The Game of TV-Tennis

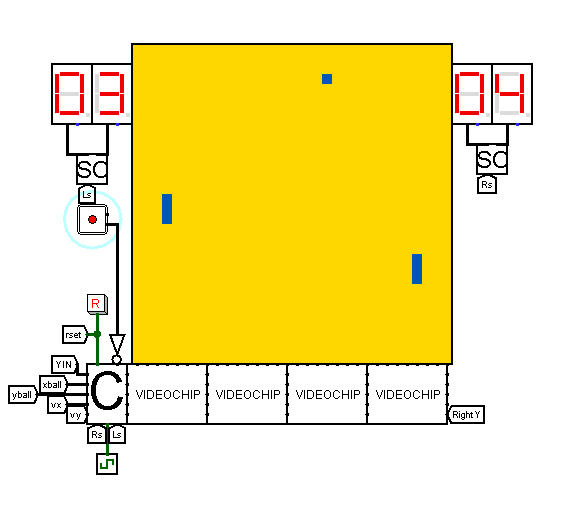
Maxim Chernyshov, Timofey Khrapovitskiy, Artyom Solovyov

2022

Overview

This project was created in 2022 at Novosibirsk State University during the course "Digital Platforms". It is a game of Tv-tennis, which was created using Logisim(hardware) and CDM-8(software). The goal of this project was to create a game against the AI, which would be interesting to play and that it would be possible to beat the AI.

On the screen there are 2 racquets that can move up and down, but not left and right. The ball can move in all directions, the player's goal is to hit it and not let it touch the side of the screen behind his racket. if he fails to hit it, his opponent (in our case the AI) gets a point. The player scores points in the same way - it is necessary that the AI fails to hit the ball. In order to balance the game, the AI racket can only move when the ball is halfway up the player's screen, at other times it always stands still. In our case, the player's racket will be on the left and the AI racket will be on the right. Under these conditions, the player has a chance to win.

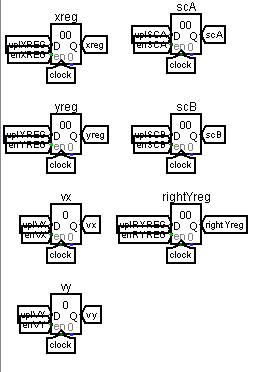


Hardware.

The screen consists of 1024 pixels and has the shape of a square (32x32). Each column receives a 32 bit input pin, one bit per pixel. Each pixel can be lit or unlit. In order to display 2 racquets and a ball, we need the coordinates of the ball and the Y coordinate of both racquets. The racquet consists of 3 pixels going in a row, lit in the same column. But we only need a central one, on either side of which we light the pixels together, getting the racquet.

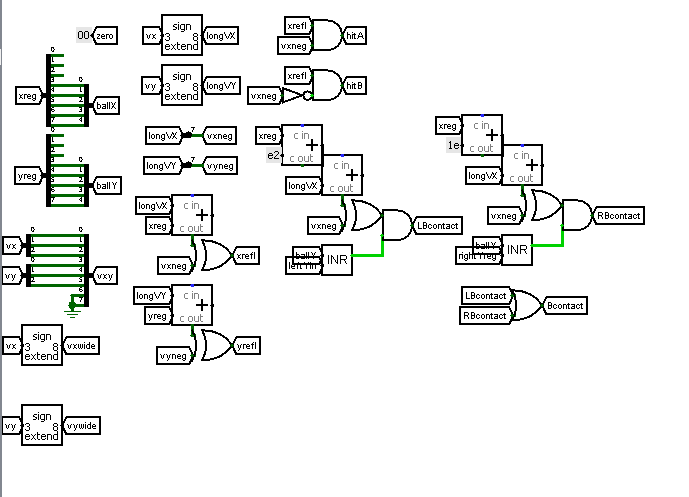
3.2) In order to display the screen we need a video system. It consists of 32 video chips, which are combined into 4 sections of 8. Each video chip is responsible for its own column numbered from 0 to 31. We number them so that they know which column they are responsible for. This works like this: each chip has an extra input in which their ID is written. We add 1 to the chip ID as we go through the video system. so we give the first chip ID 0b00000 and the chips get the correct numbering. the chip also takes the ball and Y coordinates of the right and left racquet. The special columns in the section are 3 and 4. Column 3 is the column with the left racquet (in section 0) and column 4 is the column with the right racquet (in section 3). The ball can be in any point on the screen, so both the ball and the racket can be in these columns at the same time.

**Kinematic controller.**  This project component updates the position of the ball, block the right racquet in the right side of the screen.



These registers are needed to store scores, coordinates, velocity of the ball and the Y coordinate of the racquet.

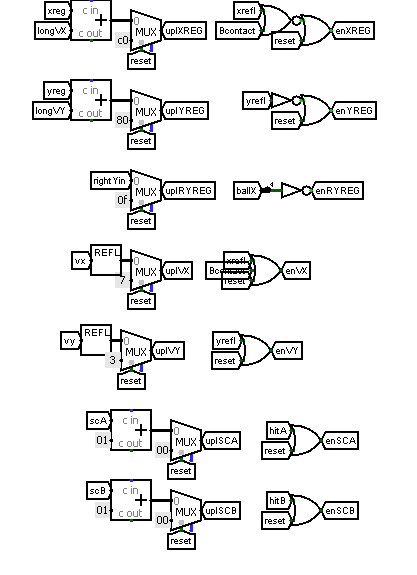
In contrast to the displayed screen, the size of the space to move the ball and calculate its coordinates is a square 256x256. So the display will be such that a pixel with coordinates (0;0) will represent any position in a square (0-7, 0-7) on that grid.  Because of the way the screen is displayed, we can easily turn 8-bit ball coordinates for the kinematic controller into 5-bit coordinates for the video system. We take up to 5 significant bits with the highest value, and if they are less than 5, we refill them with zeros. The biggest error is the sum 1+2+4 = 7 in the case of 255. 255-7=248 is still in range of the top right square. The racquet Y coordinate is a 5-bit value, so we don't have to worry about that.



Ball speed is represented by two 3-bit two’s complement values(one for the horizontal and one for the vertical velocity). They can be any value in the range [-4;3]. This allows us to have both coordinates stored in one 8bit value for the CdM-8. Vx(horizontal) will be the 0-2 bits and vy(vertical) will be the 3-5 bits. Bits 6 and 7 are set to zero. What happens when the ball hits the wall or racquet? The horizontal speed is negated if it hit the horizontal surface, the vertical speed is negated if it hit the vertical surface. If the ball hit a corner, both vx and vy are negated. Velocity equal to -4 is the exception to that rule, because there is no velocity +4. Instead of +4, -4 becomes +3.

Movement of the ball is iterative. The ball updates it’s coordinate and, if needed, velocity. To achieve this movement we use a local clock. First the ball changes it’s coordinates, then the velocity. It means that there are two steps every cycle of the clock. The first step is changing the coordinate, the second step is changing the velocity.

How do we check if the ball collides with a wall? We detect carry bit, and if vx is lower than 0, carry should be 1, then the collision happens. If vx is higher than 0, carry should be 0, then the collision happens. The same way we detect vertical collisions. When we detect any kind of collision, we negate the velocity and instead of continuing to move with the same velocity the ball bounces in the opposite direction. The corner collision is just two collisions at the same time.



Software.

#we check if vx is positive. if it is, we compute the position of the ball.

#if not, we put the bat in the middle of Y axis.

get vx

get ybat

Main:

if (vx >0)

Compute

else

ybat = 127

while (vx <= 0)

get vx

if (vx > 0)

Compute

Compute:

get xball,yball,vy

# add the velocity of the ball to the x coordinate,because vx is read 2 ticks after

# yball, and 2 cdm-8 ticks equal 1 kinematic tick

xball = xball + vx

#we are supposed to calculate 224-xball, but we will count 227 because for big vy

#the ball travels too much

res = 227 - xball

#now the division by vx

if (vx == 2)

#shift right by 1 and make res positive

res = res >> 1

res = res & 127

else

if (vx == 4)

#same as with 2, but shift it by 2

res = res >>2

res = res & 63

else

if (vx == 3)

# to divide res by 3 we need to do something else.

# Let's imagine AB and CD are numbers, A,B,C,D are 4 bits of data #each.

# AB\*CD = EF, where E and F are 8 bits of data each

# F = B\*D + (B\*C) <<2 + (A\*D) << 2

# E = carry\_bit + A \* C + (A \* D) >> 2 + (B \* C) >> 2

# res / 3 = res \*(2^8 /3 ) >> 8 = res \* 86 = E

# We'll start by computing A and B

# And we know that C = 5, D = 6

A = (res >> 4) & 15

B = res & 15

#B\*D (which is 6)

F = B <<2 + B << 1

#B\*C<<2 (which is 5)

F = F + ( B << 2 + B ) << 2

#(A\*D)<<2

F = F + (A <<2 + A <<1) << 2

#carry bit

if (F > 255)

carry\_bit = 1

else

carry\_bit = 0

# We need to multiply our number by vy

get vy

flag\_vy\_negative = 0

carry\_cnt = 0

# In case of negative vy we negate it , it will affect the result later , so it's #okay

if ( vy < 0 )

vy = - vy

flag\_vy\_negative = 1

#now we count carry bits when multiplying res by vy

if (vy == 2)

res = res << 1

if (res > 255)

carry\_cnt = carry\_cnt + 1

res = res - 256

else

if (vy ==4)

res = res << 1

if ( res > 255)

carry\_cnt = carry\_cnt +1

res = res - 256

res = res << 1

if ( res > 255)

carry\_cnt = carry\_cnt +1

res = res - 256

else

if (vy == 3 )

res0 = res

res = res << 1

if ( res > 255)

carry\_cnt = carry\_cnt +1

res = res - 256

res = res0 + res

if (res > 255)

carry\_cnt = carry\_cnt +1

res = res - 256

get yball

# if vy was negative , we negate the y coordinate

if (flag\_vy\_negative == 1)

yball = -yball

res = res + yball

if (res > 255)

carry\_cnt = carry\_cnt +1

res = res - 256

# we count carry bits and if the number is odd we negate the result

if (carry\_cnt % 2 = 1)

res = -res