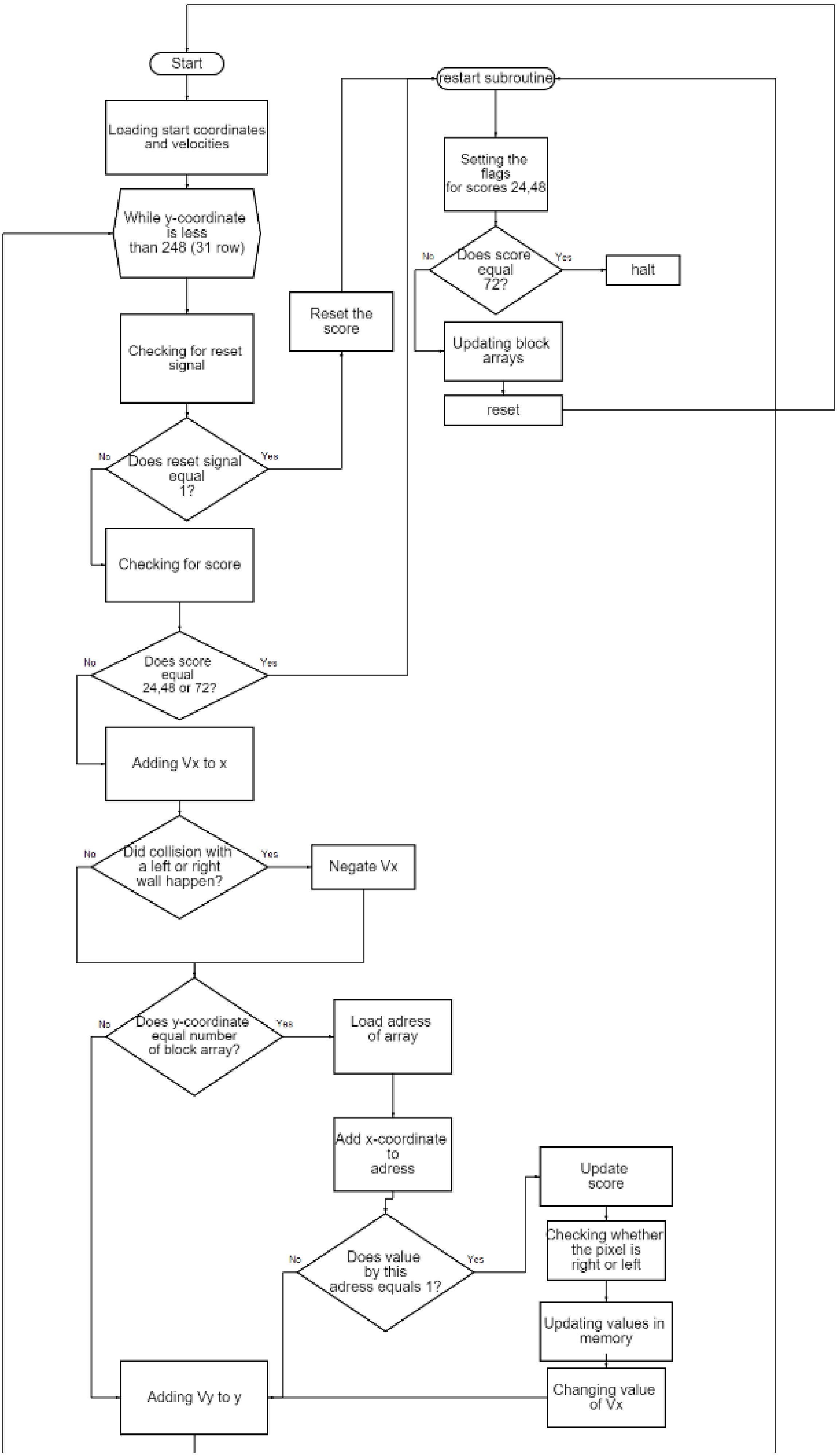
We use the same algorithm for processing collisions in both horizontal and vertical directions, with the exception of additional verification for the middle pixel. If the value at the current address is 0, we assume that it is the right pixel. However, if the value is not 0, we move two pixels backward and check the value at the new address. If it is 0, it is the left pixel; otherwise, it is considered the middle pixel.

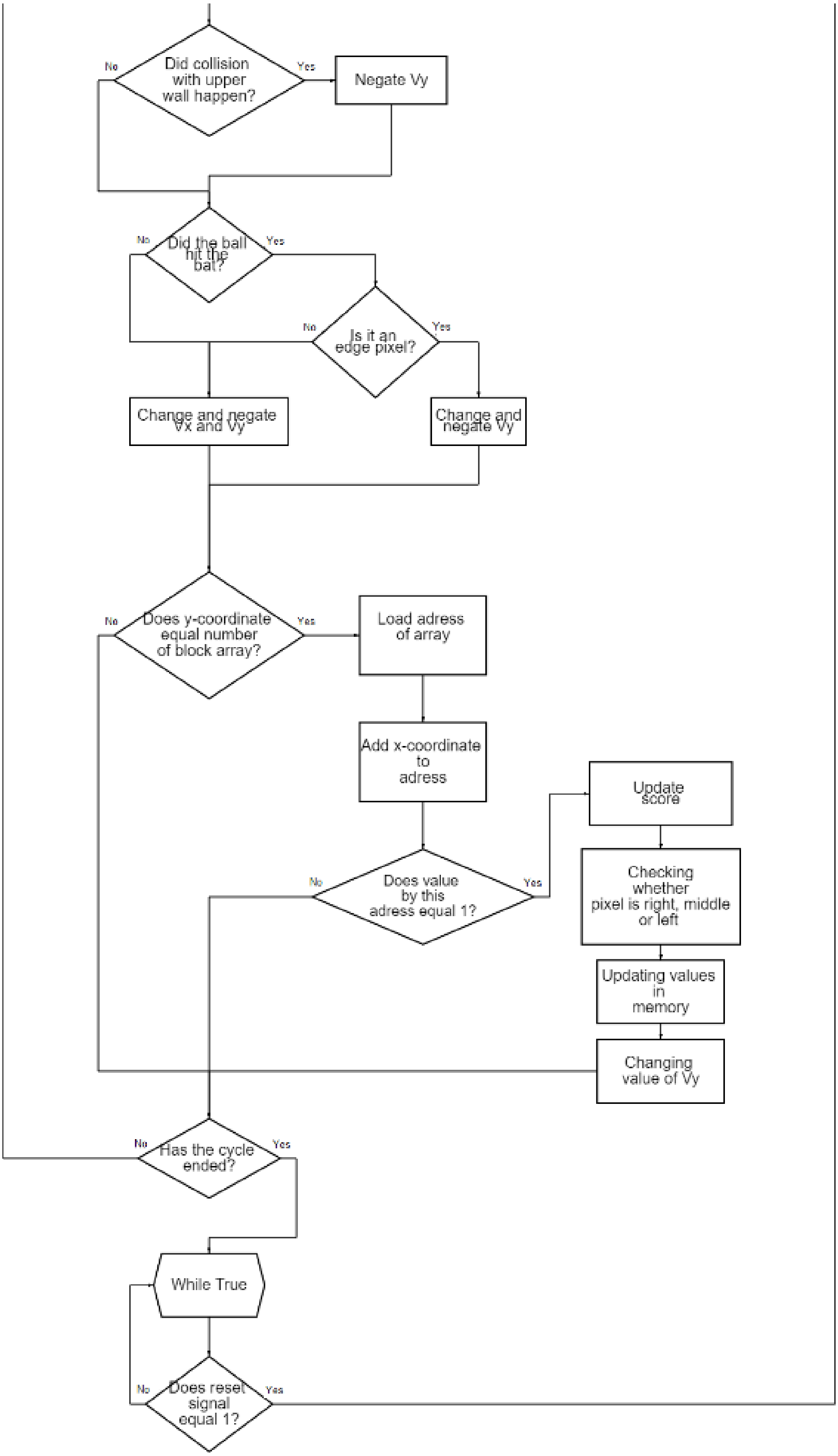
```
        ld r5,r1
        if
            cmp r1,1 #checking for collision
        is eq
            push r5
            ldi r6,0×cccc #updating score
            ld r6,r5
            inc r5
            stb r6,r5
            pop r5
            add r5,2 #moving one pixel forward, if it is zero, it means that ball hit the right pixel
            ld r5,r1
            if
                cmp r1,1
            is eq
                sub r5,2    #moving back two pixels, if it is zero - it means that ball hit the left pixel
                            # else - middle pixel
                sub r5,2
                ld r5,r1
```

If the ball reaches the 31st row, the cycle ends. We then start a new "while true" loop and wait for a reset signal.

```
ldi r6,2
while
    cmp r6,2 #starting infinite loop
stays eq
    ldi r5,0×a002
    ldb r5,r5
    if
        cmp r5,1
    is eq
        ldi r6,0×cccc # if reset signal equals 1 - jump to restart subroutine
        ldi r5,0
        st r6,r5
        ldi r6,0×b001
        stb r6,r5
        ldi r6,0×b002
        stb r6,r5
        jsr restart
    fi
wend
halt
```

The full version of the code can be found here: https://github.com/Nikolay56615/Arkanoid.git


Start
Loading start coordinates and velocities
While y-coordinate is less than 248 (31 row)
Checking for reset signal
Does reset signal equal 1?
Yes
No
Reset the score
Checking for score
Does score equal 24,48 or 72?
Yes
No
Adding Vx to x
Did collision with a left or right wall happen?
Yes
No
Negate Vx
Does y-coordinate equal number of block array?
Yes
No
Load adress of array
Add x-coordinate to adress
Does value by this adress equals 1?
Yes
No
Update score
Checking whether the pixel is right or left
Updating values in memory
Changing value of Vx
Adding Vy to y
restart subroutine
Setting the flags for scores 24,48
Does score equal 72?
Yes
No
halt
Updating block arrays
reset



Did collision with upper wall happen?
No
Yes
Negate Vy
Did the ball hit the bat?
No
Yes
Is it an edge pixel?
No
Yes
Change and negate Vx and Vy
Change and negate Vy
Does y-coordinate equal number of block array?
No
Yes
Load adress of array
Add x-coordinate to adress
Update score
Does value by this adress equal 1?
No
Yes
Checking whether pixel is right, middle or left
Updating values in memory
Changing value of Vy
Has the cycle ended?
No
Yes
While True
Does reset signal equal 1?
No
Yes

3 Interaction between software and hardware

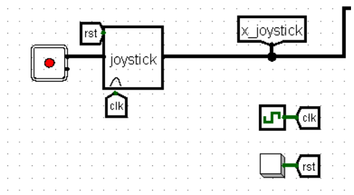
From Hardware:

- Random values of ball speed modules *Vx, Vy*
- *Reset* signal (restarts the program)
- Coordinate of the right pixel of the joystick

From Software:

- ball *x* and *y* coordinates
- game score

4 User manual



The game starts by clicking on the first (reset) button. To control the bat, a joystick is used. The joystick moves by clicking on the mouse. If the player wishes to generate blocks again, they should click on the *rst* (reset) button. If they wish to restart the game from the beginning, they should hold down the *rst* (reset) button.

Conclusion

For our team, the project work has been very interesting. We have gained a significant amount of experience through working in a collaborative environment, writing assembly code, designing chips, and creating documentation.