

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

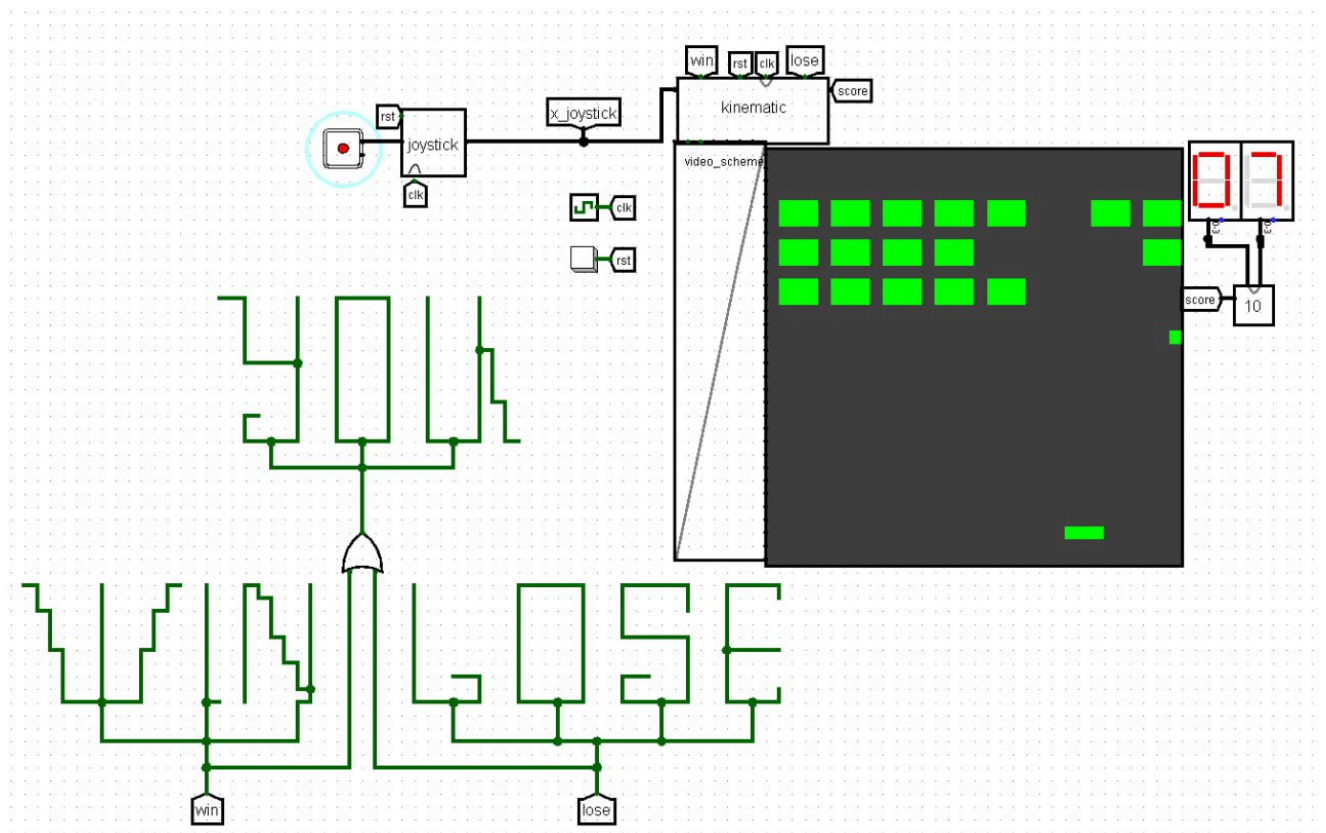
Anisimov Kirill
Lebedev Nikolay
Romankin Daniil

Contents

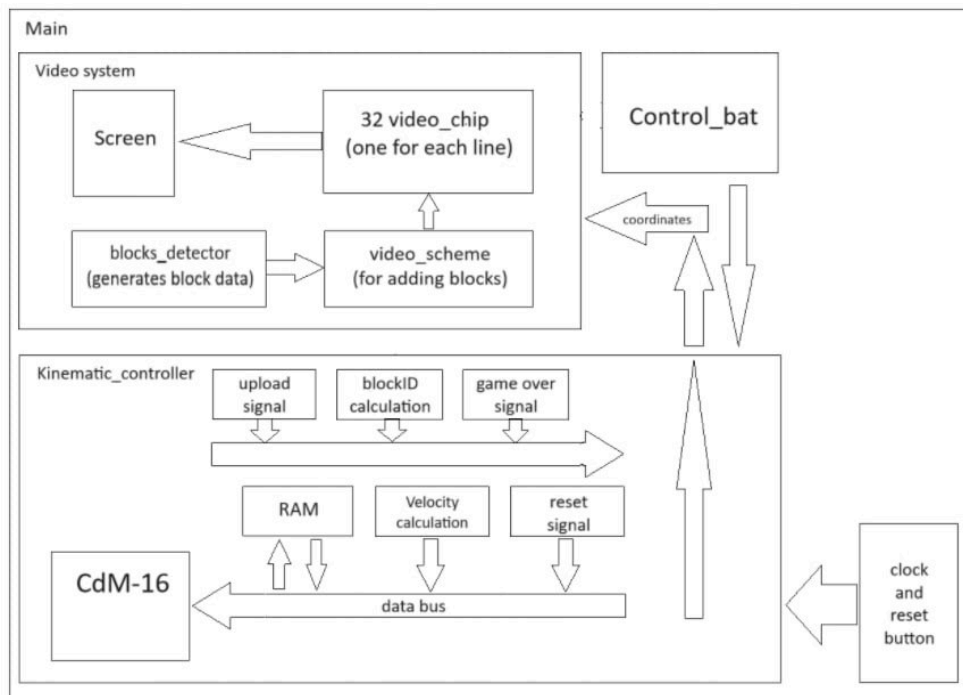
Introduction	2
1. Hardware	3
1.1 Video system	3
1.1.1 Video_chip	4
1.1.2 Video_scheme	5
1.2 Blocks_detector	6
1.3 Block_paint	6
1.4 Block_ID calculation	7
1.5 Bat_control	8
1.6 Kinematic_controller	9
2. Cdm-16 software	10
3. Interaction between software and hardware	19
4. User manual	19
Conclusion	20

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.



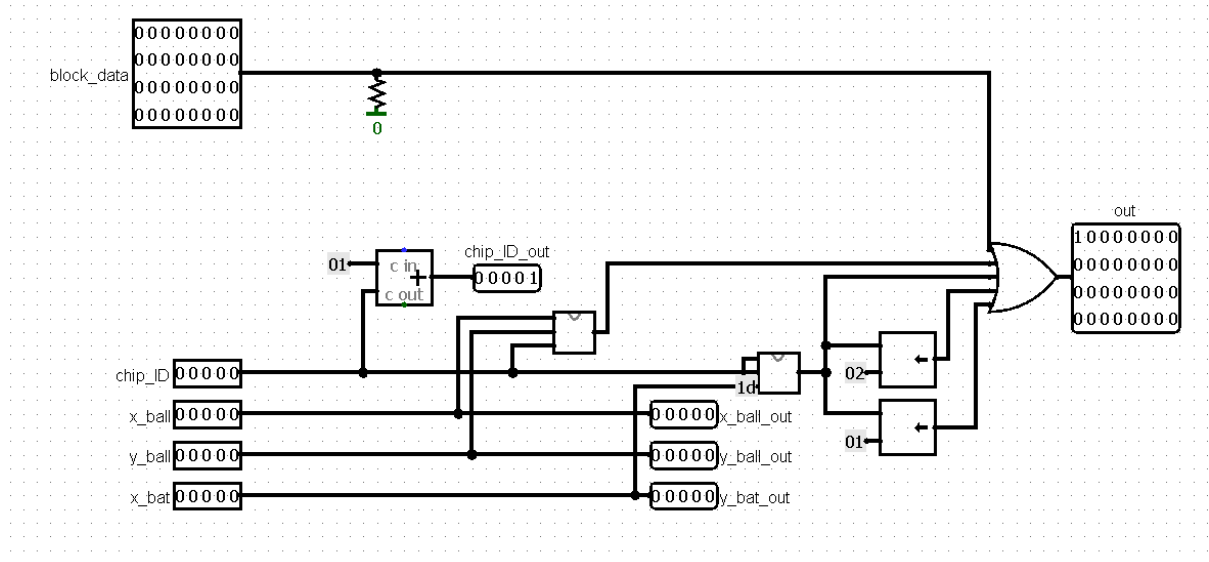
1 Hardware



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

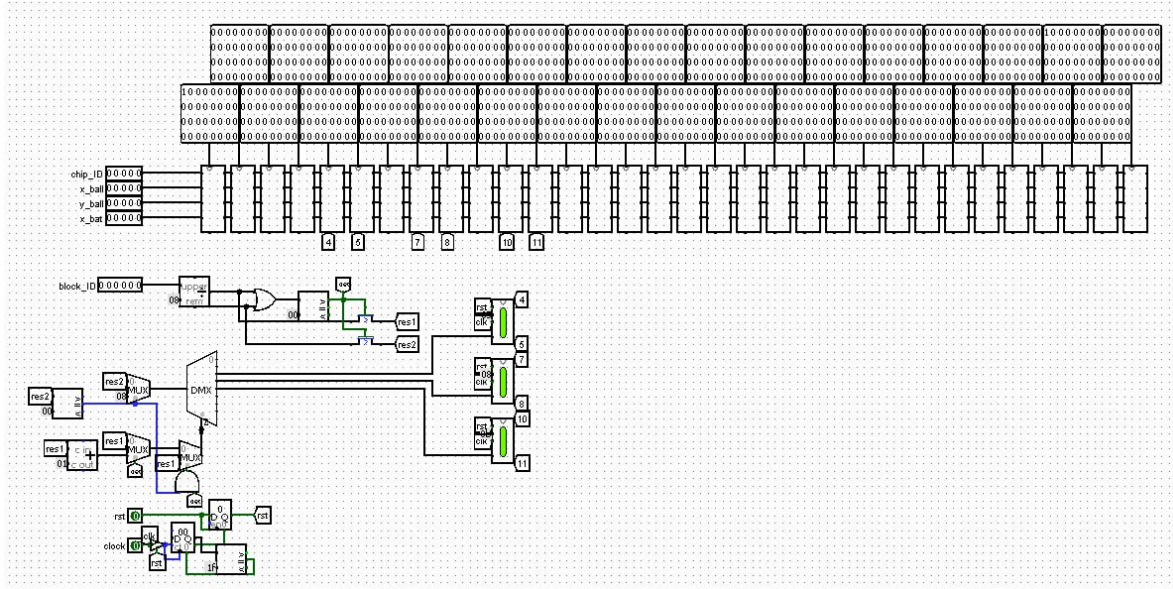


The circuit draws a line on the display. The input data includes the *chip_ID* from 0 to 31, the coordinates of the ball (*x_ball* and *y_ball*), the position of the right edge of the bat (*x_bat*), and information about blocks in the line.

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

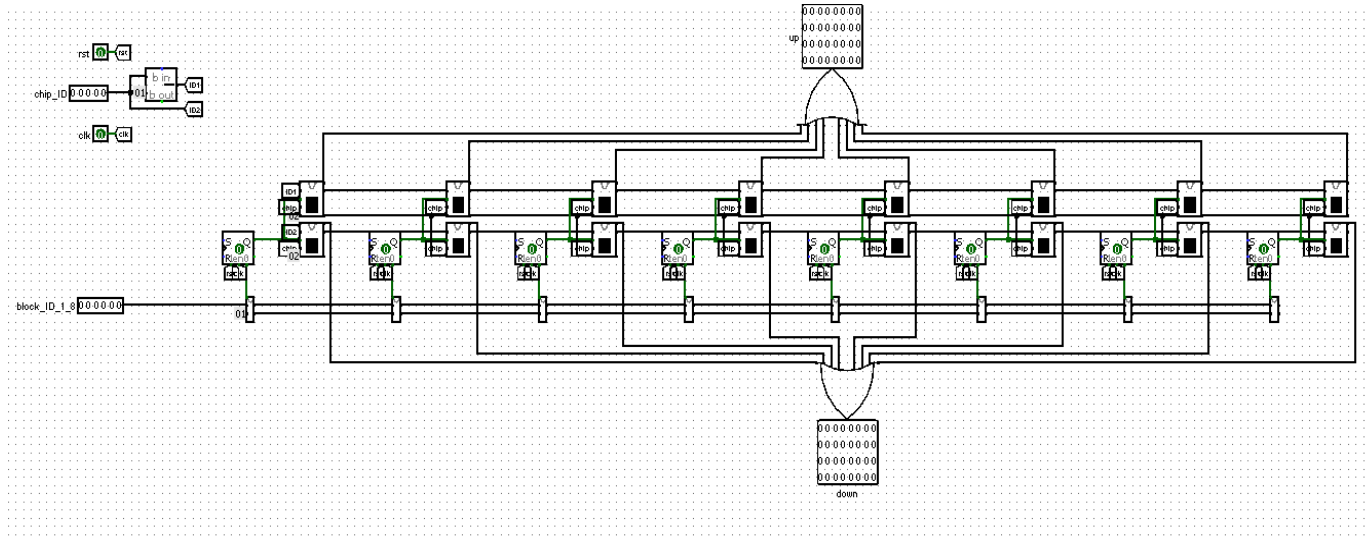


The circuit consists of 32 video chips and 3 detector block circuits. It establishes a connection between the video chips and the display, as well as converting the *block_ID* into a signal that removes the desired block from the display when the ball hits it.

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.

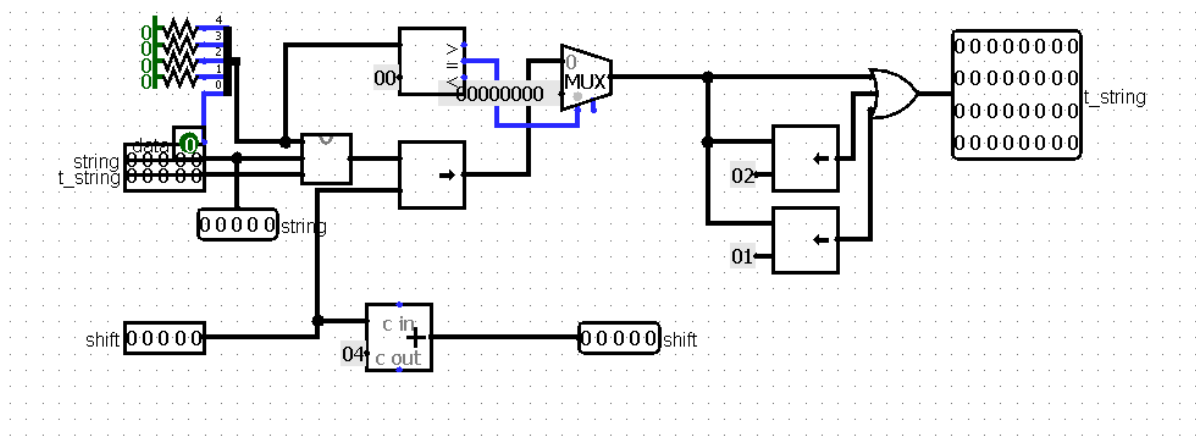
Input to the circuit includes the x-coordinate of the bat (*x_ball* and *y_ball*), *chip ID*, *block ID*, a *rst* signal (which will be described later), and a *clock* signal (provided by a clock generator).

1.2 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.

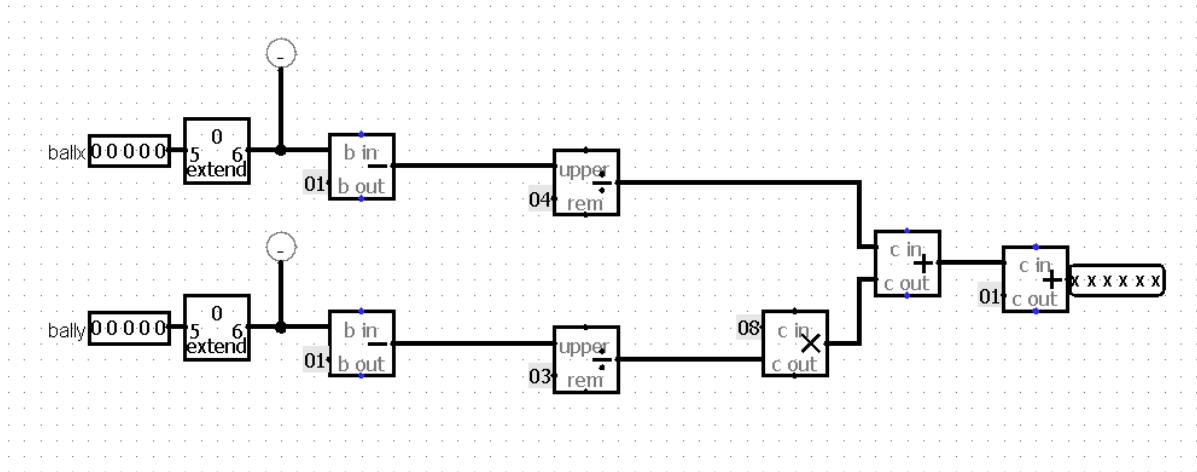
1.3 Block_paint



In this 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.

1.4 Block_ID calculation

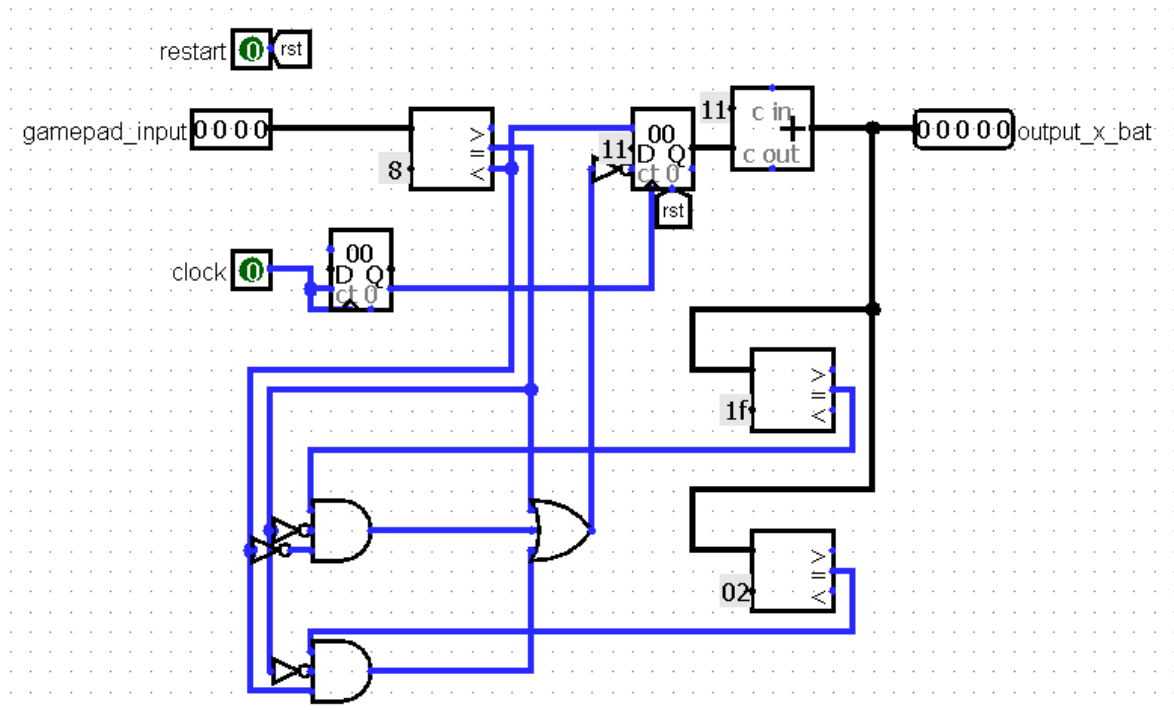
[OBJ]



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

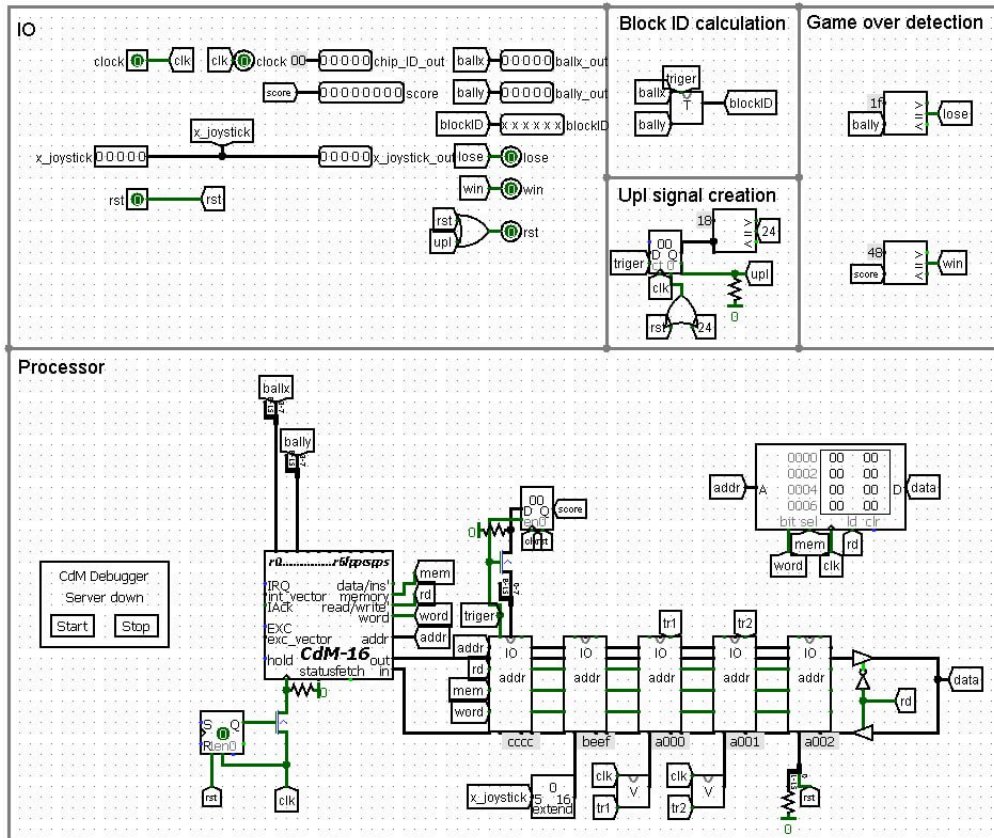
$$((bally - 1) // 3) * 8 + ((ballx - 1) // 4) + 1$$

1.5 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.

1.6 Kinematic_controller



In this 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.

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).

```
1702 # arrays of blocks, every array corresponds to logisim display string
1703 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
1704 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
1705
1706 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
1707 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
1708
1709 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
1710 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:

```

26  main>
27  #we use:
28  #r0 - x-coordinate
29  #r1 - x-velocity
30  #r2 - y-coordinate
31  #r3 - y-velocity
32  #r4,r5,r6 - for temporary calculations
33  ei
34  ldi r0,128 #x-coordinate
35  ldi r1,0xa000 #random x-velocity
36  ldb r1,r1
37  ldi r2,224 #y-coordinate
38  ldi r3,0xa001 #random y-velocity
39  ldb r3,r3
40  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).

```

41  while
42      ldi r6,248 #comparing y-coordinate
43      cmp r2,r6
44  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.

```

45  ldi r5,0xa002
46  ldb r5,r5
47  if
48      cmp r5,1 #checking for reset
49  is eq
50      ldi r6,0xcccc
51      ldi r5,0
52      st r6,r5
53      ldi r6,0xb001
54      stb r6,r5
55      ldi r6,0xb002
56      stb r6,r5
57  jsr restart # if reset signal is 1, jump to restart subroutine
58  fi

```

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

```

1590 restart: #subroutine that called if reset or level ends
1591     ldi r6,0×cccc
1592     ld r6,r6
1593     if
1594         ldi r5,24
1595         cmp r6,r5
1596     is eq
1597         ldi r5,0×b001 #setting the flag, that means that restart procedure happened to this score value
1598         ld r5,r4
1599         inc r4
1600         stb r5,r4
1601     fi
1602     if
1603         ldi r5,48
1604         cmp r6,r5
1605     is eq
1606         ldi r5,0×b002
1607         ld r5,r4
1608         inc r4
1609         stb r5,r4
1610     fi
1611     if
1612         ldi r5,72
1613         cmp r6,r5 #end of game
1614     is eq
1615         halt
1616     fi
1617

```

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.

```

1618     ldi r6,0
1619     ldi r5,bricks4
1620     ldi r4, bricks5 #rewriting the arrays of blocks
1621     while
1622         cmp r6,8
1623     stays lt
1624         ldi r3,0
1625         stb r5,r3
1626         stb r4,r3
1627         ldi r3,1
1628         add r5,2
1629         add r4,2
1630         stb r5,r3
1631         stb r4,r3
1632         add r5,2
1633         add r4,2
1634         stb r5,r3
1635         stb r4,r3
1636         add r5,2
1637         add r4,2
1638         stb r5,r3
1639         stb r4,r3
1640         add r5,2
1641         add r4,2
1642         inc r6
1643     wend

```

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

```
1697      wend
1698      reset #reseting the programm
1699      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.

```
86      add r1,r0,r4 #we use r4 register to pre-calculate x|coordinate for correct display of the ball
87      #checking wall collisions by x-coordinate
88      if
89          ldi r6, 255
90          cmp r4,r6
91      is gt
92          neg r1
93          add r1,r4,r4
94          ldi r6,0xa000 # loading random Vx
95          ldb r6,r6
96          if
97              cmp r1,0
98          is lt
99              neg r6
100             move r6,r1
101         else
102             move r6,r1
103         fi
104     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.

```

134 ldi r5, bricks11 #loading address of necessary array
135 move r0,r4 #copying x-coordinate
136 shr r4 #taking 5 most significant bits from 8-bit value of x-coordinate
137 shr r4
138 shr r4
139 add r5,r4,r5 #adding x-coordinate to the address of array
140 add r5,r4,r5
141 push r1
142 ld r5,r1
143 if
144     cmp r1, 1 #checking whether a collision occurred
145 is eq # we need to know if this left or right pixel
146     push r5
147     ldi r6,0xcccc #updating the score, which is stored at 0xcccc address
148     ld r6,r5
149     inc r5
150     stb r6,r5
151     pop r5
152     add r5,2 # if we increase address by 1 x-coordinate, and value by that address equals 1
153     # we are adding 2 to r5 because we are working with 16-bit values
154     #it is left pixel, else it is right pixel
155
156     ld r5,r1

```

We are updating memory values and adjusting random access speed.

```

160 dec r1 # updating values in current row
161 sub r5,2
162 st r5,r1
163 add r5,2
164 st r5,r1
165 add r5,2
166 st r5,r1
167 ldi r5, bricks10 # updating values in adjacent row
168 add r5,r4,r5
169 add r5,r4,r5
170 st r5,r1
171 add r5,2
172 st r5,r1
173 add r5,2
174 st r5,r1
175 pop r1
176 neg r1
177 ldi r6,0xa000 #loading random Vx
178 ldb r6,r6
179 if
180     cmp r1,0
181 is lt
182     neg r6
183     move r6,r1
184 else
185     move r6,r1
186 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).

```

729     then
730         ldi r5,0xbeef
731         ld r5,r4 #now it has coordinate of bat
732         move r0,r6
733         shr r6
734         shr r6
735         shr r6 #taking 5 most significant bits of 8-bit x-coordinate
736         if
737             cmp r6,r4 #checking whether a collision occurred with right pixel of the bat
738         is eq
739             neg r3 # if ball hits edge pixels of bat, vx and vy velocities are being negated,
740                 # if ball hits central pixel - only vy
741             neg r1
742             ldi r6,0xa001 #loading random Vy
743             ldb r6,r6
744             if
745                 cmp r3,0
746             is lt
747                 neg r6
748                 move r6,r3
749             else
750                 move r6,r3
751             fi
752

```

```

753     else
754         dec r4 #checking whether a collision occurred with middle pixel of the bat
755         if
756             cmp r6,r4
757         is eq
758             neg r3
759             ldi r6,0xa001 #loading random Vy
760             ldb r6,r6
761             if
762                 cmp r3,0
763             is lt
764                 neg r6
765                 move r6,r3
766             else
767                 move r6,r3
768             fi
769

```

```

770     else
771         dec r4 #checking whether a collision occurred with left pixel of the bat
772         if
773             cmp r6,r4
774         is eq
775             neg r3
776             neg r1
777             ldi r6,0xa001 #loading random Vy
778             ldb r6,r6
779             if
780                 cmp r3,0
781             is lt
782                 neg r6
783                 move r6,r3
784             else
785                 move r6,r3
786             fi
787         else
788             fi
789         fi
790     fi
791

```


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.

```

815     ld r5,r1
816     if
817         cmp r1,1 #checking for collision
818     is eq
819         push r5
820         ldi r6,0×cccc #updating score
821         ld r6,r5
822         inc r5
823         stb r6,r5
824         pop r5
825         add r5,2 #moving one pixel forward, if it is zero, it means that ball hit the right pixel
826         ld r5,r1
827         if
828             cmp r1,1
829         is eq
830             sub r5,2 #moving back two pixels, if it is zero - it means that ball hit the left pixel
831             # else - middle pixel
832             sub r5,2
833         ld r5,r1

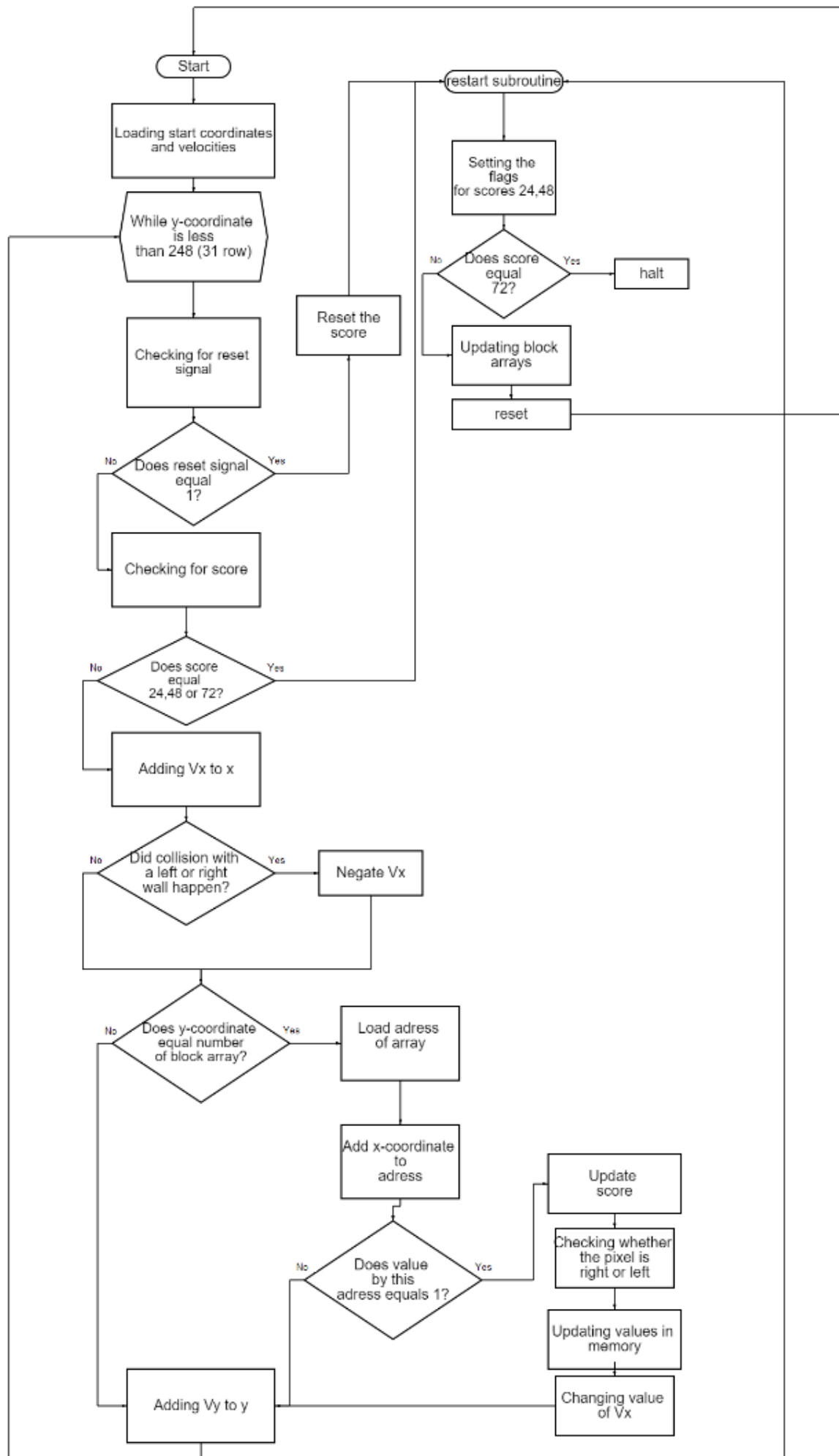
```

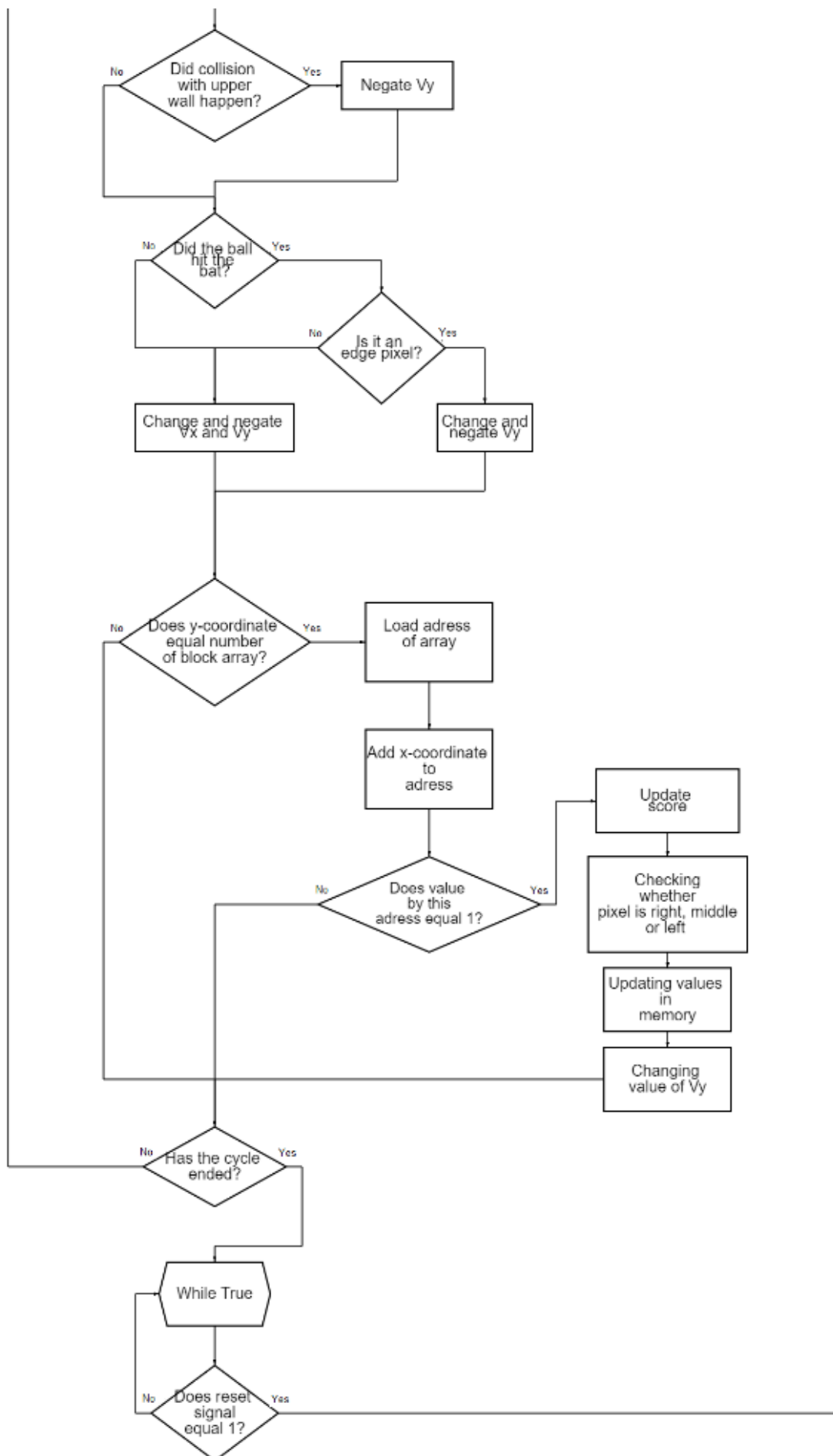
If the ball reaches the 31st row, the cycle ends. We then start a new "while true" loop and wait for a reset signal.

```

1577     ldi r6,2
1578     while
1579         cmp r6,2 #starting infinite loop
1580     stays eq
1581         ldi r5,0×a002
1582         ldb r5,r5
1583         if
1584             cmp r5,1
1585         is eq
1586             ldi r6,0×cccc # if reset signal equals 1 - jump to restart subroutine
1587             ldi r5,0
1588             st r6,r5
1589             ldi r6,0×b001
1590             stb r6,r5
1591             ldi r6,0×b002
1592             stb r6,r5
1593             jsr restart
1594         fi
1595     wend
1596     halt
1597

```





3 Interaction between software and hardware

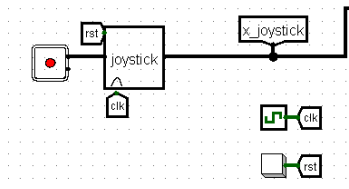
From Hardware:

- Random values of ball speed modules V_x , V_y
- *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.