Enhanced Pong
Digital platforms project B

Merzlyakov Ilya, Batenko Nikita, Feshchenko Igor

Novosibirsk State University
2021

Contents

1 Overview

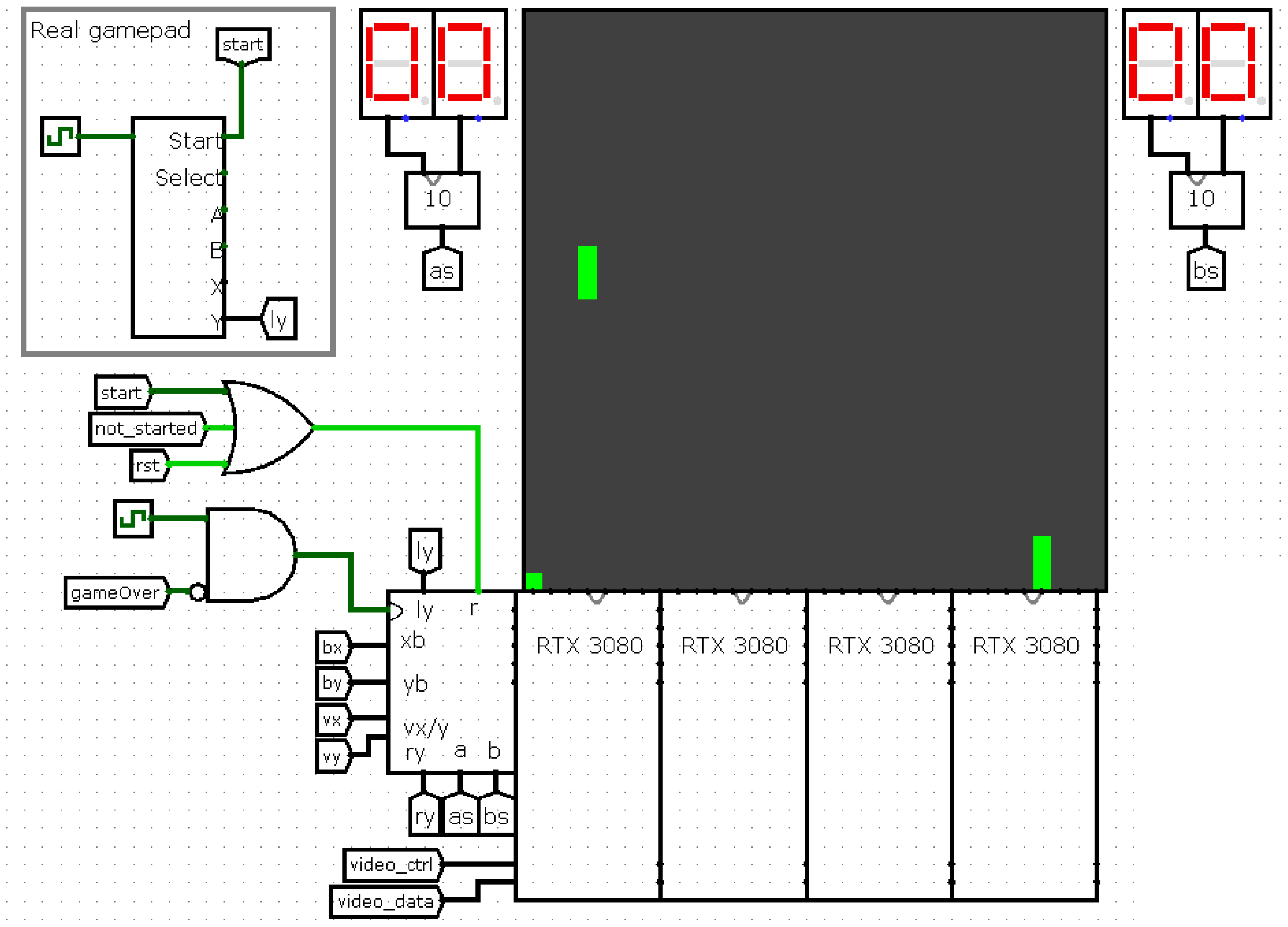

Figure 1: Game display

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

- now, ball 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 is not endless now, it ends when one player gets 5 points
- animations: they play on system start and after game over
- real gamepad is used, to support it logisim library was written in Kotlin
- 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
- ball velocity always equals to 1

2 Hardware design

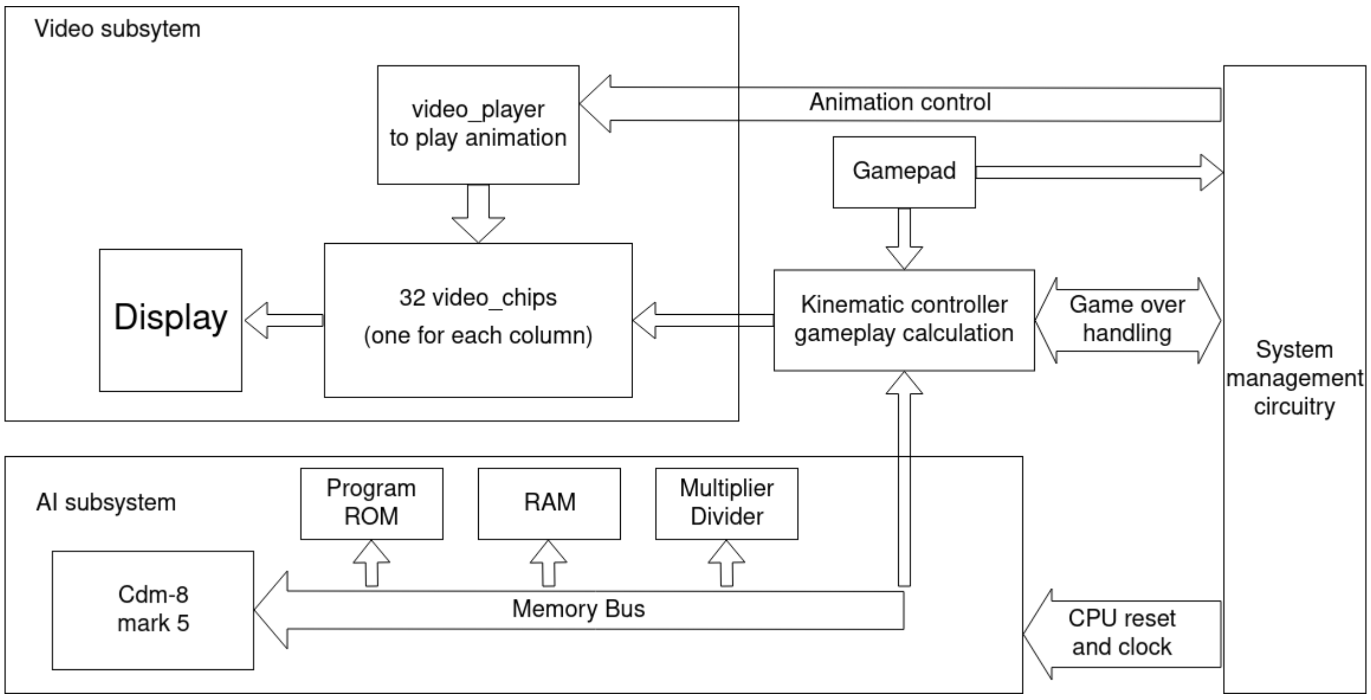

Figure 2: Hardware architecture

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

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

Also there is *main* chip, which glues all these 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

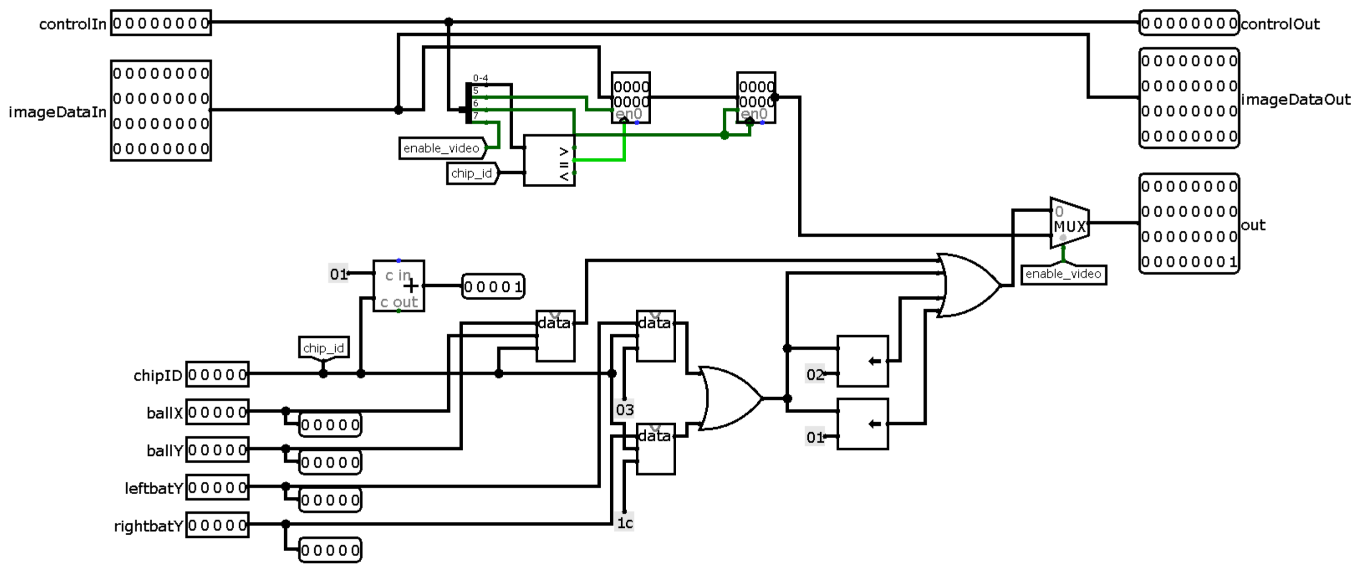

Figure 3: Video_chip

This chip controls one column of the screen; to control display we need to connect 32 of these chips in a row. Because this chip is intended to be connected to other *video_chip*s, most of it's 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 Y 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 chip

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

2.1.3 Video_player

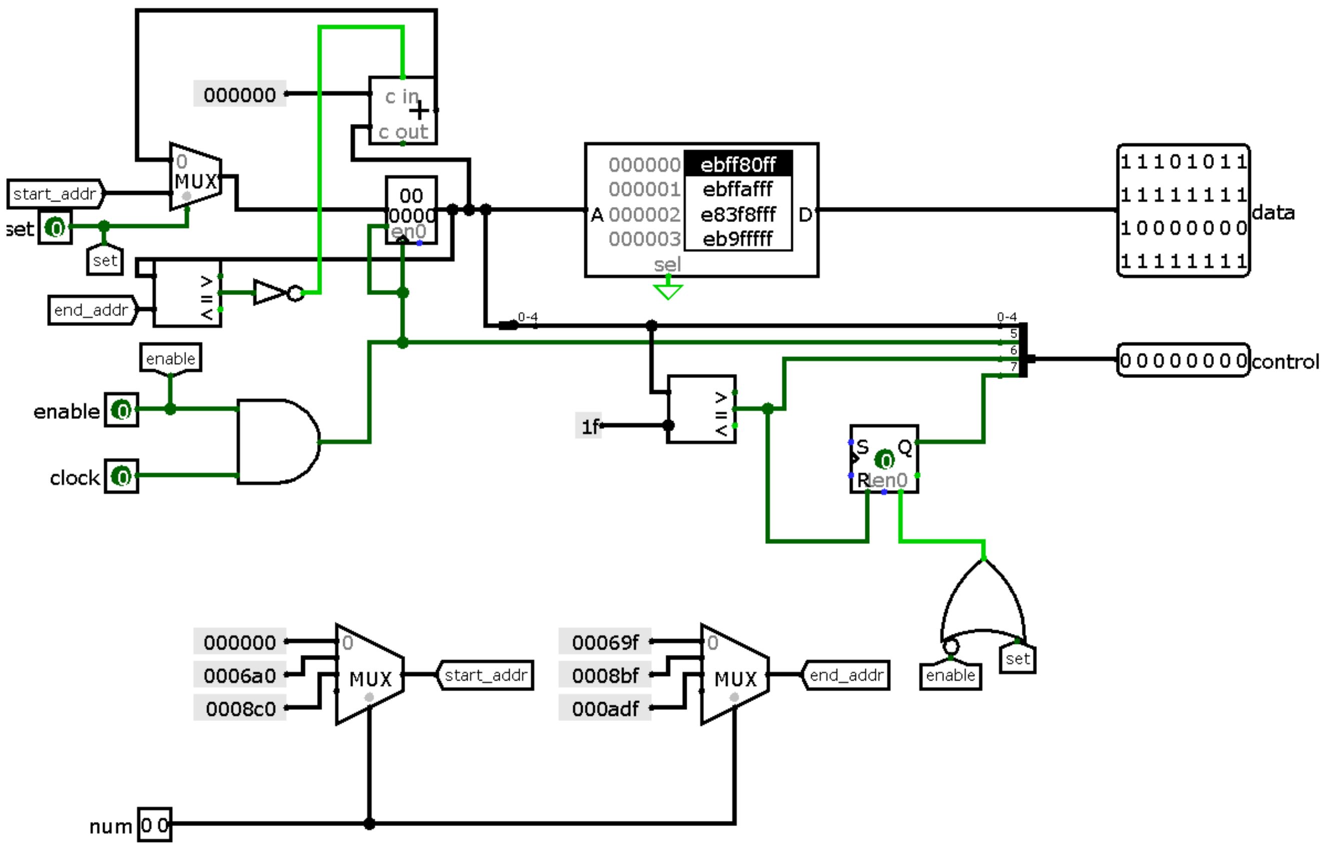

Figure 4: Video_player chip

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_chip*s. 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_chip*s update their outputs simultaneously. Bit 7 of **ctrl** switches *video_chip*s into *video* mode.

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

2.2 AI

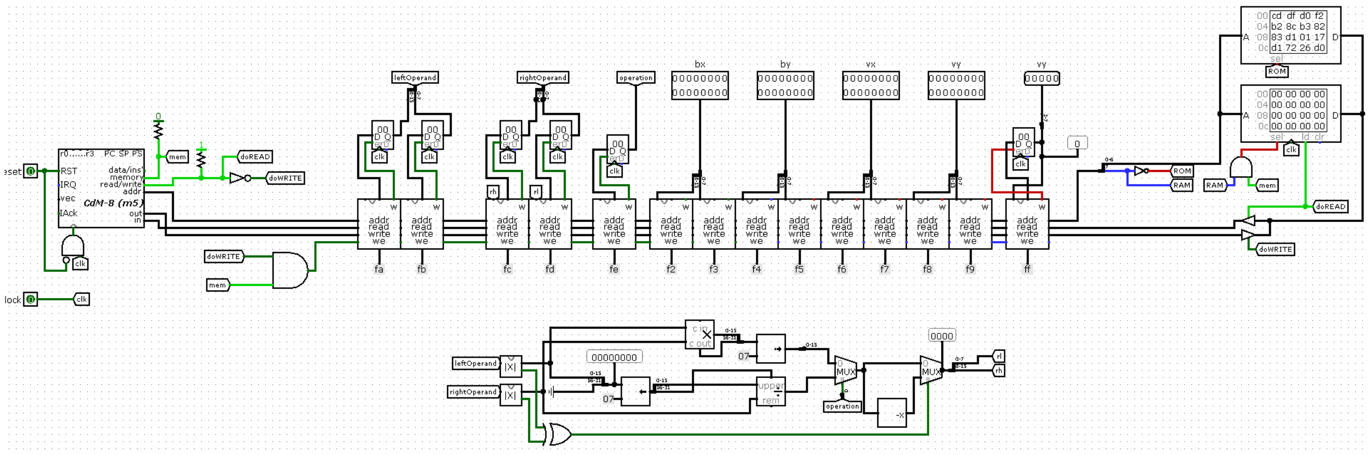

Figure 5: AI chip

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

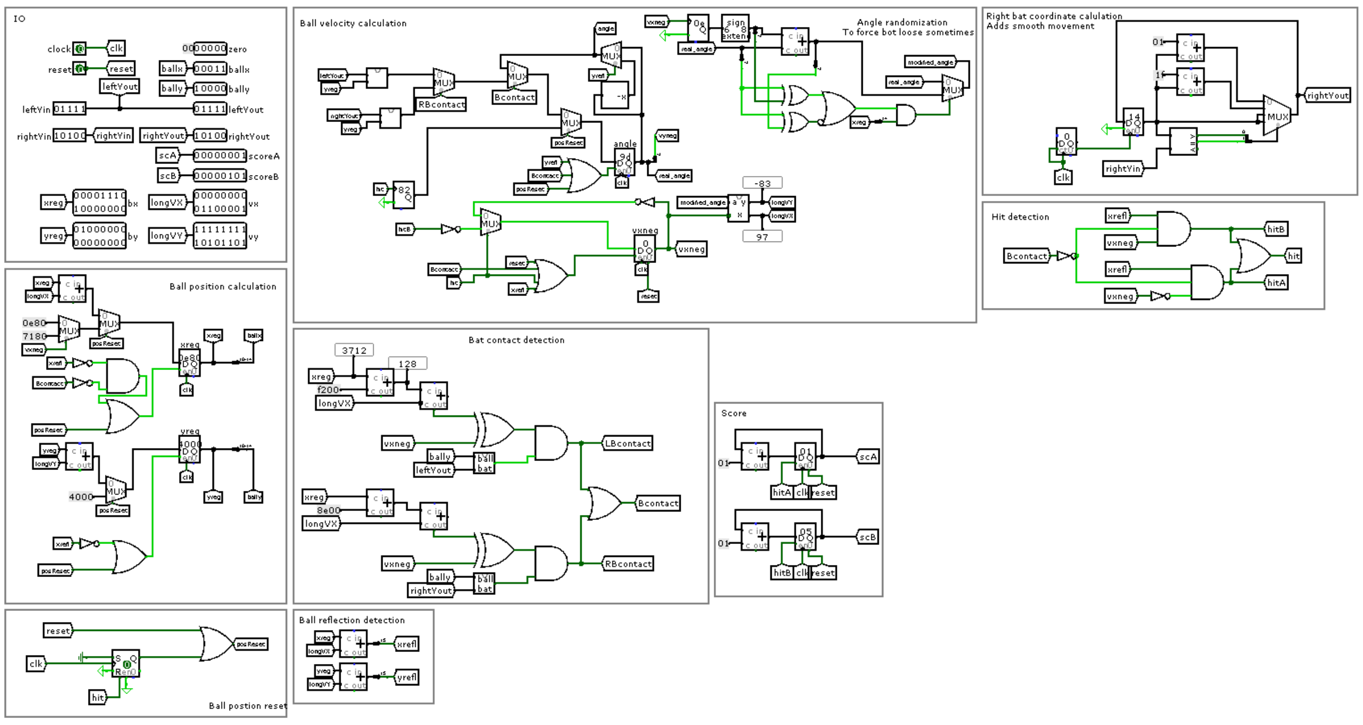

Figure 6: Kinematic controller overview

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_chip*s

This chip is responsible for all gameplay mechanics.

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

- hit - event triggered when ball touches vertical edge of the screen
- posReset - event triggered on game 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 is detected, when addition of velocity component to ball coordinate causes 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 got a point at this tick (side is selected based on the ball direction).

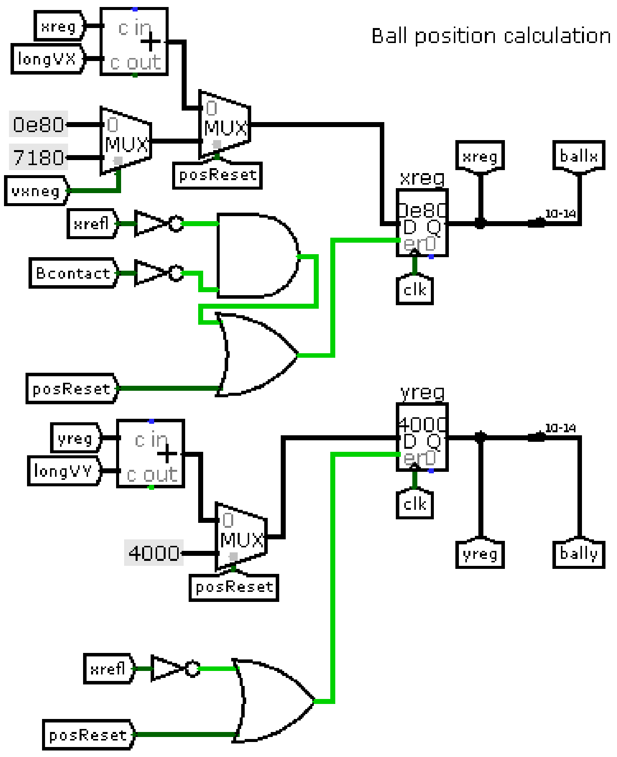

Figure 7: Ball position calculation unit

2.4 Ball velocity calculation

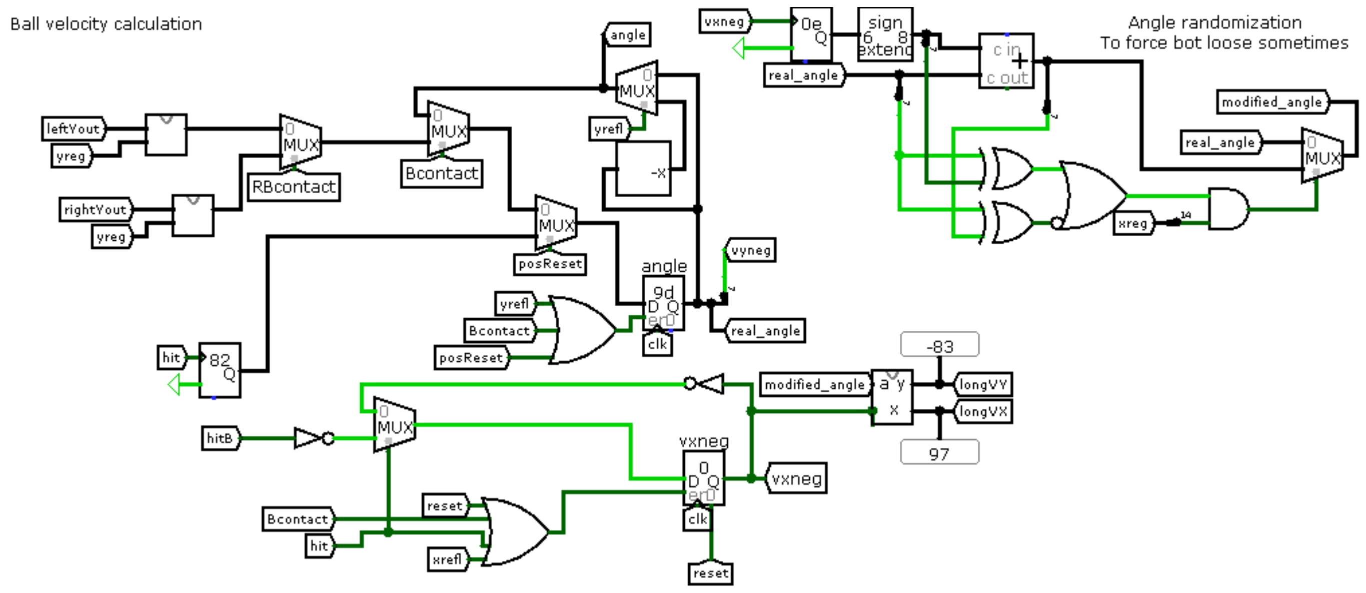

Figure 8: Ball velocity calculation unit

This part is responsible for storing and updating of ball velocity. It separately stores ball angle (a value from $[-52°; +52°]$ 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 and 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 $by - bat$. 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 a distance from left side to right side of the screen, so AI will miss sometimes.

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

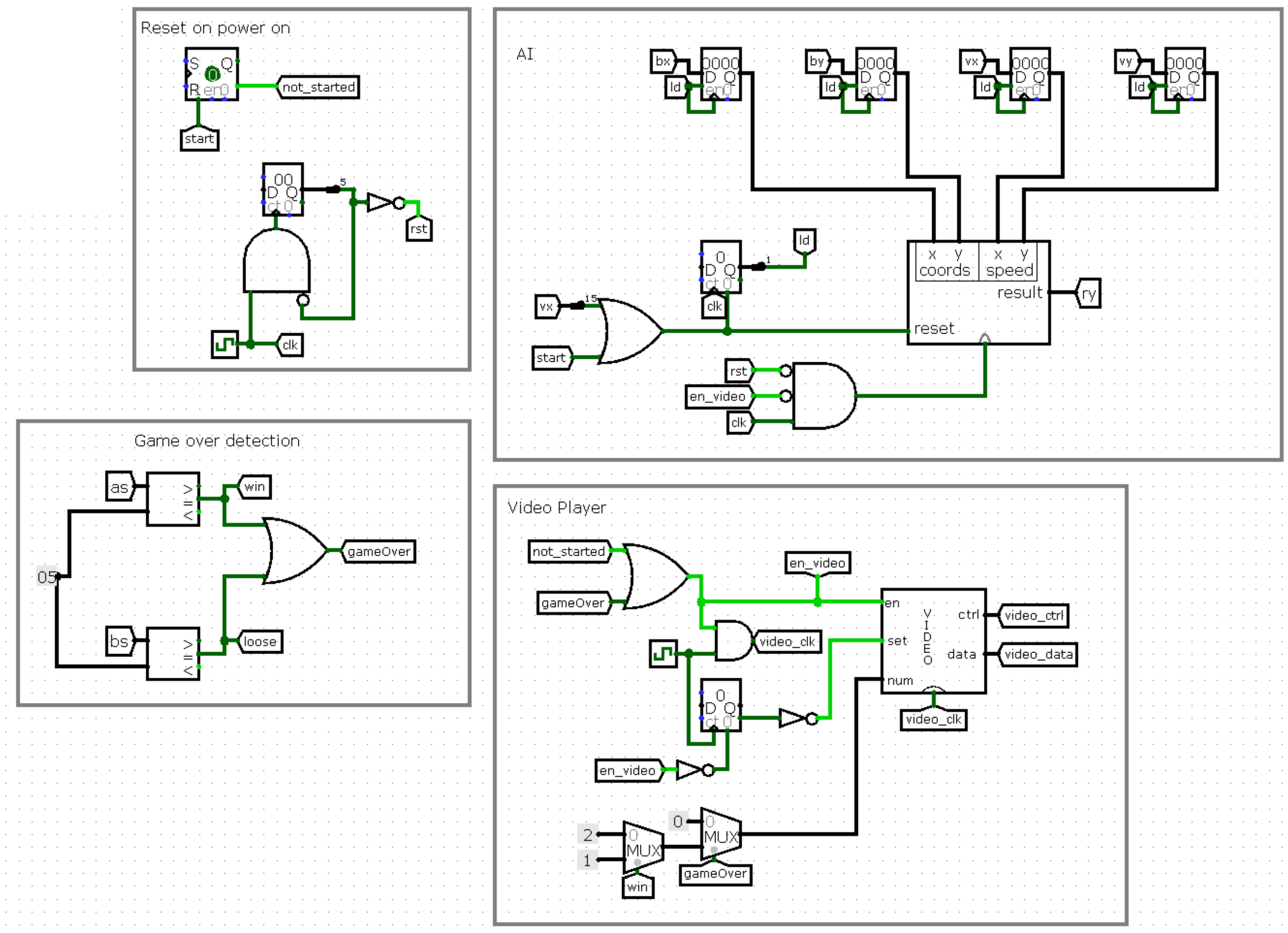

Figure 9: Miscelanious circuitry

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, they prevent ai from cheating by setting bat coordinate to ball coordinate in loop, and 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.

```
# This file contains AI source code for Pong game
#
# Memory map:
# 0x00 - 0x7f: program ROM
# 0x80 - 0xf1: RAM
# 0xf2 - 0xff: IO devices described below
#
# Cdm-8 is connected to hardwre multiplier/divider
#
# Readable IO devices (two addresses => 16-bit device):
# 0xf2, 0xf3: x coordinate of the ball
# 0xf4, 0xf5: y coordinate of the ball
# 0xf6, 0xf7: x velocity of the ball
# 0xf8, 0xf9: y velocity of the ball
# 0xfc, 0xfd: result of multiplication/division
#
# Writable IO devices
# 0xfa, 0xfb: first operand for multiplier/divider
# 0xfc, 0xfd: second operand for multiplier/divider
# 0xfe      : operation (0 - multiplication, 1 - division)
# 0xff      : bat position

asect 0x00
# 0xCD - undocummented instruction that sets SP to it's operrand
dc 0xcd, 0xdf



# main logic goes here
# it computes the following formula: by + (228 - bx)*vy/vx
# 228 = x coordinate of our bat
# all numbers are 16-bit fixed-point  big-endian
# (fractional part is stored in 7 least significant bits)
start:
ldi r0, bx          # load x coordinate of the ball into r2 and r3
ld r0, r2
inc r0
ld r0, r3
not r2              # now we have -bx-1 in registers
not r3


ldi r1, 0x01        # add 228 + 1
add r1, r3
```

```
ldi r1, 0x72
addc r1, r2
# now we have 228 - bx in r2 and r3


# now multiply by vy
# since cdm-8 isn't powerful enough,
# we use hardware multiplier and divisor
# op  - operation code
# op1 - first operand
# op2 - second operand when writing, result when reading
ldi r0, op1        # set operand 1 to 228-bx
st r0, r2
inc r0
st r0, r3

# set operand 2 to vy
ldi r0, vy
ldi r1, op2
jsr mvnum

# set operation to multiplication (operation 0)
ldi r0, 0
ldi r1, op
st r1, r0

# we cannot just copy result to operand 1
# because when first byte of result is written to operand,
# result will immediately change
# so we store the product in memory
# TODO: rewrite it using registers only
ldi r0, op2
ldi r1, 0xc0
jsr mvnum

# now divide by vx
ldi r0, 0xc0            # set operand 1 to (228-bx)*vy
ldi r1, op1
jsr mvnum

ldi r0, vx              # set operand 2 to vx
ldi r1, op2
jsr mvnum

ldi r0, 1               # set operation to division (operation 1)
ldi r1, op
st r1, r0



```

```
# load y coorinate of the ball into r2, r3
ldi r0, by
ld r0, r2
inc r0
ld r0, r3

# load (228-bx)*vy/vx into r0, r1
ldi r1, op2
ld r1, r0
inc r1
ld r1, r1

# add them and store result in r2, r3
add r1, r3
addc r0, r2

# now we only need contents of r2 since bat position are 5 bit
# if bit 7 of r2 is set,
# the ball was reflected either from top or from bottom
tst r2
bpl ready
# if reflection occured, we need to know was it top or bootom
# the following block of code handles it, but i cannot remember how
# TODO: understand my code
ldi r0, vy
ld r0, r0
bmi low_negate
# high_add_128
ldi r0, 127
add r0, r2
br ready
low_negate:
neg r2

# now we have our answer - bat coordinate
ready:
shl r2                                # it was shifted right
ldi r0, 0b11111000                    # we need only 5 high bits
and r0, r2

# in kinematic controller, bat coordinate represents it's lowest pixel
# but we want to reflect ball with central pixel
# so we decrease r2 if it is not zero
tst r2                                # probably unnecessary tst
bz do_write
ldi r0, 0b00001000
sub r2, r0
move r0, r2

# finally move bat
```

```
do_write:
ldi r0, bat
st r0, r2

halt

# this subroutine moves 16-bit numbers
# prameters:
# r0 - source addres
# r1 - destination address
mvnum:
ld r0, r2
st r1, r2
inc r0
inc r1
ld r0, r2
st r1, r2
rts



# here defined addresses of io devices
# dc directives are for debugging
asect 0xf2
bx: dc 0x19, 0x80

asect 0xf4
by:

asect 0xf6
vx: dc 0x13,0x37

asect 0xf8
vy:

asect 0xfa
op1:

asect 0xfc
op2:

asect 0xfe
op:

asect 0xff
bat:

# This is the
end
# my only friend
```

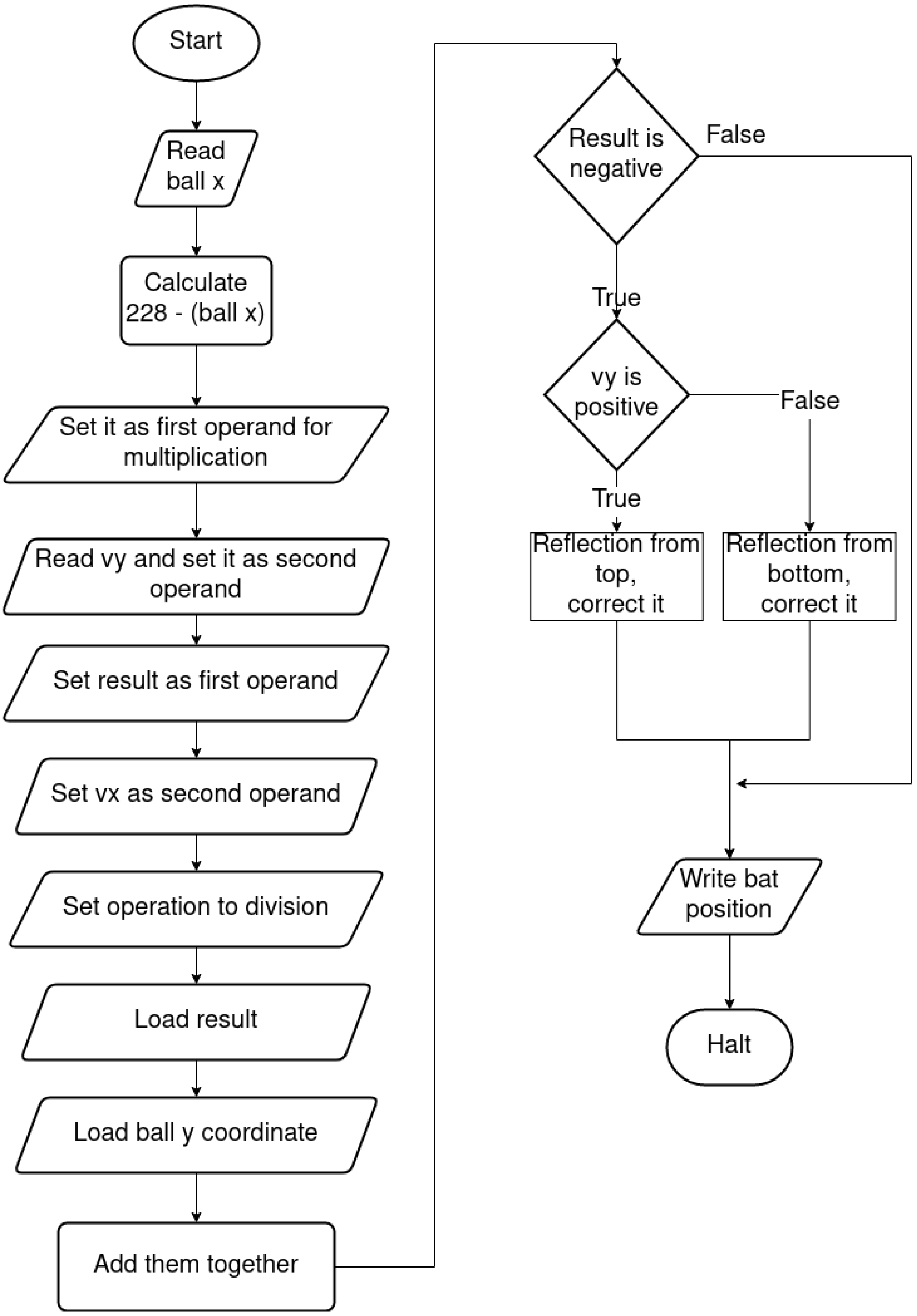
Start
Read ball x
Calculate 228 - (ball x)
Set it as first operand for multiplication
Read vy and set it as second operand
Set result as first operand
Set vx as second operand
Set operation to division
Load result
Load ball y coordinate
Add them together
Result is negative
False
True
vy is positive
False
True
Reflection from top, correct it
Reflection from bottom, correct it
Write bat position
Halt

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)