

Enhanced Pong

Digital platforms project B

Merzlyakov Ilya, Batenko Nikita, Feshchenko Igor

Contents

1	Overview	2
2	Hardware design	3
2.1	Video system	3
2.1.1	Video_chip	4
2.1.2	Video_section	4
2.1.3	Video_player	5
2.2	AI	6
2.3	Kinematic controller	7
2.3.1	Ball position calculation	8
2.4	Ball velocity calculation	9
2.5	Miscellaneous circuitry	10
3	Cdm-8 software	11
4	Software	16
4.1	generate_firmwares.py script	16
4.2	AutoRAM Logisim library	16

1 Overview

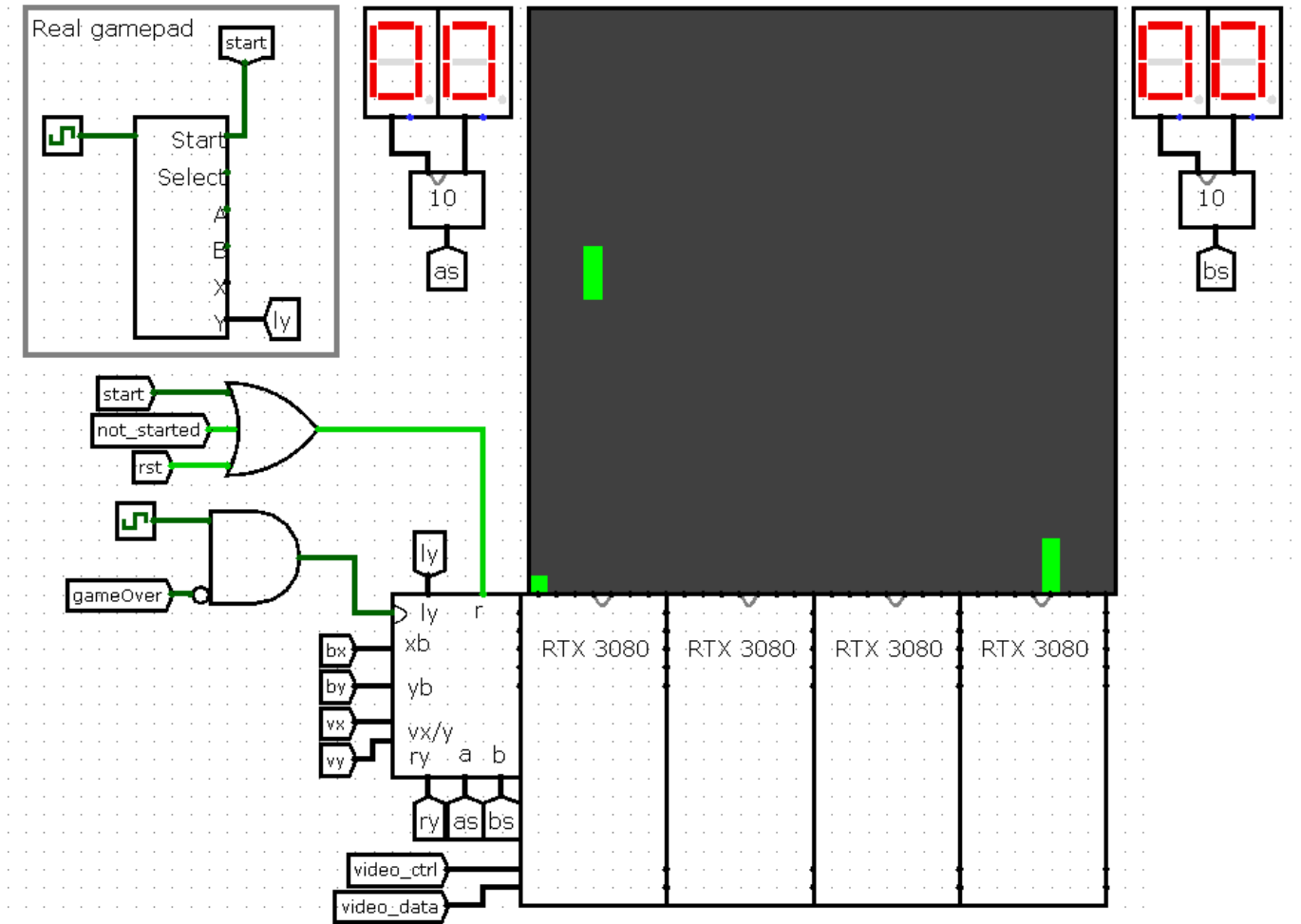


Figure 1: Game display

We implemented a fully functional Pong (or TV-Tennis) game, based on design document, but with some extensions:

- ball now can fly at 512 different angles (compared to 4 from original specification)
- when one player gets a point, the ball respawns at the side of hit player directed to other player
- game now is not endless, it ends after one player reaches 5 points
- animations: they play on system start and after game over
- real gamepad is used, logisim library was written in Kotlin to support it
- ball velocity is slightly changed in the middle of the screen, so AI will miss sometimes
- ball direction changes based on which part of the bat it was reflected by

2 Hardware design

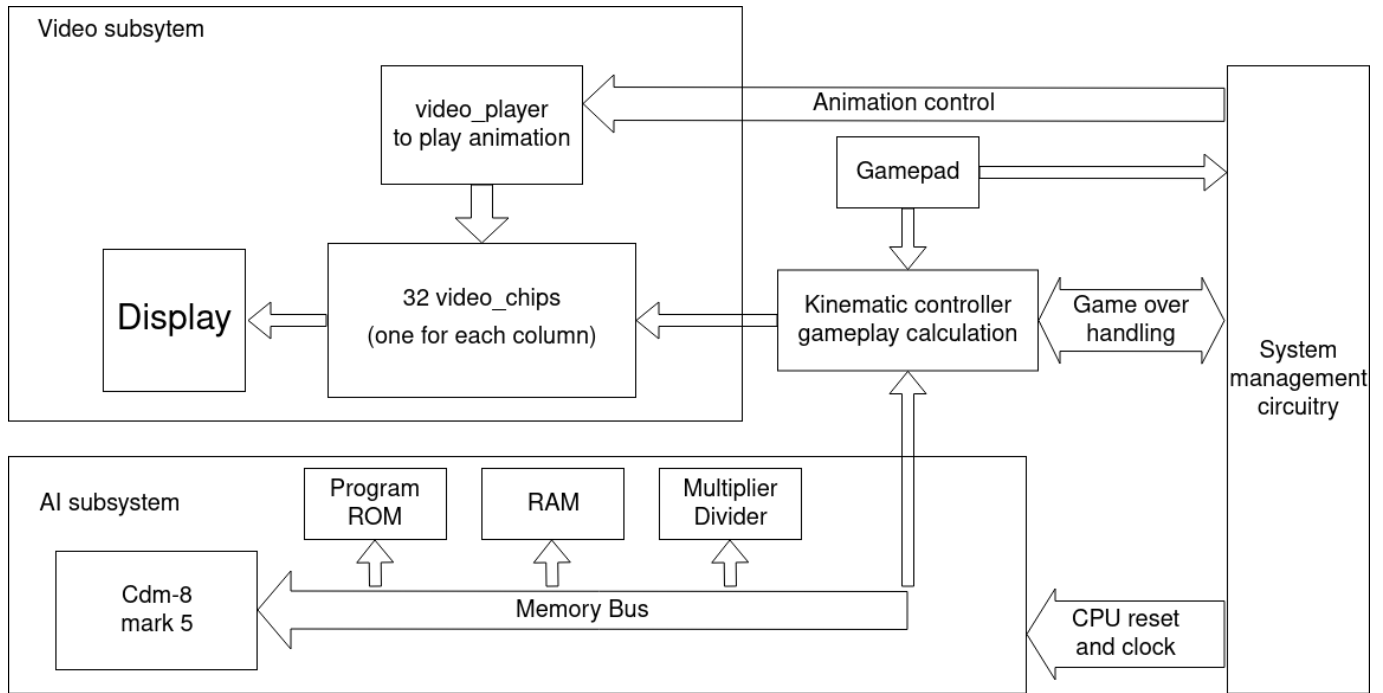


Figure 2: Hardware architecture

Or project's circuit can be divided into three subsystems:

- video system - draws gameplay elements (ball and bats) and plays animations;
- game mechanics system - responsible for actual gameplay;
- ai system - controls bot movement.

Also there is *main* chip, which glues all this systems together and also contains graphical display. Now all said systems will be described.

2.1 Video system

This system is responsible for rendering graphics on 32x32 display. It can operate in two modes: *gameplay* mode and *video* mode. These mode are selected with **enable** pin of *video_player* chip. In the first mode, ball and bats are visible on the screen; in the second mode *video_player* can draw arbitrary images on screen. This subsection consists of the following chips: *video_chip*, *video_section* and *video_player*.

2.1.1 Video_chip

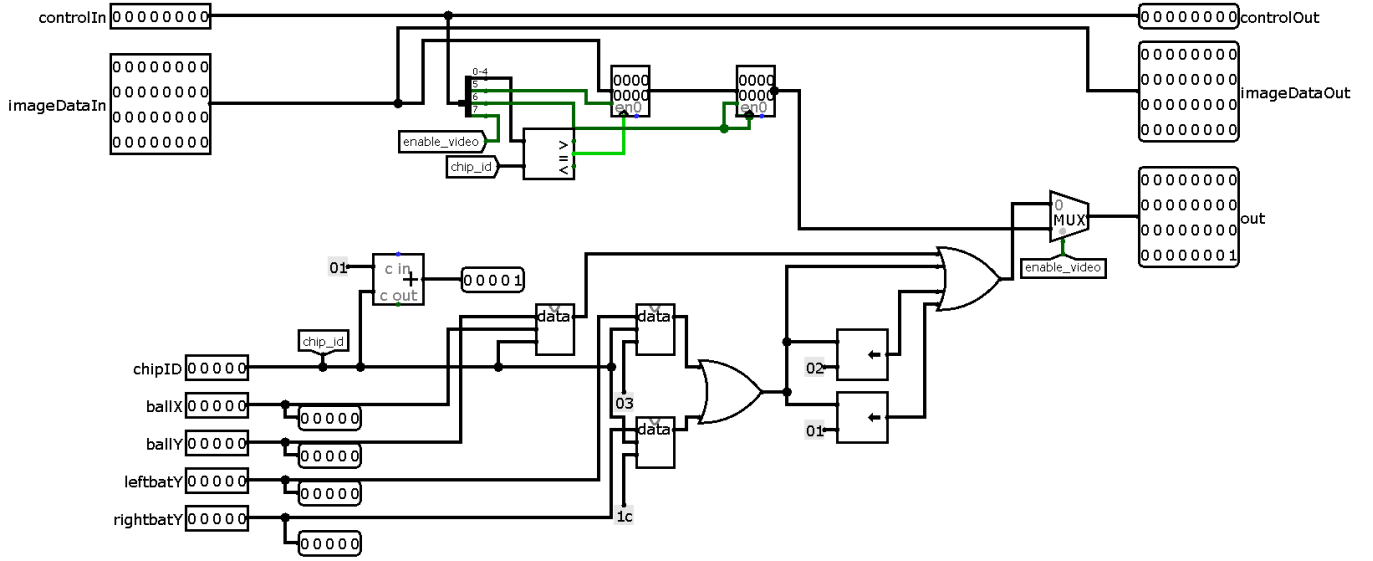


Figure 3: Video_chip

This chip controls one column of the screen; to control one display we need to connect 32 of these chips in a row. Because this chip is intended to be connected to other *video_chips*, most of its inputs are wired directly to corresponding outputs (except **chipID**). **chipID** actually defines column number the chip is connected to, so **chipID out** pin is set to **chipID input**+1. In *gameplay* mode this chip draws ball at specified height if **ballX** equals to **chipID**. Also it draws bat if **chipID** equals to 3 (left bat) or 28 (right bat). **leftbatY** and **rightbatY** define coordinate of the bottommost pixel of corresponding bat. The *video* mode is enabled by bit 7 of **controlIn** and will be described later.

2.1.2 Video_section

This is just an assembly of 8 *video_chips* connected one after another.

2.1.3 Video_player

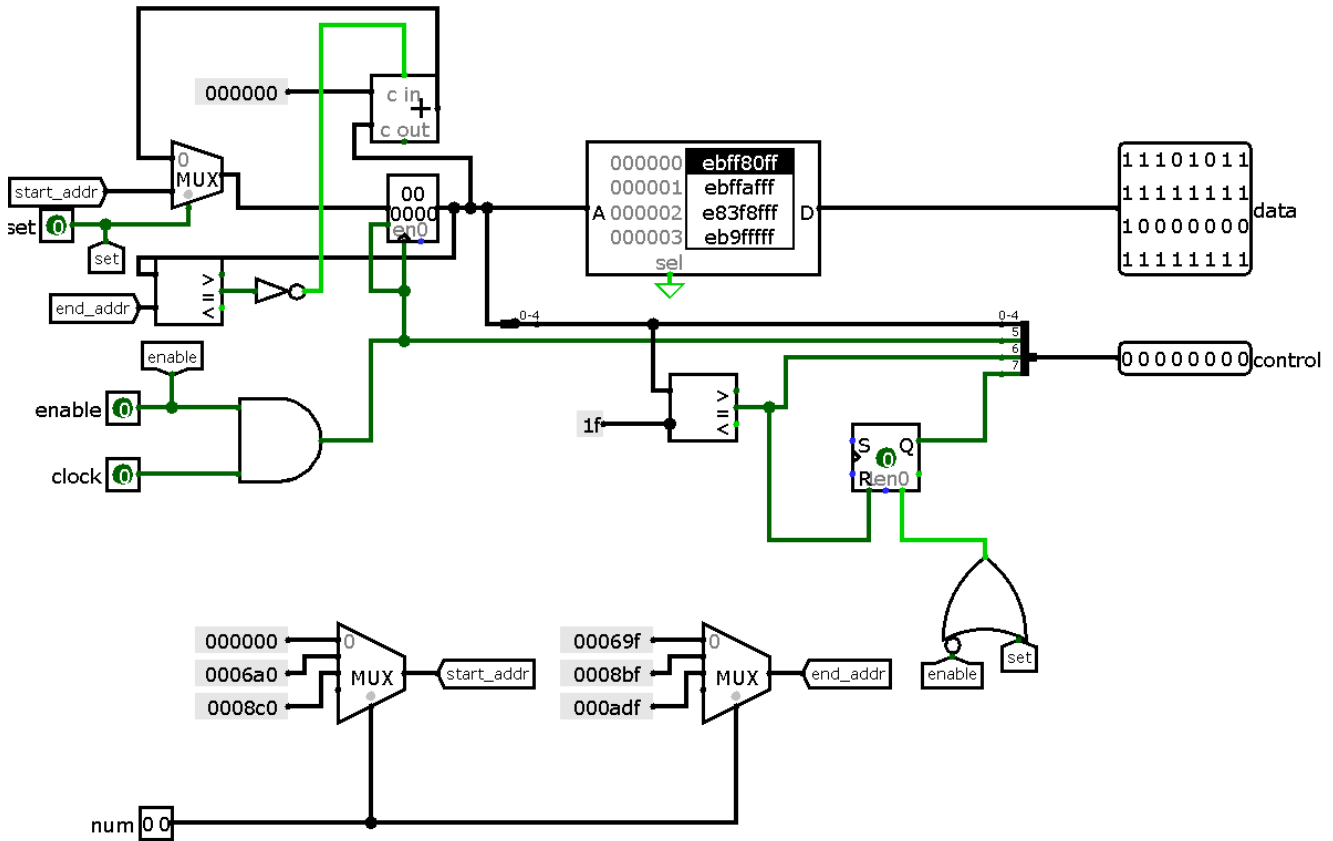


Figure 4: Video_player chip

Input pins:

- **enable** - If 1, video mode is active
- **num** - Animation number
- **set** - When 1, starts animation with specified number
- **clock**

Output pins:

- **ctrl** - *video_chip* control
- **data** - Current pixel column

This chip has ROM with all animations. One animation is simply a sequence of frames, and one frame is a sequence of 32 32-bit values (columns). To draw one frame of animation, this chip must send all columns of this frames to corresponding *video_chips*. To do so, it firstly selects needed chip with bits 0-4 of **ctrl**. Then it sets **data** to current column of the image, and by rising bit 5 of **ctrl** it tells selected *video_chip* to store column data in intermediate register. When the whole frame is stored in these registers, it raises bit 6 of **ctrl**, so all *video_chips* update their outputs simultaneously. Bit 7 of **ctrl** switches *video_chips* into *video* mode.

Constants below the chip define start and end addresses of all animations.

2.2 AI

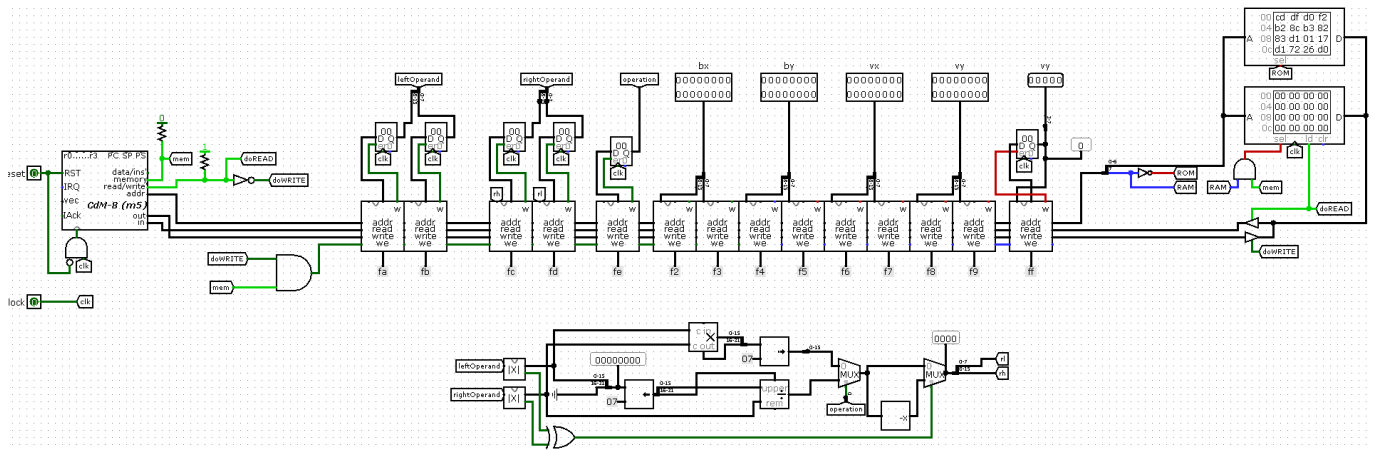


Figure 5: AI chip

Input pins:

- **bx** - X coordinate of ball
- **by** - Y coordinate of ball
- **vx** - X component of velocity
- **vy** - Y component of velocity
- **reset** - Cdm-8 reset
- **clock**

Output pins:

- **result** - Calculated bat position

To predict ball position, Cdm-8 must perform multiplication and division. Since ball coordinates and velocity are represented as 16-bit fixed-point numbers (fraction part stored in bits 0-6), Cdm-8 is not powerful enough to perform multiplication and division, so hardware multiplier and divider are mapped to it's memory to speed up calculations.

Cdm-8 here is used in Von Neumann mode (code and data are in the same address space), but it reads program from ROM mapped to a lower half of address space, and uses RAM mapped to a higher half of address space, so there is no need for loading firmware into RAM on every system start.

2.3 Kinematic controller

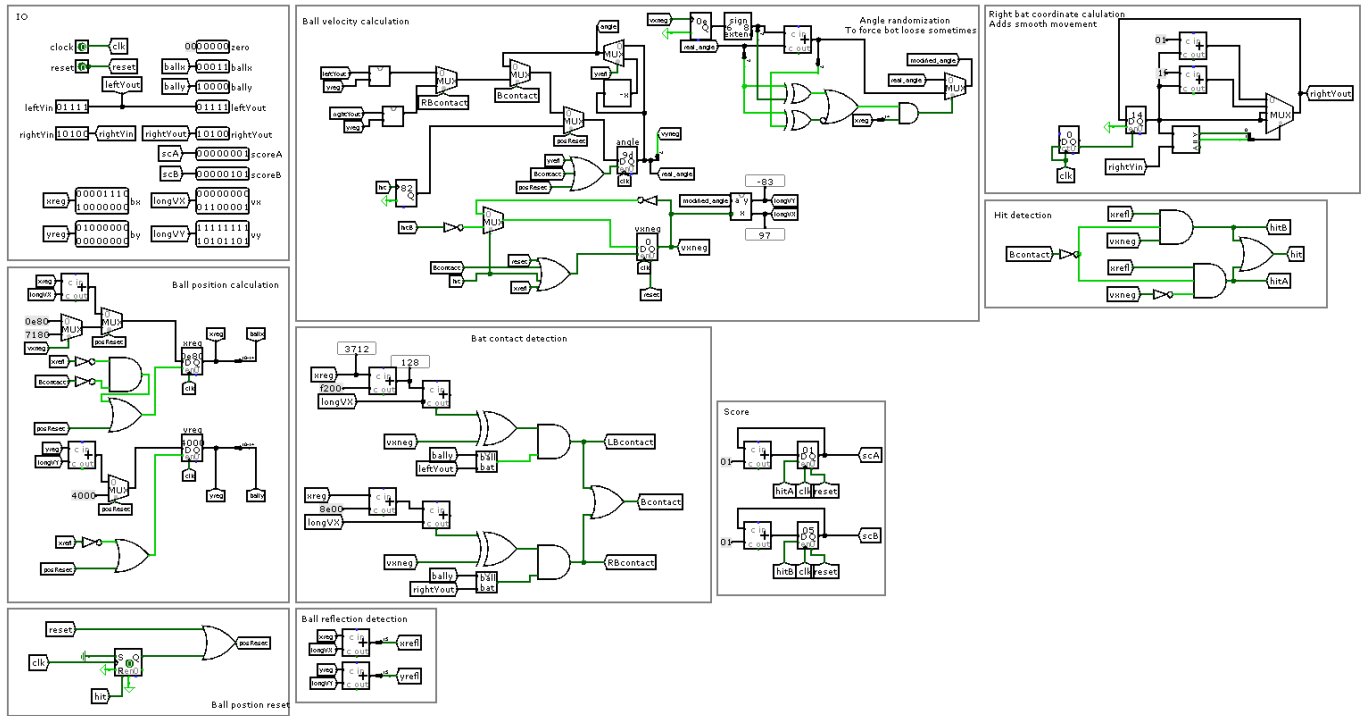


Figure 6: Kinematic controller overview

Input pins:

- **reset** - game restart
- **ly** - left bat position
- **ry** - right bat position

Output pins:

- **xb** - 16-bit X coordinate of ball
- **yb** - 16-bit Y coordinate of ball
- **vx** - 16-bit X component of ball velocity
- **vy** - 16-bit Y component of ball velocity
- **a** - left player score
- **b** - right player score
- **five pins on east side** - output to *video_chips*

This chip is responsible for all gameplay mechanics.

Let's define some internal terminology for later use in this subsection:

- **hit** - event when ball touches vertical edge of the screen
- **posReset** - event triggered on reset and hit

- bx , by - ball coordinates
- vx , vy - ball velocity

Ball coordinates and velocity are represented as 16-bit fixed-point numbers (fraction part stored in bits 0-6).

Every clock cycle ball coordinates are updated by corresponding velocity components, then velocity is updated if ball collision occurred. Also it counts players' score and adds smoothness to right bat's movement (since AI sets it's position only once while the ball is moving towards it). Bat collision is detected by comparison of x coordinate of ball and bar, and wall collision detection is detected when addition of velocity component to ball coordinate cause overflow.

Now some parts of kinematic controller will be described with more detail.

2.3.1 Ball position calculation

This part of *kinematic* controller is responsible for storing and updating (on every clock cycle) ball coordinates. If there was no hit, it calculates $bx \leftarrow bx + vx$ and $by \leftarrow by + vy$. If there was a hit, it resets ball position and places the ball on side of the screen so the ball will fly through the whole screen to player who get the point at this tick (side is selected based on the ball direction).

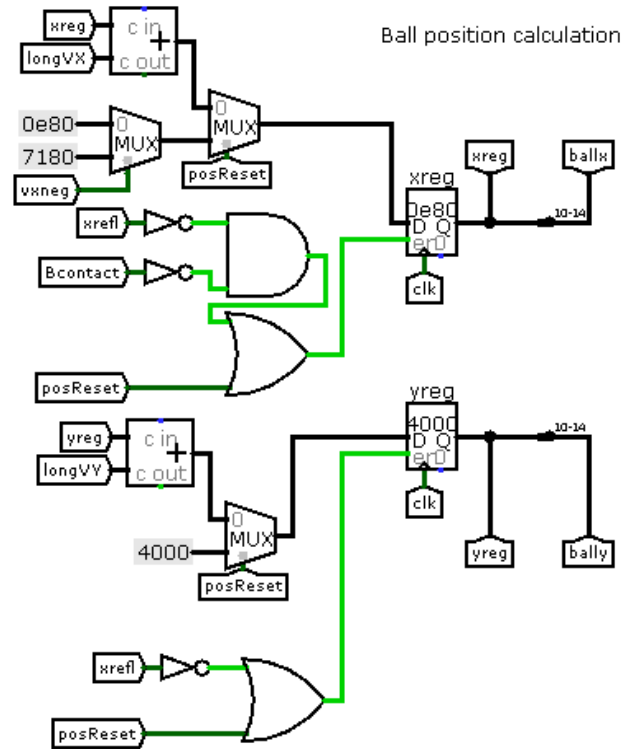


Figure 7: Ball position calculation unit

2.4 Ball velocity calculation

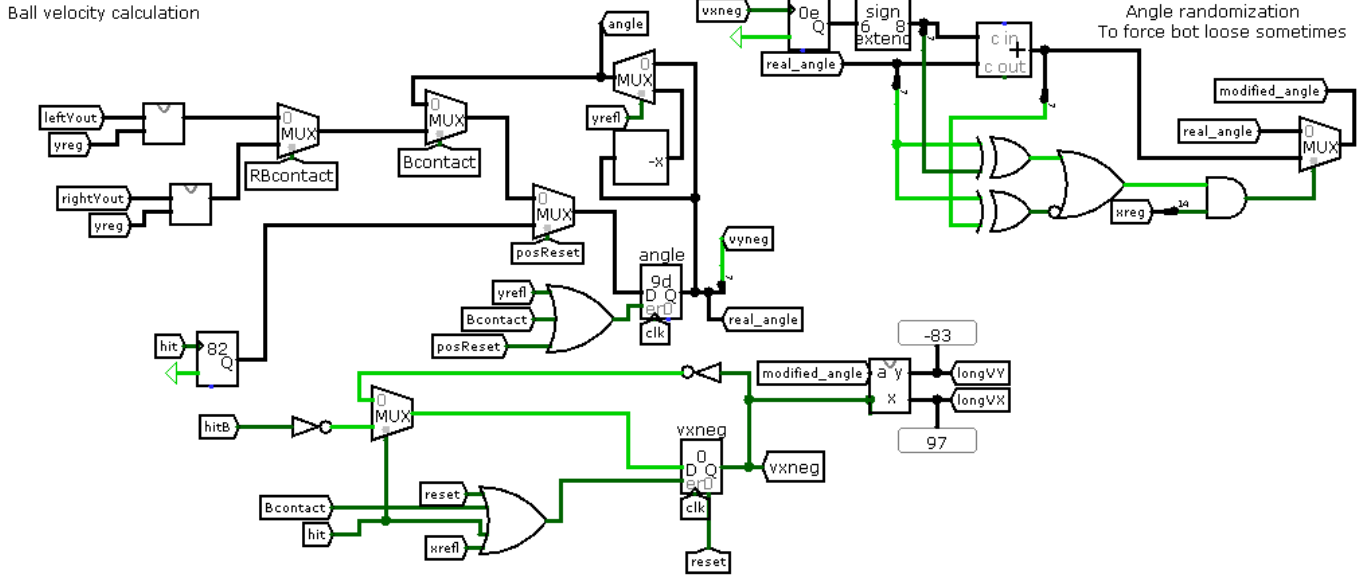


Figure 8: Ball velocity calculation unit

This part is responsible for storage and updation of ball velocity. It separately stores ball angle (a value from $[-52^\circ; +52^\circ]$ mapped to $[-127; 127]$) and ball direction along X axis (in vxneg register, 1 - negative direction, 0 - positive direction). If posReset happens, it sets angle to random value, and sets X direction so the ball will fly towards the player who got point on this tick. If the ball collides with the bat, it changes X direction to opposite. When ball hits horizontal edge of the screen, it just negates it's angle, but when the ball is reflected by bat, it sets angle to $by - bat$, so, for example, if the ball was reflected by top side of the bat, it will fly upwards.

Also this unit slightly changes angle by random value after the ball travels half the distance from left side to right side of the screen, so AI sometimes will miss.

Vx and vy are calculated as $\cos angle$ and $\sin angle$ respectively by *velocity_calculator* chip. It just reads this values from pregenerated ROM chips.

52° is the maximum angle that does not cause overflow when calculating $(228 - bx) * \tan angle$

2.5 Miscellaneous circuitry

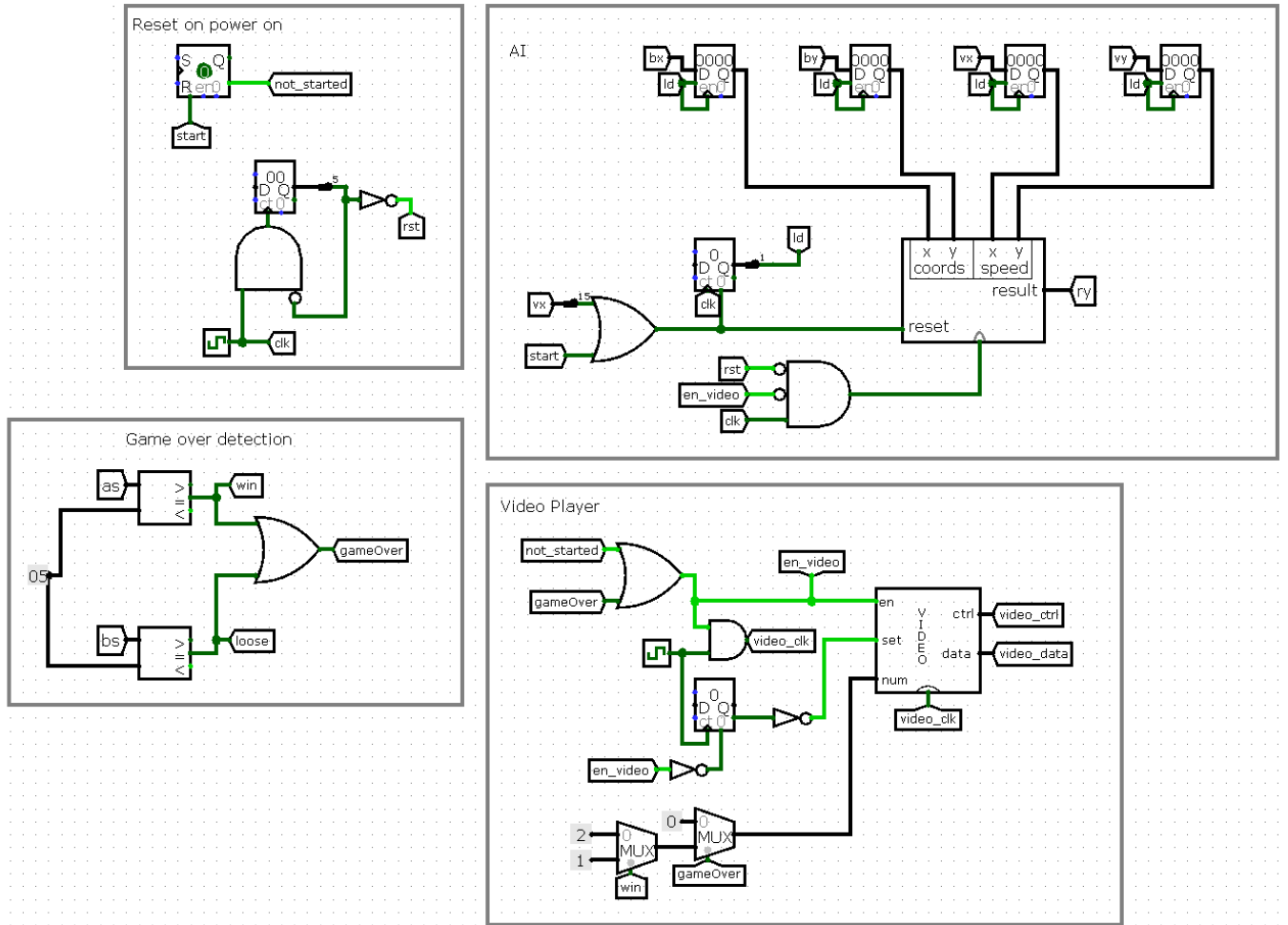


Figure 9: Miscelanious circuitry

AI chip is placed in the top right corner of the picture. Note that *bx*, *by*, *vx*, *vy* are connected to it through registers. Their values are updated only when ball starts flying from left to right. These registers serve two purposes: first, the prevent ai from cheating by setting bat coordinate to ball coordinate in loop, an second, they prevent data corruption, since Cdm-8 cannot read input values instantly, and these values can change in the middle of reading.

Video_chip is placed in the bottom right corner. It is connected to animation selection circuitry. Animation numbers: 0 - startup, 1 - win, 2 - loose.

3 Cdm-8 software

To predict ball position when it's x coordinate will be equal to 228 (coordinate of right bat), Cdm-8 must calculate following formula: $result = by + \frac{(228 - bx) * vy}{vx}$. Here is complete and well-commented source code for Cdm-8. It's flowchart is attached below.

```
1 # This file contains AI source code for Pong game
2 #
3 # Memory map:
4 # 0x00 – 0x7f: program ROM
5 # 0x80 – 0xf1: RAM
6 # 0xf2 – 0xff: IO devices described below
7 #
8 # Cdm-8 is connected to hardware multiplier/divider
9 #
10 # Readable IO devices (two addresses => 16-bit device):
11 # 0xf2, 0xf3: x coordinate of the ball
12 # 0xf4, 0xf5: y coordinate of the ball
13 # 0xf6, 0xf7: x velocity of the ball
14 # 0xf8, 0xf9: y velocity of the ball
15 # 0xfc, 0xfd: result of multiplication/division
16 #
17 # Writable IO devices
18 # 0xfa, 0xfb: first operand for multiplier/divider
19 # 0xfc, 0xfd: second operand for multiplier/divider
20 # 0xfe      : operation (0 – multiplication, 1 – division)
21 # 0xff      : bat position
22
23 asect 0x00
24 # 0xCD – undocumented instruction that sets SP to it's operand
25 dc 0xcd, 0xdf
26
27
28
29 # main logic goes here
30 # it computes the following formula:  $by + (228 - bx) * vy / vx$ 
31 # 228 = x coordinate of our bat
32 # all numbers are 16-bit fixed-point big-endian
33 # (fractional part is stored in 7 least significant bits)
34 start:
35 ldi r0, bx           # load x coordinate of the ball into r2 and r3
36 ld r0, r2
37 inc r0
38 ld r0, r3
39 not r2               # now we have -bx-1 in registers
40 not r3
41
42
43 ldi r1, 0x01        # add 228 + 1
44 add r1, r3
```

```

45 ldi r1, 0x72
46 addc r1, r2
47 # now we have 228 - bx in r2 and r3
48
49
50 # now multiply by vy
51 # since cdm-8 isn't powerful enough,
52 # we use hardware multiplier and divisor
53 # op - operation code
54 # op1 - first operand
55 # op2 - second operand when writing, result when reading
56 ldi r0, op1          # set operand 1 to 228-bx
57 st r0, r2
58 inc r0
59 st r0, r3
60
61 # set operand 2 to vy
62 ldi r0, vy
63 ldi r1, op2
64 jsr mvnum
65
66 # set operation to multiplication (operation 0)
67 ldi r0, 0
68 ldi r1, op
69 st r1, r0
70
71 # we cannot just copy result to operand 1
72 # because when first byte of result is written to operand,
73 # result will immediately change
74 # so we store the product in memory
75 # TODO: rewrite it using registers only
76 ldi r0, op2
77 ldi r1, 0xc0
78 jsr mvnum
79
80 # now divide by vx
81 ldi r0, 0xc0          # set operand 1 to (228-bx)*vy
82 ldi r1, op1
83 jsr mvnum
84
85 ldi r0, vx            # set operand 2 to vx
86 ldi r1, op2
87 jsr mvnum
88
89 ldi r0, 1             # set operation to division (operation 1)
90 ldi r1, op
91 st r1, r0
92
93
94

```

```

95 # load y coordinate of the ball into r2, r3
96 ldi r0, by
97 ld r0, r2
98 inc r0
99 ld r0, r3
100
101 # load (228-bx)*vy/vx into r0, r1
102 ldi r1, op2
103 ld r1, r0
104 inc r1
105 ld r1, r1
106
107 # add them and store result in r2, r3
108 add r1, r3
109 addc r0, r2
110
111 # now we only need contents of r2 since bat position are 5 bit
112 # if bit 7 of r2 is set,
113 # the ball was reflected either from top or from bottom
114 tst r2
115 bpl ready
116 # if reflection occurred, we need to know was it top or bottom
117 # the following block of code handles it, but i cannot remember how
118 # TODO: understand my code
119 ldi r0, vy
120 ld r0, r0
121 bmi low_negate
122 # high_add_128
123 ldi r0, 127
124 add r0, r2
125 br ready
126 low_negate:
127 neg r2
128
129 # now we have our answer — bat coordinate
130 ready:
131 shl r2 # it was shifted right
132 ldi r0, 0b11111000 # we need only 5 high bits
133 and r0, r2
134
135 # in kinematic controller, bat coordinate represents it's lowest pixel
136 # but we want to reflect ball with central pixel
137 # so we decrease r2 if it is not zero
138 tst r2 # probably unnecessary tst
139 bz do_write
140 ldi r0, 0b00001000
141 sub r2, r0
142 move r0, r2
143
144 # finally move bat

```

```

145 do_write:
146 ldi r0, bat
147 st r0, r2
148
149 halt
150
151 # this subroutine moves 16-bit numbers
152 # prameters:
153 # r0 — source addres
154 # r1 — destination address
155 mvnum:
156 ld r0, r2
157 st r1, r2
158 inc r0
159 inc r1
160 ld r0, r2
161 st r1, r2
162 rts
163
164
165
166 # here defined addresses of io devices
167 # dc directives are for debugging
168 asect 0xf2
169 bx: dc 0x19, 0x80
170
171 asect 0xf4
172 by:
173
174 asect 0xf6
175 vx: dc 0x13,0x37
176
177 asect 0xf8
178 vy:
179
180 asect 0xfa
181 op1:
182
183 asect 0xfc
184 op2:
185
186 asect 0xfe
187 op:
188
189 asect 0xff
190 bat:
191
192 # This is the
193 end
194 # my only friend

```

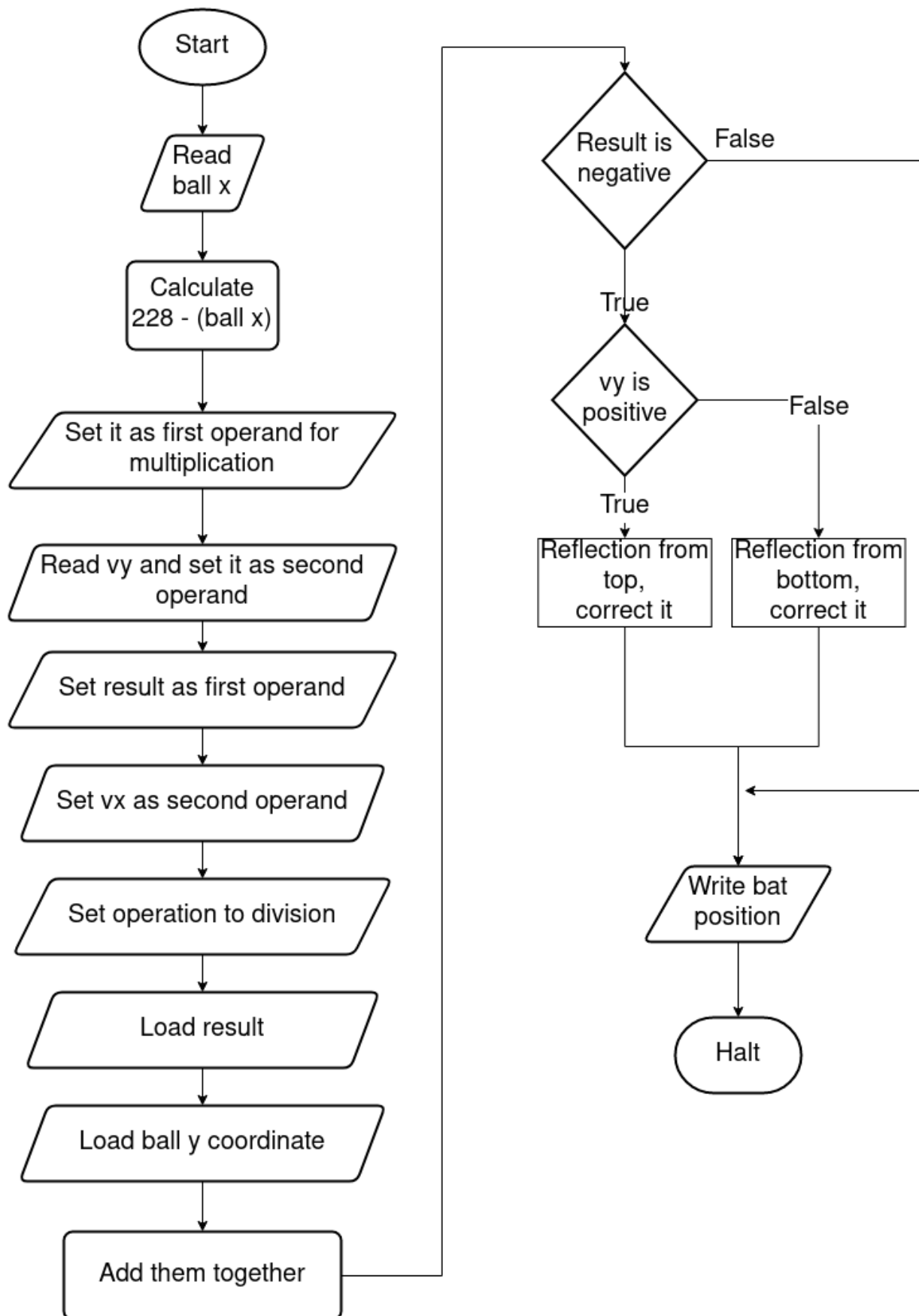


Figure 10: Algorithm flowchart

4 Software

To make development process easier, and to add more features to the project, some software was developed.

4.1 `generate_firmwares.py` script

This script was written to automatically generate 4 binary images:

- `ai.img` - compiled Cdm-8 program, this script uses `cocas.py` from CocoIDE distribution to do it.
- `animation.img` - all animations generated by this script. Used in *video_player*
- `cos.img` and `sin.img` - tables for *velocity_calculator* chip

This script is pretty well commented and will not be described here.

4.2 AutoRAM Logisim library

It is named AutoRAM due to historical reasons (it was used to automatically load Cdm-8 firmware to RAM on simulation reset). Now it is used to access real gamepad connected to host logisim runs on. This library is written in kotlin, it's sources are available at github: https://github.com/leadpogrommer/logisim_convinient_ram (yes, I know about a typo in repository name)