Novosibirsk State University

Course project

in the discipline of "Digital platforms"

"The Snake" Game

Made by:
1st year students, group 21933
Babenko Egor Stepanovich
Ivanov Gleb Evgenievich

Novosibirsk, 2022

Table of contents

1 Introduction

Our goal was to create our own version of the Snake game. To implement our plans, we used Logisim and the CDM-8 processor, which is emulated in Logisim. We also used CocoIDE to write code for the processor.

1.1 Concept

The player controls a snake that moves around the playing field.

Food appears on the field. The coordinates of the food are random. If the snake "eats" it (the snake's head is in the same place as the food), then the snake becomes longer and the player gets 1 point.

If a player scores a certain number of points (in our project, 96 points are needed to win), then the game ends and the inscription «WIN» appears on the screen.

If the snake bump into the border of the playing field or into the tail of the snake, the game ends and the inscription «LOSE» appears on the screen.

The snake should not be able to move "inside itself" (for example, first move to the left, and then immediately start moving to the right).

1.2 Purposes

- Create a screen to display the playing field and create a scheme to control this screen.
- Create a screen to display the points that the player has scored.
- Implement the interaction of the CDM-8 processor with the user (use buttons or keyboard).
- Create an additional Logisim scheme for random number generation.
- Implement the connection of additional Logisim modules to the CDM-8 processor.
- Implement the Concept using the software or hardware part of the project.

1.3 Versions

We have two main versions of our project:

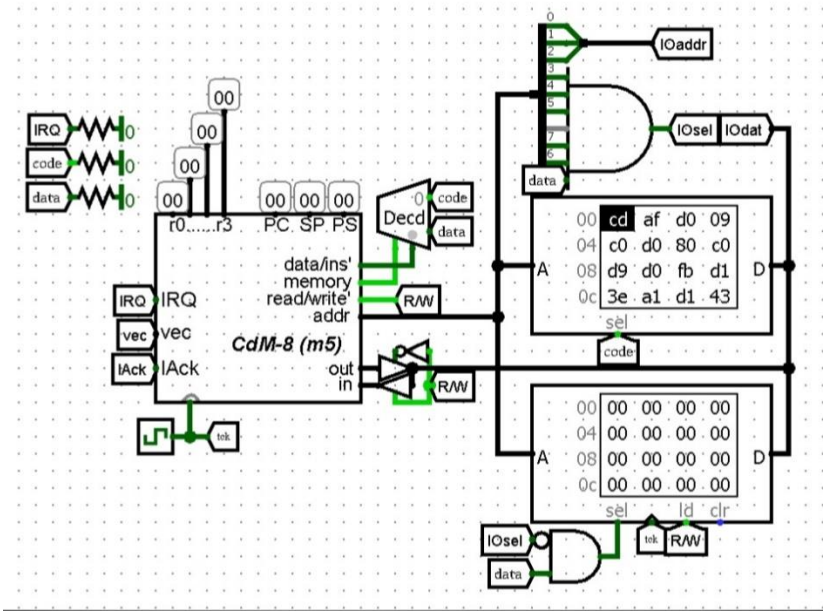
- Hardware version. In this version, the detection of bumps into walls or into snake is performed using Logisim elements. Also, "eating apples" is detected outside the processor.
- Software version. In this version, bumps and "eating apples" are detected in the processor program

We also have a hardware version in which the bot controls the second snake. But due to lack of time, we did not have time to optimize this version.

We have some ideas on how to optimize this version:

- Implement a more correct choice of a new path, if there is an obstacle on the way. The bot must take into account the current position of the apple when changing the route due to an obstacle (If horizontal movement is blocked, then the bot should try to go vertically so that it gets nearer to the apple).
- To improve the algorithm of laying the starting route better. The bot must choose which coordinate it aligns dynamically at the beginning and be able to change the priority direction along the way.
- Minimize the number of checks.

2 Processor



Pic.1 – Processor, memory elements and a circuit that connects external devices

For our project, we used the fifth version of CDM-8 with an 8-bit address bus. We used the Harvard microcontroller architecture.

In the bot version, an additional ROM element is used. Switching between ROM elements is implemented using an external device "MemorySelector", which redirects the address bus.

All code is written in the CDM-8 assembler language using the CocoIDE development environment.

2.1 Connecting external devices

In our configuration, the processor uses up to 8 addresses of external devices (3-bit addressing). The addresses 0xf8 – 0xff are used for this. All external devices for the processor look like ordinary memory cells, but splitter and AND gate redirect requests from these addresses to external devices.

2.2 Assembler program

2.2.1 Software version

```
1.  #Markup of the address space of external devices
2.  asect 0xf8
3.    displayIO: #The first address of the screen (Responsible for rendering the head)
4.  asect 0xf9
5.    randomGeneratorIO: #Address of the device for receiving random numbers
6.  asect 0xfa
7.    appleIO: #The second address of the screen (Responsible for rendering the apple)
8.  asect 0xfb
9.    controllerIO: #Address of the keyboard controller
10. asect 0xfc
11.   scorePointerIO: #Address of the score screen controller
12. asect 0xfd
13.   freeDrawIO: #The third address of the screen (Responsible for rendering preloaded images)
14. #RAM markup
15. asect 0x90
16.   tailArr: ds 96 #Addresses for an array of snake elements
17. asect 0xf0
18.   appleCoords: ds 1 #Address for storing apple coordinates
19.   tailPos: ds 1 #Address for storing a pointer to the tailArr array
20.   gameScore: ds 1 #Address for storing the current score
21.   yPlayer: ds 1 #Address for storing the coordinates of the snake's head by Y
22.   xPlayer: ds 1 #Address for storing the coordinates of the snake's head by X
```

In the first 13 lines, the distribution of addresses for external devices takes place. The address space for external devices starts with address F8 and ends with address FF. In the next block, starting from line 14 and ending with line 22, the processor RAM is allocated.

```
23. #Segment of initialization of RAM and external devices
24. data_initilization:
25.       setsp 0x8F #Move stack pointer
26.  #Initializing the keyboard handler
27.       ldi r0, controllerIO #Loading function addresses into the keyboard module
28.       ldi r1, move_left
29.       st r0, r1
30.       ldi r1, move_right
31.       st r0, r1
```

```
32.     ldi r1, move_up
33.     st r0, r1
34.     ldi r1, move_down
35.     st r0, r1
36. #Initializing RAM and screen
37.     ldi r0, displayIO
38. #Loading the initial position of the snake
39.     ldi r3, yPlayer #Loading to memory and to the screen initial position of snake
40.     ldi r1, 0b00000110
41.     st r3, r1 #RAM    (Y)
42.     st r0, r1 #Screen (Y)
43.     inc r3
44.      st r3, r1 #RAM    (X)
45.     st r0, r1 #Screen (X)
46.     ldi r1, 0b00000000
47.     st r0, r1 #Loading an empty value to update the frame
48. #Initializing an array of snake elements
49.     ldi r0, tailPos
50.     ldi r1, tailArr
51.     ldi r2, 0b01100110
52.     st r1, r2 #Loading the starting position of the snake into array
53.     st r0, r1 #Updating the pointer to the current element in the array
```

This segment implements initialization of external devices and RAM. From line 27 to line 35, the keyboard module is initialized (the "keyboardChip"[4.2] circuit), 4 addresses are sequentially loaded into it and stored in the "Address storage"[4.2.3] module.

From line 39 to line 47, the display and memory are initialized using the coordinates of the initial position of the snake(X = 6, Y = 6).

From line 49 to line 53, an array of snake elements is initialized with the initial coordinates, and a pointer to the tailArr array is stored at the tailPos address. tailPos indicates the coordinates that will be transmitted in the next frame to turn off the pixel on the screen.

```
54. #Button press waiting cycle
55.     ldi r1, controllerIO
56.     wait_for_press:
57.             ld r1, r0
58.             tst r0
59.     bz wait_for_press  #If the keyboard handler returned a non-zero value -> exit the loop
60.
61.     jsr generate_new_apple #Create apple on the screen
62.     ldi r0, displayIO
```

```
63. #Starting the main program cycle
64.     mainloop:
65.         ldi r1, controllerIO
66.         ld r1, r1 #Reading the value from the keyboard handler
67.         push r1 #Push it on stack
68.         jsr load_XY_packets #This function loads values from Y and X from memory into registers r1 and r2, respectively
69.         rts #Go to the received address
70.     br mainloop
```

55-59 line - the processor waits for any of the buttons ("w ", "a ", "s ", "d ") to be pressed, after that an apple is created (61 line) and the processor enters the main program cycle (64-70 lines). Each iteration of this cycle is equal to one game frame.

In this cycle, the processor reads the last key pressed from the "keyboardChip"[4.2] circuit and loads the Y and X coordinates into registers r1 and r2. The processor gets the address from the keyboard module to which the processor needs to switch. After that, the processor calculates the new position of the snake's head and sends it to the "displayControllerChip"[3.1] circuit. After that, the processor deletes the last element of the tail. Next, it checks whether the head bump into the tail or into the border of the playing field. The processor also checks whether the snake has eaten an apple.

```
71. move_left:
72.     dec r2
73.     br update_head
74. move_right:
75.     inc r2
76.     br update_head
77. move_up:
78.     inc r1
79.     br update_head
80. move_down:
81.     dec r1
82.     br update_head
```

From line 71 to line 82 are the movement commands. The corresponding coordinate changes and the processor switches to the segment of drawing the head and checking the rules.

```
83. update_head: #Uses values from r1(Y) and r2(X) to update r0(IO-0)
84.     ldi r0, displayIO
85.  #Draw new head position on the screen
86.     st r0, r1 #Send Y to screen
87.     st r0, r2 #Send X to screen
88.  #Update the coordinates of the head in RAM
```

```
89.     ldi r3, yPlayer
90.     st r3, r1 #Save Y to RAM
91.     inc r3
92.     st r3, r2 #Save X to RAM
```

In lines 86 and 87, the new coordinates of the head are sent to the "displayControllerChip"[3.1] circuit. After that, from line 89 to line 92, these coordinates are stored in RAM.

```
93.  #Checking for crossing the boundaries of the playing field (if cross -> die)
94.     ldi r3, 0b11110000 #load mask
95.     and r1, r3 #Check Y coordinate
96.     bnz PLAYER_LOSE
97.     ldi r3, 0b11110000 #load mask
98.     and r2, r3 #Check X coordinate
99.     bnz PLAYER_LOSE
100.#Check if snake ate apple
101.    ld r0, r1 #load coordinates of head like YX
102.    ldi r3, appleCoords
103.    ld r3, r3 #load coordinates of apple
104.    cmp r1, r3 #Compare it
105.    bz APPLE_ATE #If they match, run the corresponding subroutine
106.APPLE_UPDATED: #Return from apple ate handler
```

From line 93 to line 99, there is a check at the intersection of the boundaries of the playing field. If any of the top 4 bits in the coordinate is not 0, then the snake has crossed the border of the playing field. In this case, the processor switches to the PLAYER_LOSE label and processes the player's loss.

From line 101 to 106, checking whether the apple was eaten by a snake. The coordinates of the snake's head are loaded from the "displayControllerChip"[3.1] circuit and compared with the coordinates of the apple stored in memory. If the coordinates match, go to the APPLE_ATE label. And further return to the APPLE_UPDATED label.

```
107.    ldi r1, tailPos #read ptr on ptr on element in tailArr
108.    ld r1, r1     #read ptr on tailArr
109.    ld r1, r3     #read pixel coordinates
110.    st r0, r3     #send coordinates for turning off a pixel on the screen
```

The lines above contain commands that delete the last element of the snake's tail from the screen. The tailPos label stores a pointer to an element in the tailArr array, which is the tail of a snake.

```
111.#Checking for the snake crossing itself
112.#r2 - current head coordinates | r1 – pointer of the tailArr | r3 - game score
113.    ldi r3, gameScore
```

```
114.    ld r3, r3
115.    inc r3
116.    ldi r1, tailArr #Load start of array
117.    ld r0, r2 #read current head (Like YX) From screen module
118.    check_tail_bump_loop: #Check loop
119.        ld r1, r0 #Load element from array
120.        cmp r2, r0 #Compare it with head
121.        bz PLAYER_LOSE #If it equal -> we have collision
122.        inc r1 #Switch to next element
123.        dec r3 #Decrease counter (Like game score | Snake elements on screen)
124.    bnz check_tail_bump_loop
```

From line 113 to line 117, all the necessary values are loaded into registers to check whether the snake crossed its tail.

The registers store:

- r0 – Display address
- r1 – Pointer to the beginning of the tailArr array
- r2 – Coordinates of the current head position
- r3 – Game score (Snake length)

The coordinates of the head are read from the "displayControllerChip"[3.1]. From line 118 to line 124, there is an intersection check cycle. Coordinates are read from the array of snake elements and compared with the current position of the head. If the result of the comparison is 0, the processor switches to the PLAYER_LOSES label.

```
125.    ldi r0, displayIO
126.#add new element into tailArr on tailPos position
127.    ldi r1, tailPos #read ptr on ptr on element in tailArr
128.    ld r1, r1   #read ptr on tailArr
129.    ld r0, r3   #read current head (Like YX) From screen module
130.    st r1, r3 #store head on tailPos ptr
131.    dec r1    #Decrease tailPos
132.    ldi r2, tailArr #load address of start of array
133.    cmp r1, r2 #check if new tailPos lesser thank start of array
134.    bge updated #if it bigger goes to updated
135.    ldi r3, gameScore #else fix tailPos. Move it to the end of the array
136.    ld r3, r3 #Load into r3 game score
137.    add r3, r2 #Simple add game score to start of array
138.    move r2, r1 #Copy new tailPos to r1
139.updated:
140.    ld r0, r2 #read current head (Like YX) From screen module
141.    ldi r0, tailPos
```

```
142.    st r0, r1 #Store new tailPos
143.    ldi r0, displayIO #Reload to r0 screen address
144.    br mainloop #go to start of main cycle
```

From line 125 to line 130, the processor writes the current coordinates of the head to the tailArr array. After that, on lines 131-133, it is checked whether tailPos has become less than tailAar (tailPos has gone beyond the bounds of the array). If this happens, the processor goes over the updated label on line 134. The current game score is added to the register that stores the tailArr and the resulting value is stored in tailPos (line 141). If the tailPos has not gone out of bounds, then it is saved (line 142). After that, the processor returns to the beginning of the main program cycle (line 144).

```
145.load_XY_packets: #[Subroutine]# loads from memory Y and X coordinates of head into r1 and
    r2, respectively
146.    ldi r3, xPlayer
147.    ld r3, r2
148.    dec r3
149.    ld r3, r1
150.    rts
```

This is a subprogram that loads the coordinates of the snake's head from RAM into registers r1 and r2. To register r1 – Y. To register r2 – X.

```
151.generate_new_apple: #[Subroutine]# receive random number from generator and send it to
    apple render (Part of screen)
152.    ldi r0, randomGeneratorIO
153.    ld r0, r1       #Load random 8-bit number
154.    inc r0          #Switch to appleIO addr
155.    st r0, r1       #Load number to appleIO
156.    ldi r0, appleCoords #Save apple coordinates to memory
157.    st r0, r1
158.    rts
```

A subprogram that gets a random 8-bit number from a "randomGenChip"[5] and sends it to the "Apple draw"[3.1.4] module.

```
159.PLAYER_WIN: #[Subroutine]#
160.    ldi r1, 16 #image code (0 - "Lose") (16 - "Win")
161.    br freeDrawer
162.PLAYER_LOSE: #[Subroutine]#
163.    ldi r1, 0 #image code (0 - "Lose") (16 - "Win")
164.    br freeDrawer
165.#Draws on the display selected preloaded image and shutdown processor
```

```
freeDrawer: #In r1 must lie number of pictures
    ldi r0, freeDrawIO
    st r0, r1 #Send image ID to freeDrawerIO
    ldi r3, 16
#Send16 signals to redraw screen
    draw_loop: #write predefined text
            st r0, r1
            dec r3
     bnz draw_loop
     halt
```

This is a subprogram that draws a pre-loaded picture. Depending on the entry point, it writes WIN or LOSE on the screen. The entry points are PLAYER_WIN or PLAYER_LOSE. They load the image id into the r1 register and transfer control to the freeDrawer subprogram. It loads the value into the "Free draw block"[3.1.5]. After that, the processor sends 16 consecutive requests to this port to render the full image.

```
APPLE_ATE: #[Subroutine]#
    push r0 #Save registers
    push r1
    push r2
    ldi r2, 96 #Current max score
    jsr generate_new_apple #Draw new apple on screen
    ldi r0, gameScore       #Load game score from RAM
    ld r0, r1
    inc r1      #Increase score
    cmp r1, r2      #Check if player have enough score to win
    bz PLAYER_WIN       #if have – goto.
    st r0, r1       #Save in memory current score
    ldi r0, scorePointerIO #Send game score to score show bar
    st r0, r1
    pop r2 #restore register values
    pop r1
    pop r0
    br APPLE_UPDATED
end
```

This is a subprogram that processes the eaten apple. It increases the number of points by 1 (184 line). The subprogram also updates the glasses display screen (Line 189). If the points have become equal to those needed for victory, then the processor switches to a subprogram to display the victory (line 186).

2.2.2 Hardware version

The main difference between the hardware and software versions is interruptions. In the hardware version, interrupt checks (bumping into the borders of the playing field or into the snake's tail, eating an apple) are transferred to external devices. The "Interruptions"[3.1.6] module checks the data from the "displayControllerChip"[3.1] module. If one of the interruption situations (apple eating, bumping into the tail or into the border) has happened, the Interruptions module sends a request to the processor using the IRQ and vec ports.

This made it possible to speed up the game and lower the needed number of clock cycles per game frame.

```
#Markup of the address space of external devices
asect 0xf8
  displayIO: #The first address of the screen (Responsible for rendering the head)
asect 0xf9
  randomGeneratorIO: #Address of the device for receiving random numbers
asect 0xfa
  appleIO: #The second address of the screen (Responsible for rendering the apple)
asect 0xfb
  controllerIO: #Address of the keyboard controller
asect 0xfc
  scorePointerIO: #Address of the score screen controller
asect 0xfd
  freeDrawIO: #The third address of the screen (Responsible for rendering preloaded images)
#Interuptions
 asect 0xf4
  dc SNAKE_DEATH
  dc 1
  dc APPLE_ATE
  dc 1
#RAM markup
asect 0x90
  playerTailArr: ds 96 #Addresses for an array of snake elements
asect 0xf0
  playerTailPos: ds 1 #Address for storing a pointer to the tailArr array
  gameScore: ds 1 #Address for storing the current score
  yPlayer: ds 1 #Address for storing the coordinates of the snake's head by Y
  xPlayer: ds 1 #Address for storing the coordinates of the snake's head by X
asect 0x00
init: #Crutch for enabling interrupts
        ldi r0, data_initilization #load address of return from crutch
```

```
31.             push r0
32.              ldi r0, 0b10000000 #load new PS value
33.             push r0
34.             rti #update PS and move to data_initilization
35.  #Segment of initialization of RAM and external devices
36.  data_initilization:
37.             setsp 0x8F #Move stack pointer
38.   #Initializing the keyboard handler
39.             ldi r0, controllerIO  #Loading function addresses into the keyboard module
40.             ldi r1, move_left
41.             st r0, r1
42.             ldi r1, move_right
43.              st r0, r1
44.             ldi r1, move_up
45.             st r0, r1
46.             ldi r1, move_down
47.             st r0, r1
48.  #Initializing RAM and screen
49.             ldi r0, displayIO
50.   #Loading the initial position of the snake
51.             ldi r3, yPlayer #Loading to memory and to the screen initial position of snake
52.             ldi r1, 0b00000110
53.             st r3, r1 #RAM    (Y)
54.             st r0, r1 #Screen (Y)
55.              inc r3
56.             st r3, r1 #RAM    (X)
57.             st r0, r1 #Screen (X)
58.             ldi r1, 0b00000000
59.              st r0, r1 #Loading an empty value to update the frame
60.   #Initializing an array of snake elements
61.             ldi r0, playerTailPos
62.             ldi r1, playerTailArr
63.             ldi r2, 0b01100110
64.             st r1, r2 #Loading the starting position of the snake into array
65.             st r0, r1 #Updating the pointer to the current element in the array
66.   #Button press waiting cycle
67.             ldi r1, controllerIO
68.             wait_for_press:
69.                     ld r1, r0
70.                     tst r0
71.             bz wait_for_press #If the keyboard handler returned a non-zero value -> exit the loop
72.             jsr generate_new_apple #Create apple on the screen
```

```
73.             ldi r0, displayIO
74.  #Starting the main program cycle
75.             mainloop:
76.                     ldi r1, controllerIO
77.                     ld r1, r1 #Reading the value from the keyboard handler
78.                     push r1 #Push it on stack
79.                     jsr load_XY_packets #This function loads values from Y and X from memory into registers
    r1 and r2, respectively
80.                     rts #Go to the received address
81.             br mainloop
82.  move_left:
83.             dec r2
84.             br update_head
85.  move_right:
86.             inc r2
87.              br update_head
88.  move_up:
89.              inc r1
90.             br update_head
91.  move_down:
92.              dec r1
93.              br update_head
94.  update_head: #Uses values from r1(Y) and r2(X) to update r0(IO-0)
95.  #Draw new head position on the screen
96.             st r0, r1 #Send Y to screen
97.             st r0, r2 #Send X to screen
98.  #Update the coordinates of the head in RAM
99.             ldi r3, yPlayer
100.            st r3, r1 #Save Y to RAM
101.            inc r3
102.            st r3, r2 #Save X to RAM
103.
104.#Deactivate last element of snake on the screen
105.            ldi r1, playerTailPos #read ptr on ptr on element in playerTailArr
106.            ld r1, r1 #read ptr on playerTailArr
107.            ld r1, r3 #read pixel coordinates
108.            st r0, r3 #send coordinates for turning off a pixel on the screen
109.
110.#Add new element into playerTailArr on playerTailPos position
111.            ld r0, r3 #read current head (Like YX) From screen module
112.            st r1, r3 #store head on playerTailArr ptr
113.            dec r1 #Decrease playerTailPos
```

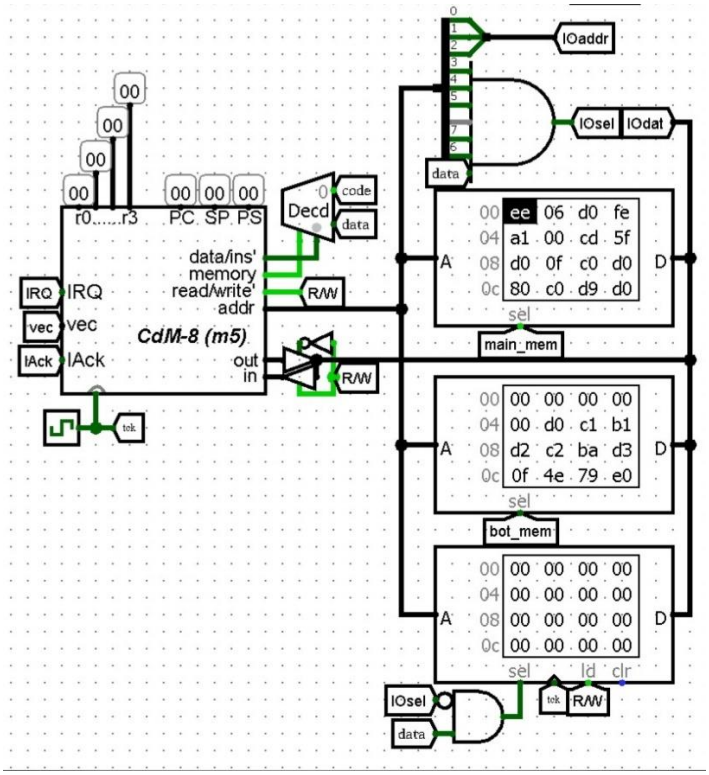
```
114.        ldi r2, playerTailArr #load address of start of array
115.        cmp r1, r2 #check if new playerTailPos lesser thank start of array
116.        bge updated #if it bigger goes to updated
117.        ldi r3, gameScore #Load into r3 game score
118.        ld r3, r3 #Load into r3 game score
119.        add r3, r2 #Simple add game score to start of array
120.        move r2, r1 #Copy new playerTailPos to r1
121.updated:
122.        ldi r0, playerTailPos
123.        st r0, r1 #Store new playerTailPos
124.        ldi r0, displayIO #Reload to r0 screen address
125.        br mainloop #go to start of main cycle
126.
127.load_XY_packets: #[Subroutine]# loads from memory Y and X coordinates of head into r1 and r2,
    respectively
128.        ldi r3, xPlayer
129.        ld r3, r2
130.        dec r3
131.        ld r3, r1
132.        rts
133.
134.generate_new_apple:  #[Subroutine]# receive random number from generator and send it to apple render
    (Part of screen)
135.
136.        ldi r0, randomGeneratorIO
137.        ld r0, r1 #Load random 8-bit number
138.        inc r0 #Switch to appleIO addr
139.        st r0, r1 #Load number to appleIO
140.        rts
141.
142.PLAYER_WIN: #[Subroutine]#
143.                ldi r1, 16 #Send image code (0 - "Lose") (16 - "Win")
144.  br freeDrawer
145.SNAKE_DEATH: #[Interruption]#
146.        ldi r1, 0 #Send image code (0 - "Lose") (16 - "Win")
147.         br freeDrawer
148.#Draws on the display selected preloaded image and shutdown processor
149.freeDrawer: #In r1 must lie id of image
150.        ldi r0, freeDrawIO
151.        st r0, r1 #Send image ID to freeDrawerIO
152.        ldi r3, 16
153.  #Send 16 signals to redraw screen
```

```
154.        draw_loop: #write predefined text
155.                st r0, r1
156.                dec r3
157.        bnz draw_loop
158.        halt
159.APPLE_ATE: #[Interruption]# called when the snake has eaten an apple
160.        push r0 #Save register values
161.        push r1
162.         push r2
163.        ldi r2, 96 #Current max score
164.        jsr generate_new_apple #Draw new apple on screen
165.        ldi r0, gameScore #Load game score from RAM
166.        ld r0, r1
167.         inc r1 #Increase score
168.         cmp r1, r2 #Check if player have enough score to win
169.        bz PLAYER_WIN #if have – goto.
170.        st r0, r1 #Save in memory new score
171.        ldi r0, scorePointerIO #Send game score to score show bar
172.         st r0, r1
173.        pop r2 #restore register values
174.        pop r1
175.        pop r0
176.        rti
177.end
```

As a result, the game works according to this algorithm:

1) Reading the pressed key from the keyboard.
2) Processing of movement in the appropriate direction.
3) Sending new head coordinates to the display.
4) Processing of snake elements (Removing the last element of the snake's tail, as well as maintaining the integrity of the array of snake elements).
5) After the processor has sent which pixel needs to turn off on the screen, it can receive two interruptions:
   - An interruption about an apple being eaten
   - An interruption that the snake has collided with something
6) In the next step, the processor processes the received interruptions, and also restores all its registers, preparing to repeat the entire cycle again.

2.2.3 Bot version



Pic.2 - Processor, memory elements and a circuit that connects external devices in the Bot version

This version is based on the hardware version. Only the memory allocation changes. Some values are added for the bot and the logic of sending the current score to the screen changes.

Main ROM element

```
1.  #Markup of the address space of external devices
2.  asect 0xf8
3.    displayIO: #The first address of the screen (Responsible for rendering the head)
4.  asect 0xf9
5.    randomGeneratorIO: #Address of the device for receiving random numbers
6.  asect 0xfa
7.    appleIO: #The second address of the screen (Responsible for rendering the apple)
8.  asect 0xfb
9.    controllerIO: #Address of the keyboard controller
10. asect 0xfc
11.   scorePointerIO: #Address of the score screen controller
12. asect 0xfd
13.   freeDrawIO: #The third address of the screen (Responsible for rendering preloaded images)
14. asect 0xfe
15.   switchMemoryCell:
16. asect 0xff
17.   botBackdoor:
```

```
18. #Interuptions
19.  asect 0xf4
20.   dc SNAKE_DEATH
21.   dc 1
22.   dc APPLE_ATE
23.   dc 1
24. #RAM markup
25. asect 0x60
26.   playerTailArr: ds 48
27.   botTailArr: ds 48
28. asect 0xc0
29.   yBot: ds 1
30.   xBot: ds 1
31.   yPlayer: ds 1
32.   xPlayer: ds 1
33.   appleCoords: ds 1
34.   playerTailPos: ds 1
35.   botPlayerPos: ds 1
36.   playerScore: ds 1
37.   botScore: ds 1
38.   isloop: ds 1
```

And also transitions between ROM modules are added.

```
48. asect 0x00
49.   br init #On start of program goto init
50.   enable_bot_memory:  #label for switch to second bank
51.   ldi r0, switchMemoryCell #Load address of switcher
52.   st r0, r1 #push something to it
53. asect 0xEB #Return address from the bot's memory bank
54.   #Return from bot_memory
55.   br mainloop
56. asect 0x06
57. #Next comes the code as in the hardware version
```

This part of the code implements the change of ROM modules. To switch from the main ROM module to the bot ROM module, the processor switches to the enable_bot_memory label. After that, the memory bank is switched using an external device by the switchMemoryCell label and the code execution continues from address 0x05.

To return to the main ROM module, the same subprogram as enable_bot_memory above should be run in the bot ROM module, but it should return to the address 0xEB.

The enable_bot_memory subprogram is called after the full frame is drawn for the player, at the place where in the software version the transition to the beginning of the cycle takes place. There are small changes in the apple eating interruption.

```
168. APPLE_ATE: #[Interruption]# called when the snake has eaten an apple
169.         push r0 #Save register values
170.         push r1
171.         push r2
172.         ldi r2, 48 #Current max score
173.         jsr generate_new_apple #Draw new apple on screen
174.         ldi r0, playerScore #Load game score from memory
175.         ld r0, r1
176.         inc r1 #Increase score
177.         cmp r1, r2 #Check if player have enough score to win
178.         bz PLAYER_WIN #if have – goto.
179.         st r0, r1 #Save in memory current score
180.         ldi r2, 0b10000000
181.         or r2, r1 #Set 7 bits per unit, a marker that these are the human's points
182.         ldi r0, scorePointerIO #Redraw score on screen
183.          st r0, r1
184.          pop r2 #Restore register values
185.         pop r1
186.         pop r0
187.          rti
```

The seventh bit of the current number of points is used as a marker for an external device, about which player to add points to (181 line). 1 – to human, 0 – to bot.

Bot ROM element

```
1.  #Markup of the address space of external devices
2.  asect 0xf8
3.    displayIO: #The first address of the screen (Responsible for rendering the head)
4.  asect 0xf9
5.    randomGeneratorIO: #Address of the device for receiving random numbers
6.  asect 0xfa
7.    appleIO: #The second address of the screen (Responsible for rendering the apple)
8.  asect 0xfb
9.    controllerIO: #Address of the keyboard controller
10. asect 0xfc
11.   scorePointerIO: #Address of the score screen controller
```

```
asect 0xfd
 freeDrawIO: #The third address of the screen (Responsible for rendering preloaded images)
asect 0xfe
 switchMemoryCell:
asect 0xff
 botBackdoor:
#Interuptions
 asect 0xf4
 dc SNAKE_DEATH
 dc 1
 dc APPLE_ATE
 dc 1
#RAM markup
asect 0x60
 playerTailArr: ds 48
 botTailArr: ds 48
asect 0xc0
 yBot: ds 1
 xBot: ds 1
 yPlayer: ds 1
 xPlayer: ds 1
 appleCoords: ds 1
 playerTailPos: ds 1
 botPlayerPos: ds 1
 playerScore: ds 1
 botScore: ds 1
 isloop: ds 1
asect 0xE8
enable_main_memory:  #Point of transition to another memory bank
        ldi r1, switchMemoryCell
        st r1, r1
asect 0x05
        ldi r0, xBot #Load current x coordinate of bot head
         ld r0, r1
        ldi r2, appleCoords #Load coordinates of apple
        ld r2, r2
        ldi r3, 0b00001111 #Load mask for X coordinates of apple
        and r3, r2
        cmp r2, r1 #Compare it
         bz bot_YMove #If it equal -> bot need to move up/down
        bge bot_right #If the bot is to the left of the apple (the coordinate of the apple is greater than the bot)
-> go to the right
```

```
bot_left: #else go left
        dec r1 #Decrease X coordinate
        dec r0 #Switch address to yBot
#check if we have a checking loop (We checked all 4 sides and didn't found free space to move)
        save r0
        ldi r2, isLoop #load check value
        ld r2, r3
        save r2
        ldi r2, 0b00001000 #xor with some ID of side
        xor r2, r3
        restore
        ld r2, r0 #Load start check value
        cmp r3, r0 #Compare new value with old
        restore
        ble bot_drawX #If new value lesser than old => we visit this part of code twice => The bot has no choice where to go
        st r2, r3 #save new check value (to isLoop)
        ld r0, r2 #Load yBot to r2
        ldi r3, botBackdoor #peek for the left pixel
        st r3, r2 #Send coordinates where the bot wants to go  (Y)
        st r3, r1 #Send coordinates where the bot wants to go (X)
        ld r3, r3 #Receive result (1 – This cell occupied; 0 – Free)
        tst r3
        bnz bot_down #If pixel is ocuppied => try down
        br bot_drawX #else make left move

bot_right:
        inc r1 #Increase X coordinate
        dec r0 #Switch pointer on yBot
#check if we have a checking loop (We checked all 4 sides and didn't found free space to move)
        save r0
        ldi r2, isLoop #load check value
        ld r2, r3
        save r2
        ldi r2, 0b00000100 #xor with some ID of side
        xor r2, r3
        restore
        ld r2, r0
        cmp r3, r0 #Compare new value with old
        restore
        ble bot_drawX #If new value lesser than old => we visit this part of code twice => The bot has no choice where to go
```

```
93.             st r2, r3 #save new check value (to isLoop)
94.             ld r0, r2 #Load yBot to r2
95.             ldi r3, botBackdoor #peek for the right pixel
96.             st r3, r2 #Send coordinates where the bot wants to go  (Y)
97.             st r3, r1 #Send coordinates where the bot wants to go (X)
98.             ld r3, r3 #Receive result (1 – This cell occupied; 0 – Free)
99.             tst r3
100.            bnz bot_up #If pixel is ocuppied => try down
101.#br bot_drawX  # [OPTIMIZATION] #else make move to right
102.
103.bot_drawX:
104.            ld r0, r2 #load Y coordinate from RAM
105.             inc r0 #Switch address to xBot
106.            st r0, r1 #save new X coordinate to RAM
107.            br bot_draw #Send new bot head to screen
108. bot_YMove: #Check can bot move to up/down
109.            ldi r0, yBot #Load current Y coordinate of head
110.            ld r0, r2
111.            ldi r1, appleCoords  #Load coordinate of apple
112.            ld r1, r1
113.            ldi r3, 0b11110000 #Take from apple coordinate only Y
114.            and r3, r1
115.            shr r1 #Shift it to lower bits
116.            shr r1
117.            shr r1
118.             shr r1
119.            cmp r1, r2 #Compare with bot head
120.            ld r0, r1
121.            bgt bot_up #If apple coordinate if bigger than bot head => move up
122.bot_down: #Else move down
123.            dec r2 #Decrease Y coordinate
124.             inc r0 #Switch address to xBot
125.#check if we have a checking loop (We checked all 4 sides and didn't found free space to move)
126.            save r0
127.            ldi r1, isLoop #load check value
128.            ld r1, r3
129.            save r1
130.            ldi r1, 0b00000010 #xor with some ID of side
131.            xor r1, r3
132.            restore
133.            ld r1, r0 #Load start check value
134.            cmp r3, r0 #Compare new value with old
```

```
135.        restore
136.        ble bot_drawY #If new val lesser than old => we visit this part of code twice => The bot has no
    choice where to go
137.        st r1, r3 #save new check value (to isLoop)
138.        ld r0, r1 #Load xBot to r1
139.        ldi r3, botBackdoor #peek for the down pixel
140.        st r3, r2 #Send coordinates where the bot wants to go  (Y)
141.        st r3, r1 #Send coordinates where the bot wants to go (X)
142.        ld r3, r3 #Receive result (1 – This cell occupied; 0 – Free)
143.         tst r3
144.        bnz bot_right  #If pixel is ocuppied => try right
145.        br bot_drawY #else go down
146.bot_up:
147.        inc r2
148.        inc r0
149. #check if we have a checking loop (We checked all 4 sides and didn't found free space to move)
150.        save r0
151.        ldi r1, isLoop #load check value
152.        ld r1, r3
153.        save r1
154.        ldi r1, 0b00000001 #xor with some ID of side
155.        xor r1, r3
156.        restore
157.        ld r1, r0 #Load start check value
158.        cmp r3, r0 #Compare new value with old
159.        restore
160.        ble bot_drawY #If new val lesser than old => we visit this part of code twice => The bot has no
    choice where to go
161.         st r1, r3 #save new check value (to isLoop)
162.         ld r0, r1 #Load xBot to r1
163.        ldi r3, botBackdoor #peek for the left pixel
164.        st r3, r2 #Send coordinates where the bot wants to go  (Y)
165.        st r3, r1 #Send coordinates where the bot wants to go  (X)
166.        ld r3, r3 #Receive result (1 – This cell occupied; 0 – Free)
167.        tst r3  #If pixel is ocuppied => try left
168.        bnz bot_left  #If pixel is ocuppied => try left
169.  #else go up
170.
171.bot_drawY:
172.        ld r0, r1 #load X coordinate from RAM (xBot)
173.        dec r0 #Switch address to yBot
174.        st r0, r2 #save new Y coordinate to RAM (yBot)
```

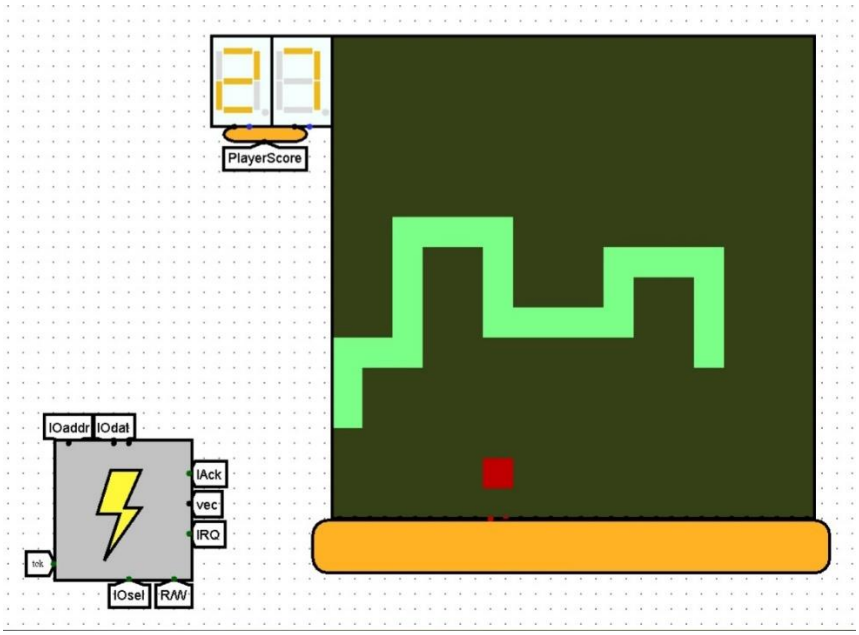
```
175.            #br bot_draw  #[OPTIMIZED] #Send new bot head to screen
176.
177.bot_draw: #Draw new head and tail on ehe screen
178.            ldi r0, isLoop  #Set 0 to check value for loop checker
179.            clr r3 #Set 0 to check value for loop checker
180.             st r0, r3 #Set 0 to check value for loop checker
181.#Start screen update
182.            ldi r0, displayIO
183.            st r0, r2 #Send Y to screen
184.            st r0, r1 #Send X yo screen
185. #Deactivate last element of snake on the screen
186.            ldi r1, botTailPos #read ptr on ptr on element in botTailArr
187.            ld r1, r1 #read ptr on botTailArr
188.            ld r1, r3 #read pixel coordinates
189.            st r0, r3 #send coordinates for turning off a pixel on the screen
190.
191. #Add new element into tailArr on tailPos position
192.            ld r0, r3 #send coordinates for turning off a pixel on the screen
193.            st r1, r3 #store head on botTailArr ptr
194.            dec r1 #Decrease playerTailPos
195.            ldi r2, botTailArr #load address of start of array
196.            cmp r1, r2 #check if new playerTailPos lesser thank start of array
197.            bge bot_updated #if it bigger goes to updated
198.             ldi r3, botScore #else #Load into r3 game score
199.            ld r3, r3 #Load into r3 game score
200.            add r3, r2 #Simple add game score to start of array
201.            move r2,r1 #Copy new botTailPos to r1
202.bot_updated:
203.             ldi r0, botTailPos
204.            st r0, r1 #Store new botTailPos
205.             ldi r0, displayIO #Reload to r0 screen address
206.             br enable_main_memory #Switch memory bank to main memory
207.
208.generate_new_apple: #[Subroutine]# receive random number from generator and send it to apple render (Part
    of screen)
209.             ldi r0, randomGenerator
210.             ld r0, r1 #Load random 8-bit number
211.            inc r0 #Switch to appleIO addr
212.            st r0, r1 #Load number to appleIO
213.            ldi r0, appleCoords #Save apple coordinates in memory
214.            st r0, r1
215.            rts
```

```
216.
217.BOT_WIN: #[Subroutine]#
218.         ldi r1, 0 #Send image code (0 - "Lose") (16 - "Win")
219.         br freeDrawer
220.BOT_LOSE: #[Interruption]#
221.         ldi r1, 16 #Send image code (0 - "Lose") (16 - "Win")
222.         #br freeDrawer #[OPTIMIZATION]
223.
224.#Draws on the display selected preloaded image and shutdown processor
225.freeDrawer: #In r1 must lie id of image
226.         ldi r0, freeDrawIO
227.         st r0, r1 #Send image ID to freeDrawerIO
228.         ldi r3, 16
229.         #Send 16 signals to redraw display
230.         draw_loop: #write predefined text
231.                 st r0, r1
232.                 dec r3
233.         bnz draw_loop
234.         halt
235.APPLE_ATE: #[Interruption]# called when the snake has eaten an apple
236.         push r0 #Save register values
237.         push r1
238.         push r2
239.         ldi r2, 48 #Current max score
240.         jsr generate_new_apple #Draw new apple on screen
241.         ldi r0, botScore #Load game score from RAM
242.         ld r0, r1
243.         inc r1 #Increase score
244.         cmp r1, r2 #Check if bot have enough score to win
245.         bz BOT_WIN #if have – goto.
246.         st r0, r1 #Save in memory new score
247.         ldi r0, scorePointerIO #Send game score to score show bar
248.         st r0, r1
249.         pop r2 #Restore register values
250.         pop r1
251.         pop r0
252.         rti
253.end
```

The bot paves its way according to a very simple algorithm. First aligns its position with the X coordinate (goes to the right or left to the apple). After that, it aligns its Y coordinate with the apple

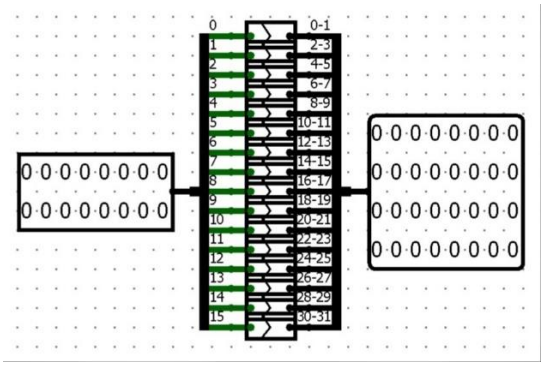
(Goes up or down). The bot also has a check to see if there is an obstacle in the position where the bot wants to get to. It works like this, the bot "peeks" at the screen and finds out if the pixel is turned on where the bot wants to go. If it is lit, the bot chooses a different direction. In this version, due to lack of time, we did not have time to optimize the code and improve the pathfinding algorithm.

3 Displaying the playing field

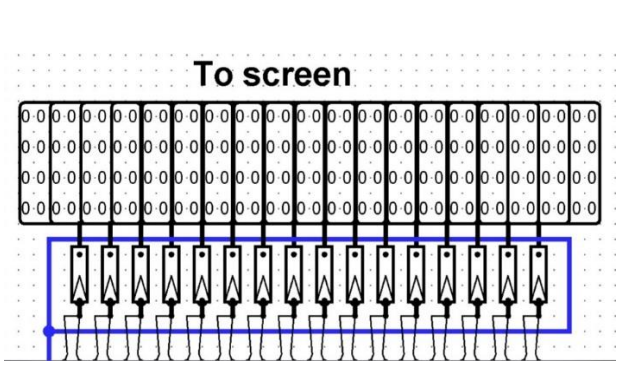


Pic.3 - LED matrix and "displayControllerChip" circuit (Front-end)

The playing field has a size of 16x16 pixels, but we used a 32x32 LED matrix to display it. To display each cell of the playing field on the matrix, a 2x2 LED square is turned on. The increase in the horizontal dimensions of the playing field pixel is realized by duplicating signals when connecting the "displayControllerChip"[3.1] circuit to the LED matrix. The increase in vertical dimensions is implemented using the "bitExtender(16-32)" subcircuit.



Pic.4



Pic.5

Pic.4 - "bitExtender(16-32)" circuit (Back-end)

Pic.5 - "bitExtender(16-32)" circuits that are connected to the To screen ports (Front-end)

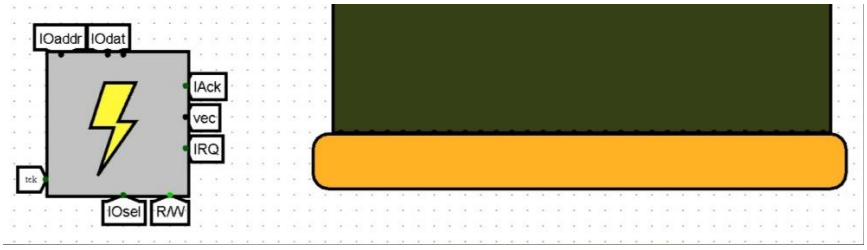
This circuit converts a 16-bit value to a 32-bit one. Expanding each bit from a 16-bit value to two bits from a 32-bit value (0 -> 0,1; 1 -> 2,3; ...).

The matrix displays 3 colors:

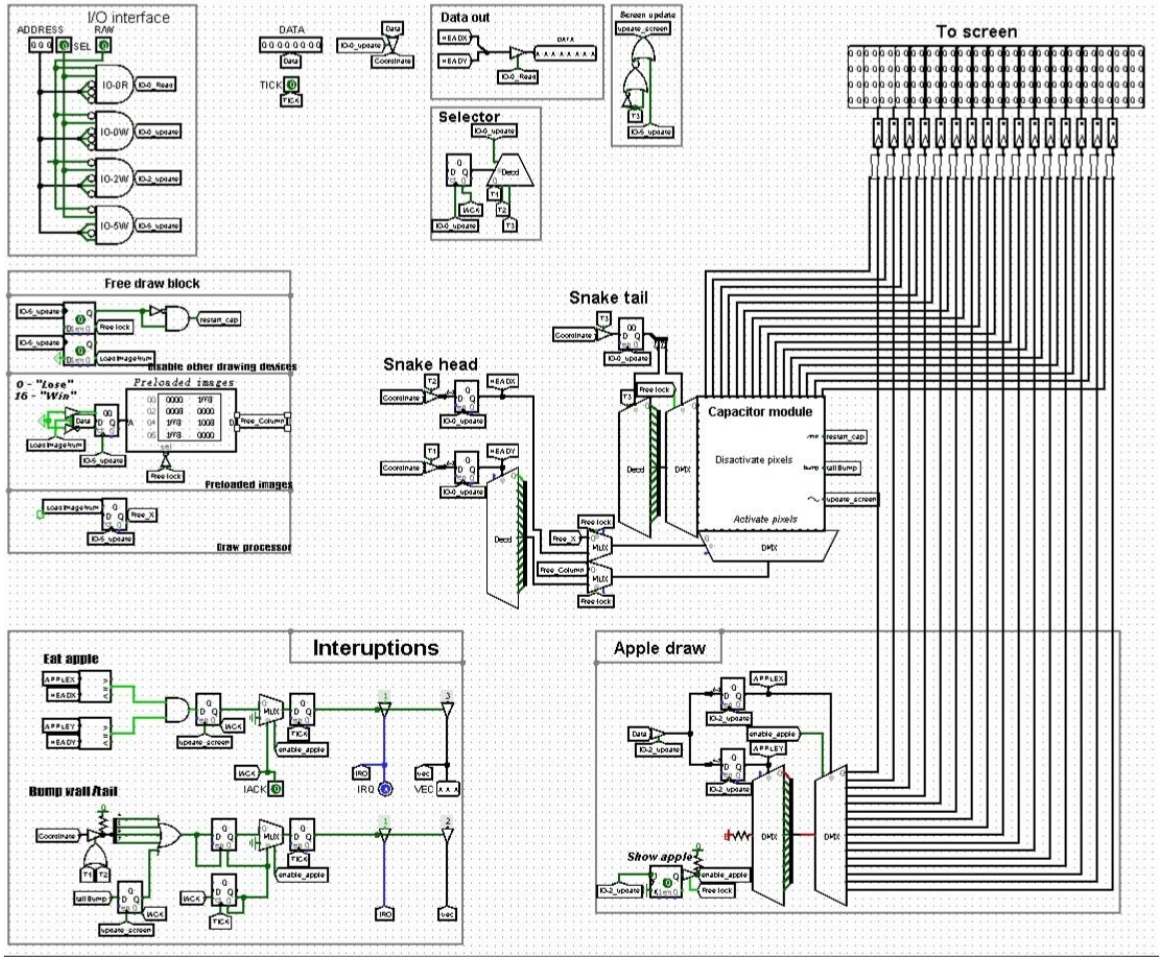
- The enabled LED has a light green color. The enabled LEDs display the location of the snake's body on the playing field.
- The disabled LED has a dark green color. The disabled LEDs are the background.
- If an error signal is applied to the LED, the LED has a red color. Red LEDs display the location of the apples on the playing field.

3.1 "displayControllerChip"

To control the LED matrix, we have created a "displayControllerChip".



Pic.6 - the "displayControllerChip" circuit that is connected to the LED matrix (Front-end)



Pic.7 - the "displayControllerChip" circuit that is connected to the LED matrix (Back-end)

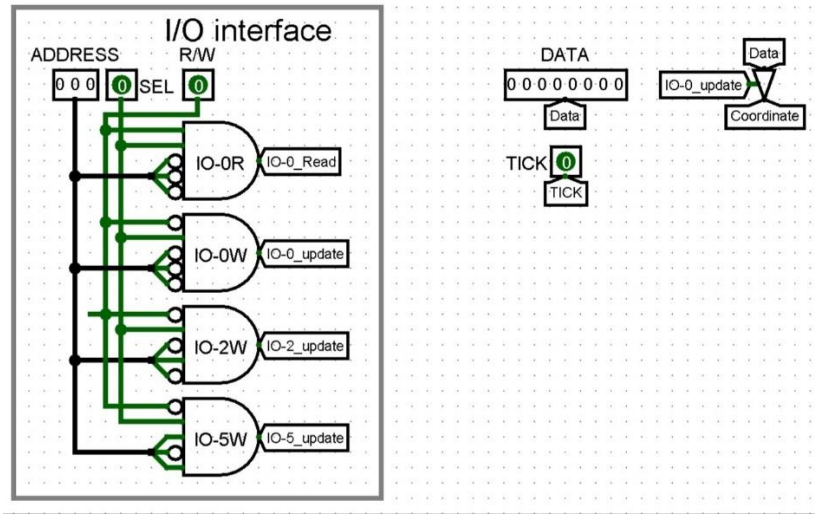
Input ports:

- Clock generator port (TICK, 1-bit)
- Device address port (ADDRESS, 3-bit)
- Port for external device activation signal (SEL, 1-bit)
- Data port (DATA, 8-bit)
- Port for signal Reading/Recording (R/W, 1-bit)
- Signal of interrupt processing by processor (IACK, 1-bit) *Available only in the hardware version

Output ports:

- Data port (DATA, 8-bit)
- Interrupt request port (IRQ, 1-bit) *Available only in the hardware version
- The interrupt number port (VEC, 3-bit)
- 16 ports for controlling columns of the LED matrix (To screen, 32-bit)

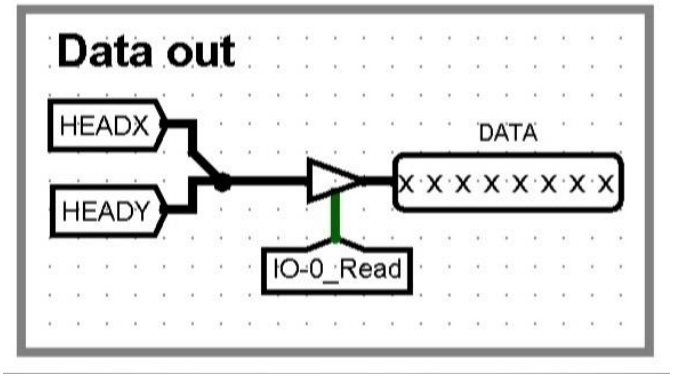
3.1.1 "I/O interface"



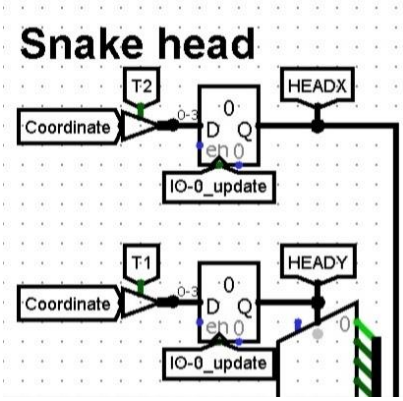
Pic.8 – the "I/O interface" module and input ports

To start working with "displayControllerChip"[3.1], you need to send a signal to the SEL port, send the command address to the ADDRESS port and select the read mode or write mode using the R/W port. After that, the "I/O interface" module activates the appropriate interface.

3.1.1.1 IO-0R



Pic.9



Pic.10

Pic.9 – the "Data out" module

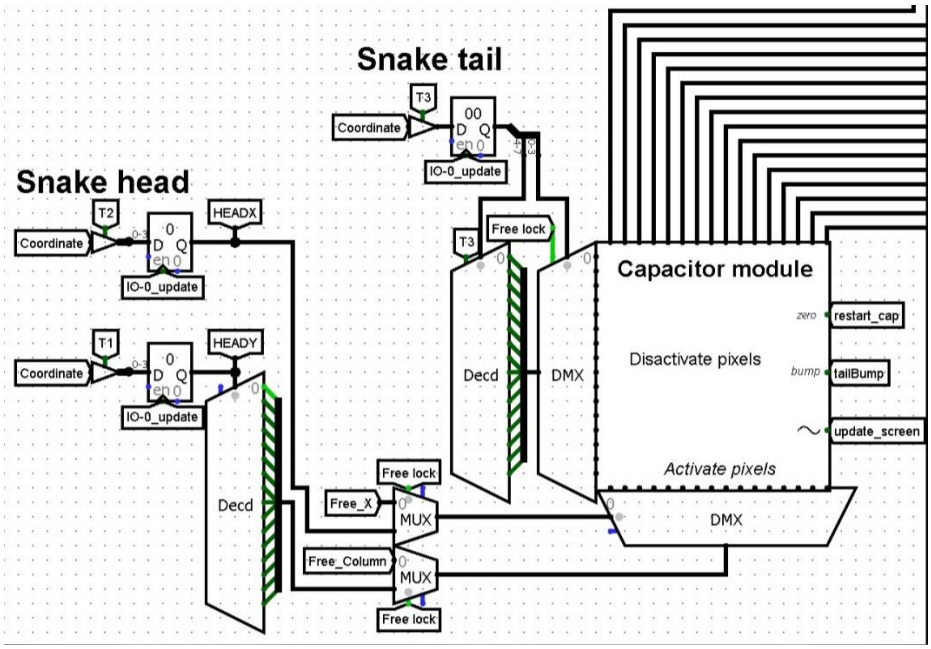
Pic.10 - registers for X and Y coordinates

The IO-0R interface sends the coordinates of the snake's head from the registers at the request of the processor. The coordinates are sent to the DATA port. 0-3 bits are the X coordinate, 4-7 bits are the Y coordinate.

This interface is activated if:

- SEL = 1
- R/W = 1
- ADDRESS = 000

3.1.1.2 IO-0W



Pic.11 - the scheme of enabling and disabling pixels

Interface IO-0W implements the movement of the snake.

This interface is activated if:

- SEL = 1
- R/W = 0
- ADDRESS = 000

First, the coordinates of the head or tail are read from the DATA port.

This interface has a cycle of 3 applications (Each full cycle is 1 game frame. The frame is updated at the beginning of each cycle):

1) The Y coordinate is stored in the register.
2) The X coordinate is stored in the register.

   According to these coordinates (the coordinates of the snake's head), the pixel of the LED matrix is turned on. The coordinates are decoded, the Y coordinate turns into a 16-bit value meaning a pixel in the column. The X coordinate is used as the selection bit in the demultiplexer to translate the Y value into the desired column. This data is transmitted to the Activate pixels ports of the "capacitorModule"[3.2] circuit.

3) The coordinates of the last element of the snake's tail are stored in the register. 0-3 bits are the X coordinate, 4-7 bits are the Y coordinate.

   According to these coordinates, the pixel is "turned off" in the "capacitorModule"[3.2] circuit. The data is transmitted in the same way as for the snake's head. However, instead of Activate pixels contacts, Disactivate pixels contacts are used.

This cycle is implemented using the "Selector"[3.1.2] module.

3.1.1.3 IO-2W

Interface IO-2W is responsible for rendering the apple.

This interface is activated if:

- SEL = 1
- R/W = 0
- ADDRESS = 010

First, the coordinates of the apple are read from the DATA port. After that, an apple is created using these coordinates. To do this, the “Apple draw” module is used.

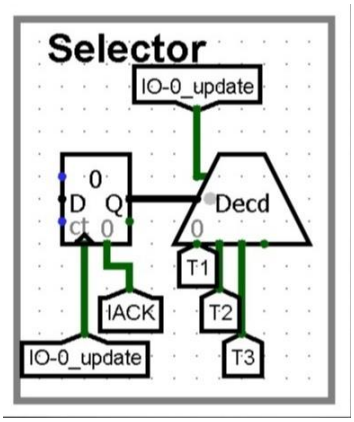
3.1.1.4 IO-5W

Interface IO-5W renders the "WIN" screen or the "LOSE" screen. Also, this interface disables the rendering of the snake and apple. To do this, the interface uses the "Free draw block"[3.1.5] module. The image number received from the DATA port is transmitted to this module.

This interface is activated if:

- R = 1
- R/W = 0
- ADDRESS = 101

3.1.2 "Selector"

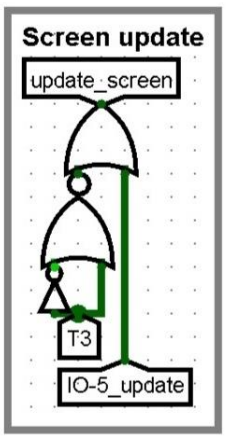


Pic.12 – the "Selector" module

When the "Selector" module is activated, the value in the counter (maximum value 0x2) increases by 1. After that, the IO-0W[3.1.1.2] interface performs the appropriate operation.

Each full cycle is 1 game frame. The frame is updated when the register overflows. Each full cycle is 1 game frame. The frame is updated when the register overflows. To do this, the "Screen update"[3.1.3] module is used.

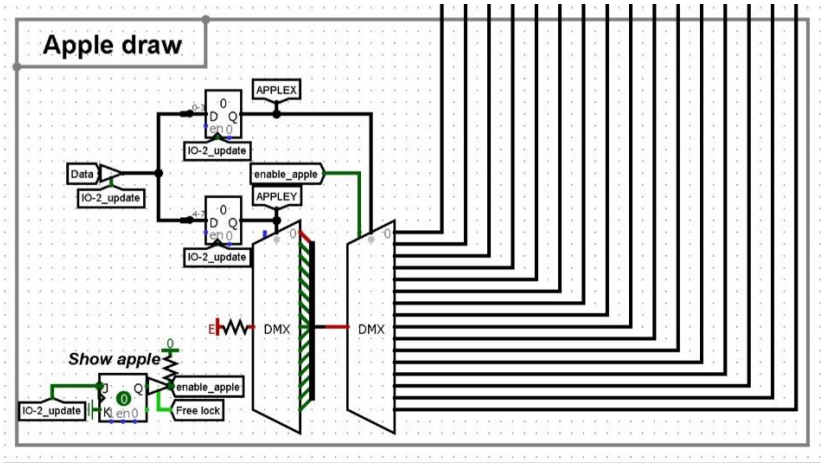
3.1.3 "Screen update"

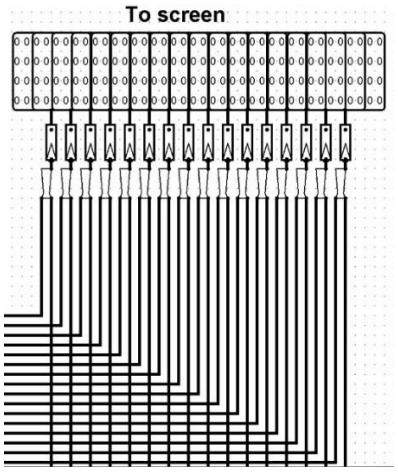


Pic.13 - the "Screen update" module

The "Screen update" module renders a new game frame. This happens when the signal drops on the T3 tunnel (the register in the "Selector"[3.1.2] module overflows) or when the IO-5W[3.1.1.4] interface is activated. The signal is transmitted to the zero port of the subcircuit "capacitorModule"[3.2].

3.1.4 "Apple draw"





Pic.14 Pic.15

Pic.14 – the "Apple draw" module

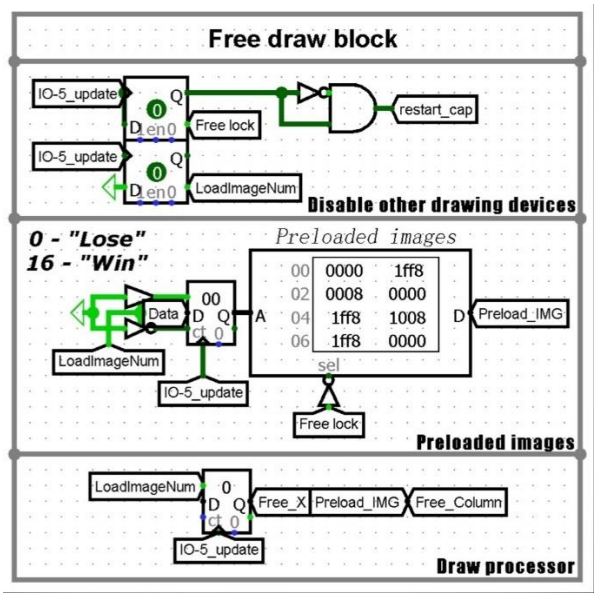
Pic.15 - ports To screen and "smallOR" subcircuits

The "Apple draw" module draws apples on the playing field.

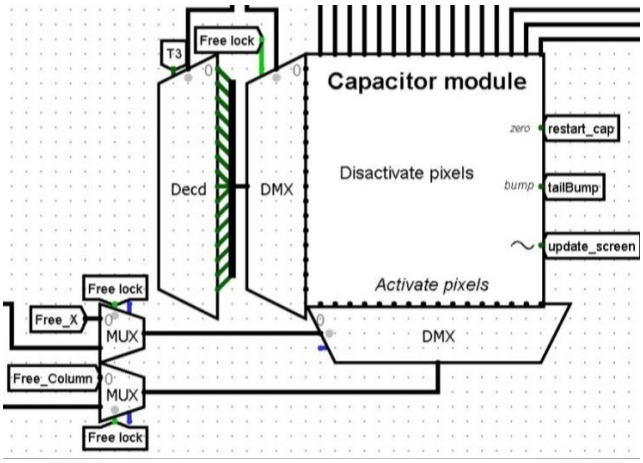
This module takes an 8-bit value as input. 0-3 bits are the X coordinate, 4-7 bits are the Y coordinate. These coordinates are stored in registers. After that, an error signal is directly transmitted to the LED matrix at these coordinates. These coordinates are the keys for demultiplexers. The first demultiplexer gives the error signal is filled with the number Y (a column with an apple for the LED matrix is formed). The second demultiplexer transmits this column to the To screen port number X. Before being transmitted to the LED matrix, the signal from the "Apple draw" module and the signal from the "capacitorModule"[3.2] subcircuit pass through the "smallOR" subcircuit (a reduced version of the OR gate for 16-bit variables). Because of this, if a snake and an apple are found in the same place on the playing field, a snake will be displayed.

The J-K Flip-Flops blocks the rendering of apples before the game starts. After the IO-2W[3.1.1.3] interface is activated for the first time, the apple starts to be displayed. When the IO-5W[3.1.1.4] interface is activated, the display of apples is disabled.

3.1.5 "Free draw block"



Pic.16 – the "Free draw block" module



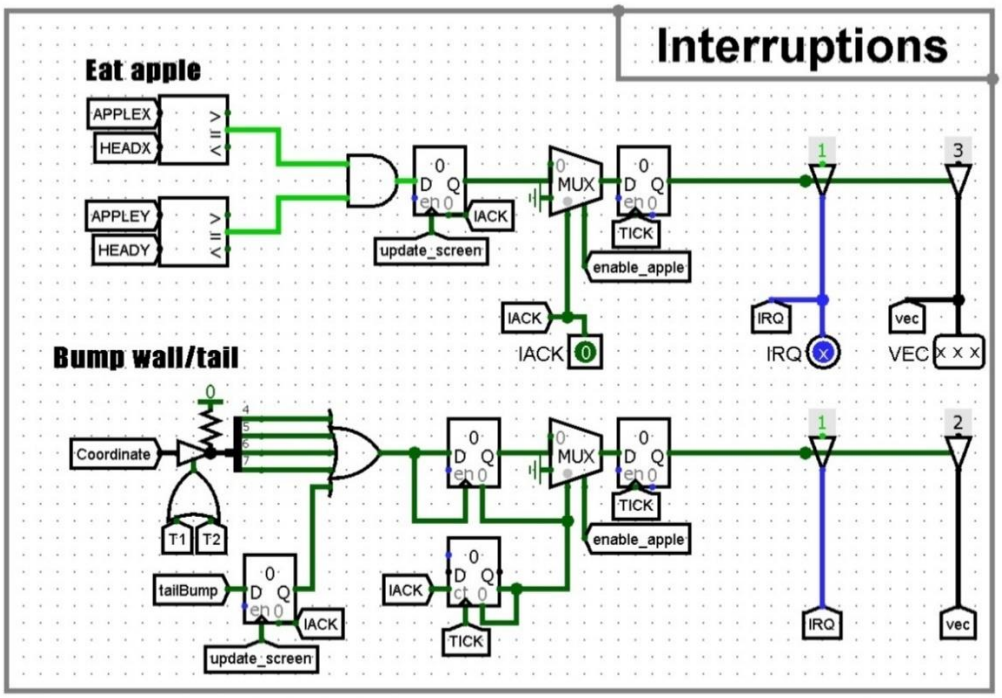
Pic.17 - the scheme of enabling and disabling pixels

The "Free draw block" module renders the "WIN" screen or the "LOSE" screen if the game has ended.

This module is activated if the IO-5W[3.1.1.4] interface is activated.

When the processor first accesses the IO-5W[3.1.1.4] interface, all other modules are disconnected from the screen, and the screen is also cleared. To clear the screen, a signal is sent to the zero port of the subcircuit "capacitorModule"[3.2]. In addition, the image number is loaded from DATA (0 is the "LOSE" screen, 16 is the "WIN" screen). After that, loading the image number becomes unavailable. These two operations are implemented using D Flip-Flops. Then this number is loaded into the counter. After that, in 16 cycles, the values of the columns of the LED matrix are loaded from the "Preloaded images" ROM module. These values are transmitted by column number. This number is calculated using the "Draw processor" counter.

3.1.6 "Interruptions" *Available only in the hardware version



Pic.18 - the "Interruptions" module

This module implements apple eating interruption and bumping interruption.

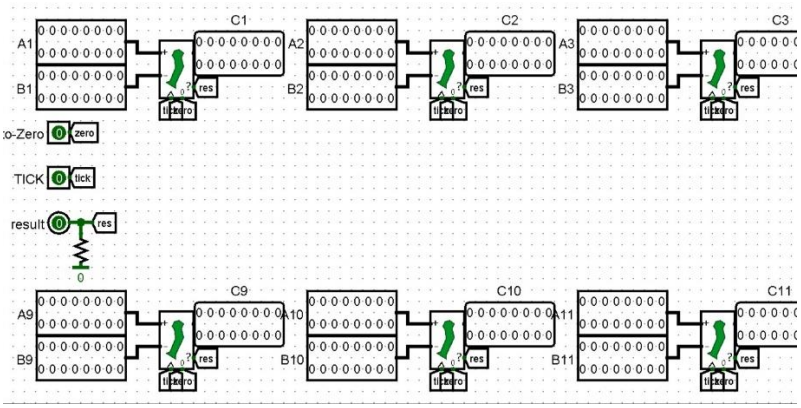
- Eat apple

If, after executing the cycle in the IO-0W[3.1.1.2] interface, the coordinates of the apple and the snake's head coincide, then the "Interruptions" module sends an interrupt request to the processor through the IRQ and VEC ports (IRQ = 1, VEC = 011). After that, if the processor has confirmed the interruption using IACK port, the interruption in "Interruptions" module is terminated (the data in the registers are reset).

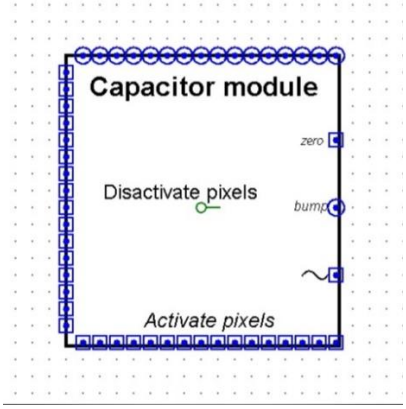
- Bump wall/tail

If the "capacitorModule"[3.2] circuit outputs a value of 1 to the bump port (the snake bumped into snake' s tail), then a bump interruption is sent to the processor. Also, if during the execution of the first two actions of the cycle in the IO-0W[3.1.1.2] interface, the DATA port received data in which the values of bits 4-7 are not equal to 0, then the "Interruptions" module activates the bump interruption (the processor is trying to process movement outside the playing field). The "Interruptions" module sends an interruption request to the processor through the IRQ and VEC ports (IRQ = 1, VEC = 010). After that, if the processor has confirmed the interruption using IACK port, the interruption in "Interruptions" module is terminated (the data in the registers are reset).

3.2 "capacitorModule"



Pic.19



Pic.20

Pic.19 - the "capacitorModule" circuit (Back-end)

Pic.20 - the "capacitorModule" circuit (Front-end)

The "capacitorModule" circuit stores 16-bit signals for each column of the LED matrix. The LEDs remain enabled if the value in "capacitorModule" has not been changed to 0.

Input ports:

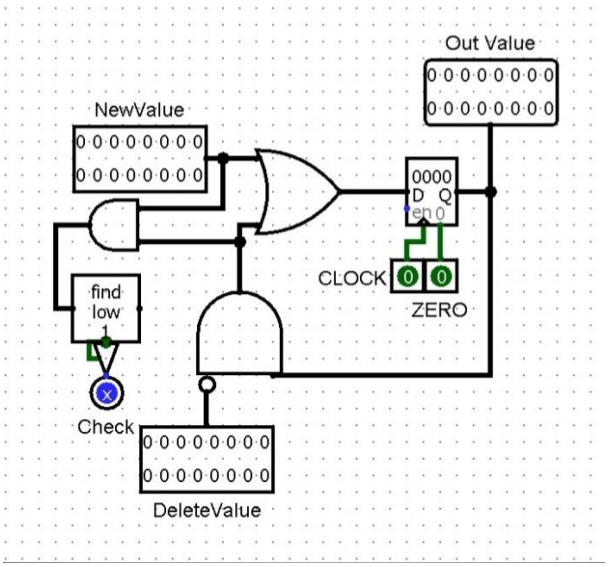
- 16 ports for pixel activation (Activate pixels/A1-A16, 16-bit)
- 16 ports for deactivating pixels (Disactivate pixels/B1-B16, 16-bit)
- Port for screen refresh signal (~/TICK, 1-bit)
- Port for the screen reset signal (zero/to-Zero, 1-bit)

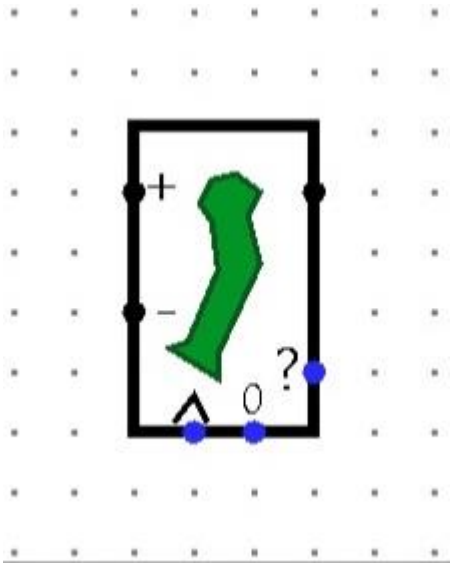
Output ports:

- 16 signal output ports for matrix columns (C1-C16, 16-bit)
- Port for signal of bump into tail (bump/result, 1-bit) *Available only in the hardware version

This scheme uses 16 subcircuits "capacitorElement"[3.2.1]. Ports A, B and C with the same number are connected to each subcircuit. Also, to-Zero, TICK and result ports are connected to all subcircuits.

3.2.1 "capacitorElement"





Pic.21 Pic.22

Pic.21 - the "capacitorElement" circuit (Back-end)

Pic.22 - the "capacitorElement" circuit (Front-end)

The "capacitorElement" circuit stores and changes the data about the included LEDs in the column of the LED matrix.

Input ports:

- Port for enabling pixels (NewValue/+, 16-bit)
- Port for disabling pixels (DeleteValue/-, 16-bit)
- Port for register update signal (CLOCK/^, 1-bit)
- Port for register reset signal (ZERO/0, 1-bit)

Output ports:

- Register output port (Out Value, 16-bit)
- Port for signal of bump into tail (Check/?, 1-bit) *Available only in the hardware version

First, the register value and the inverted value of the DeleteValue port are processed by the AND gate. So, the pixels in the column of the LED matrix are turned off.

If the result of the previous operation and the value of the newValue port have common enabled bits, then a signal 1 is sent to the Check port. Thus, a tail bumping check is implemented using the Bit Finder.

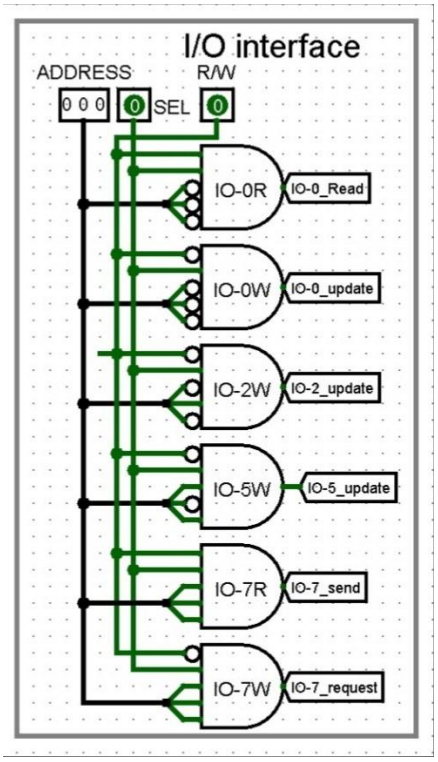
After that, the result of disabling pixels and the value from the newValue port are processed by the OR gate and the result is stored in a register. Saving occurs if the value 1 was got from the CLOCK port.

The value from the register is continuously sent to the Out Value port.

3.3 Bot version

For the bot to work correctly, the bot must be able to check the coordinates near the head of its snake. To do this, the "BotEyes"[3.3.2] subcircuit is used in the "displayControllerChip"[3.1] circuit. This subcircuit is connected to the "capacitorModule"[3.2] subcircuit.

3.3.1 Interfaces



Pic.23 - the "I/O interface" module (Bot version)

The "BotEyes"[3.3.2] circuit is controlled using the IO-7R[3.3.1] and IO-7W[3.3.2] interfaces in "displayControllerChip"[3.1].

3.3.1.1 IO-7R

The IO-7R interface is used to load the X and Y coordinates into the "BotEyes"[3.3.2] subcircuit.
This interface is activated if:

- R = 1
- R/W = 1
- ADDRESS = 111

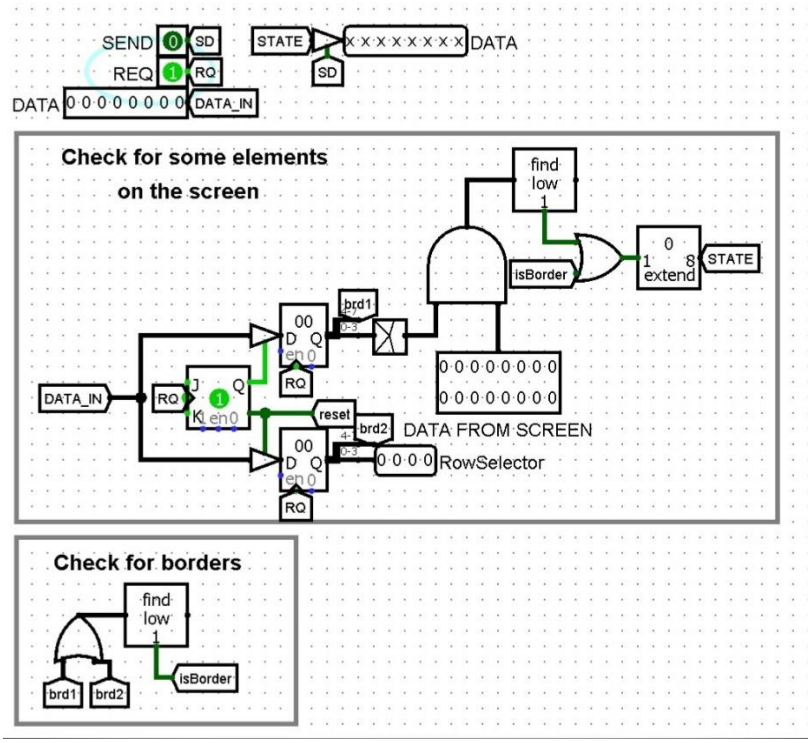
3.3.1.2 IO-7W

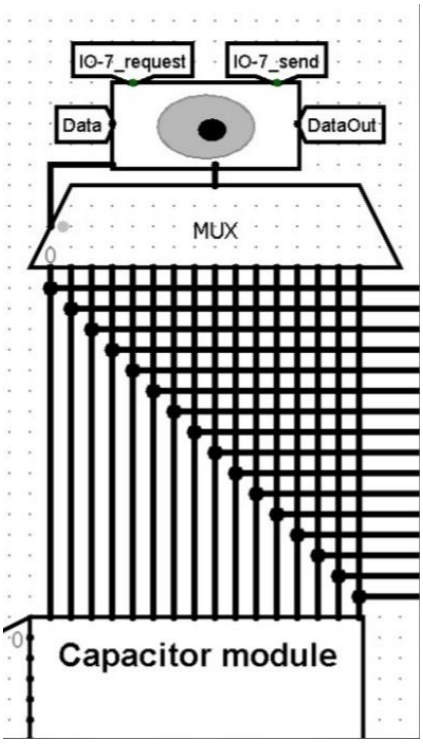
The IO-7W interface is used to download the check result from the "BotEyes"[3.3.2] subcircuit and send it to the DATA output port.

This interface is activated if:

- R = 1
- R/W = 0
- ADDRESS = 111

3.3.2 "BotEyes"





Pic.24 Pic.25

Pic.24 - the "BotEyes" circuit (Back-end)

Pic.25 - the "BotEyes" circuit (Front-end)

Input ports:

- Data port (DATA, 8-bit)
- Port for the read mode activation signal (REQ, 1-bit)
- Port for the write mode activation signal (SEND, 1-bit)
- Port for the data from "capacitorModule"[3.2](DATA FROM SCREEN, 16-bit)

Output ports:

- Data port (DATA, 8-bit)
- Port for the column number of the playing field (RowSelector, 4-bit)

First, the "BotEyes" circuit takes the Y coordinate from the DATA input port and stores it into a register and 0-3 bits is processed using the "converter coords (4bit->16bit)"[3.3.2.1] subcircuit. This happens if the IO-7R interface has been activated.

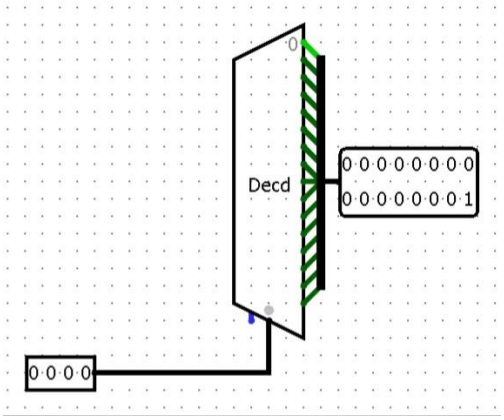
After that, the "BotEyes" circuit takes the X coordinate from the DATA input port and stores it in the register. 0-3 bits is output to the RowSelector port and used as a key for the multiplexer. This happens if the IO-7R interface has been activated.

After that, the value from the DATA FROM SCREEN port and the output value from the "converter coords (4bit->16bit)"[3.3.2.1] subcircuit are checked for the presence of paired bits equal to 1. This means that the coordinates are outside the playing field.

If 4-7 bits of the coordinates from the registers are not equal to 0, then the "Check for borders" module outputs a value equal to 1.

If at least one of the two checks (0-3 bits check or 4-7 bits check) outputs a value equal to 1, then the "BotEyes" circuit outputs a value equal to 1 to the outgoing DATA port using bit extender. The IO-7W interface is used for this.

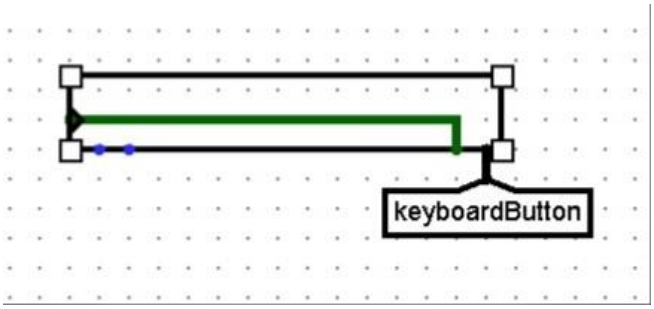
3.3.2.1 "converter coords (4bit->16bit)"



Pic.26 - the "converter cords (4bit->16bit)" circuit

This module takes the number N and outputs a 16-bit value in which the bit with the number N is 1.

4 Motion control



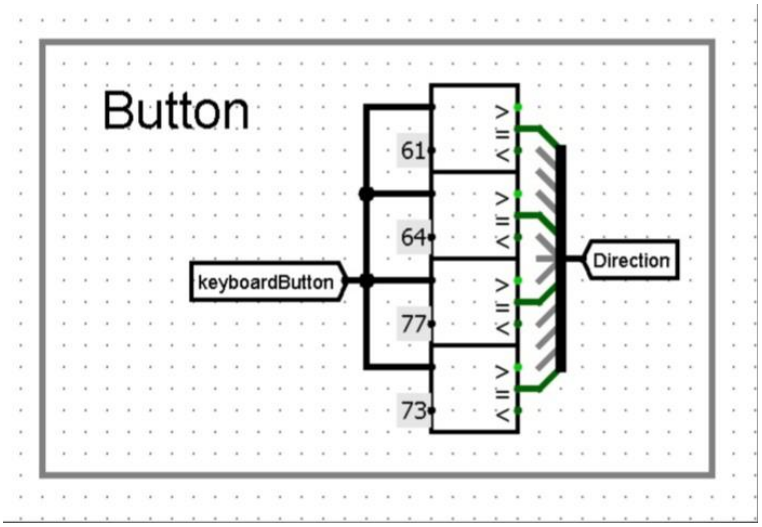
Pic.27 – the keyboard element

The keyboard is used to control the snake. The keyboard is connected to the "keyboardChip"[4.2] circuit using the "Button"[4.1] module.

The Available port and the Clock port on the keyboard are connected in order to reset the value in the keyboard buffer after pressing.

The control works only if the English keyboard layout is enabled and Caps Lock is turned off.

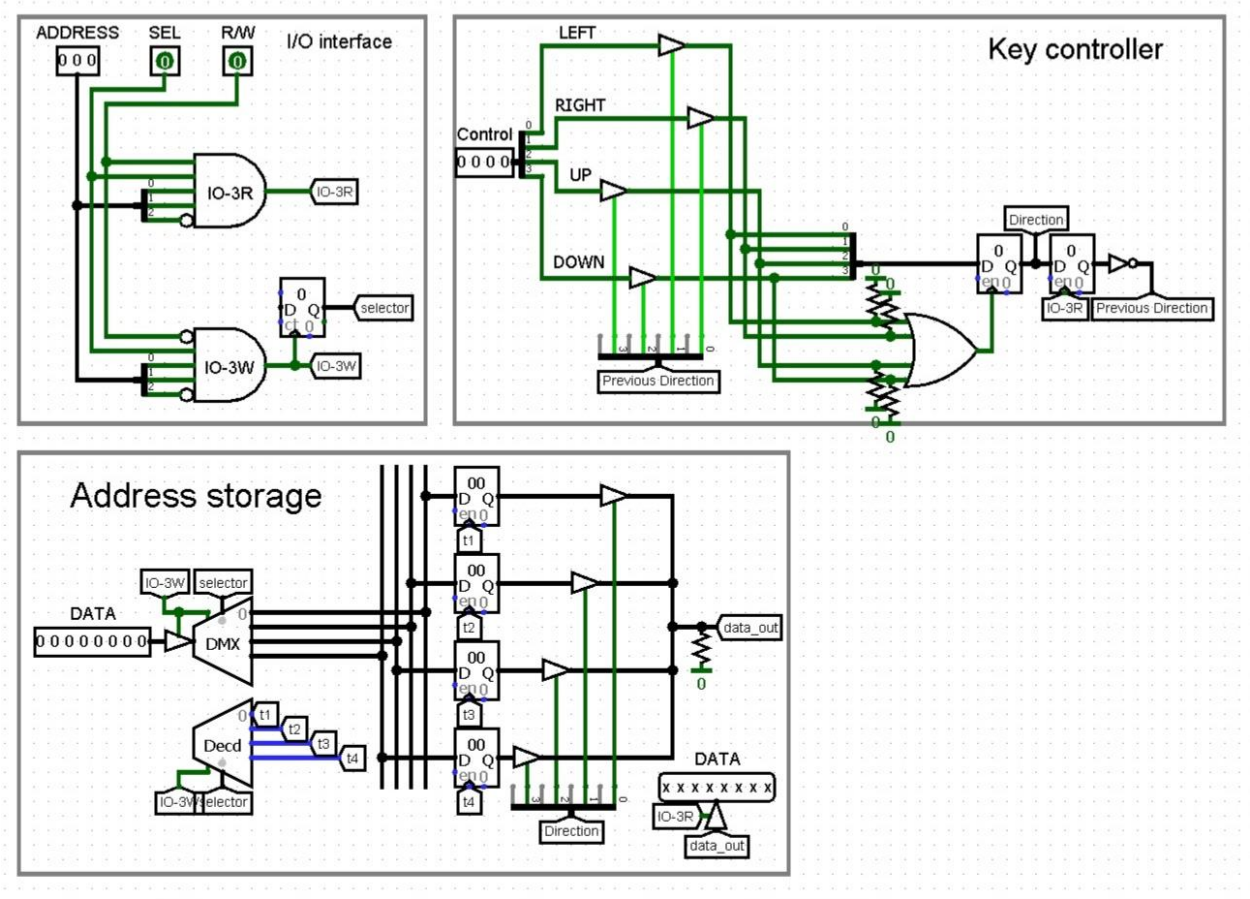
4.1 "Button"



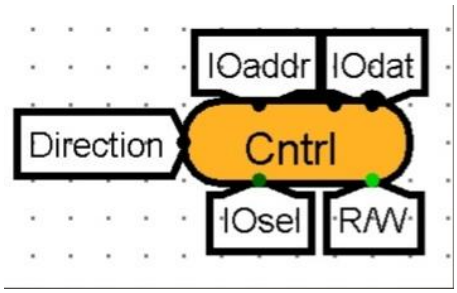
Pic.28 - the "Button" module

The "Button" module converts the ASCII code from the keyboard to a 4-bit value and transmits this value to the Control port of the "keyboardChip"[4.2] circuit. This module recognizes only ASCII codes of the symbols "w", "a", "s", "d".

4.2 "keyboardChip"



Pic.29 - the "keyboardChip" circuit (Back-end)



Pic.30 - the "keyboardChip" circuit (Front-end)

The "keyboardChip" circuit accepts and stores the addresses of the commands responsible for changing the direction of movement. Also, this circuit takes the direction of movement from the Control port and sends the address of the corresponding command to the processor.

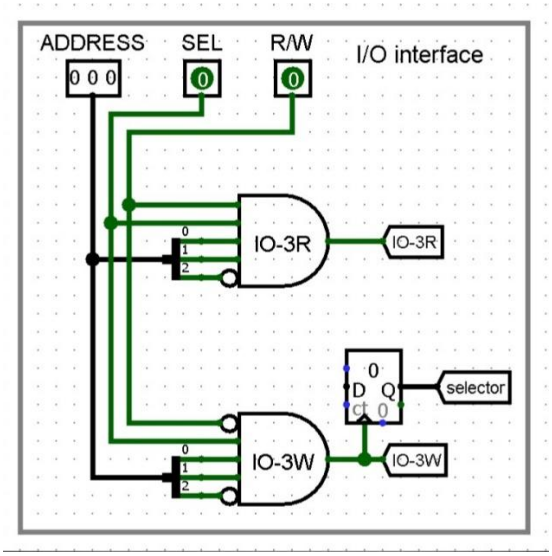
Input ports:

- Device address port (ADDRESS, 3-bit)
- Port for external device activation signal (SEL, 1-bit)
- Command address port (DATA, 8-bit)
- Port for signal Reading/Recording (R/W, 1-bit)
- Movement direction data port (Control, 4-bit)

Output ports:

- Command address port (DATA, 8-bit)

4.2.1 "I/O interface"



Pic.31 - the "I/O interface" module ("keyboardChip")

To start working with "keyboardChip"[4.2], you need to send a signal to the SEL port, transmit the command address to the ADDRESS port and select the read mode or write mode using the R/W port. After that, the "I/O interface" module activates the corresponding interface.

4.2.1.1 IO-3R

The IO-3R interface sends the command address for the processor from the "Address storage"[4.2.3] module to the output DATA port.

Also, this interface updates the value in the second register (the previous direction of movement).

This interface is activated if:

- SEL = 1
- R/W = 0
- ADDRESS = 011

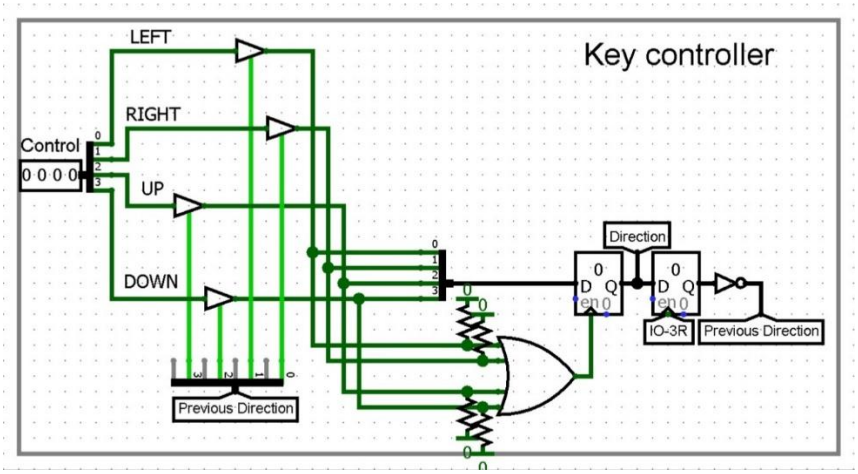
4.2.1.2 IO-3W

The IO-3W interface stores the addresses of movement commands for the processor in registers in the "Address storage"[4.2.3] module.

This interface is activated if:

- SEL = 1
- R/W = 1
- ADDRESS = 011

4.2.2 "Key controller"

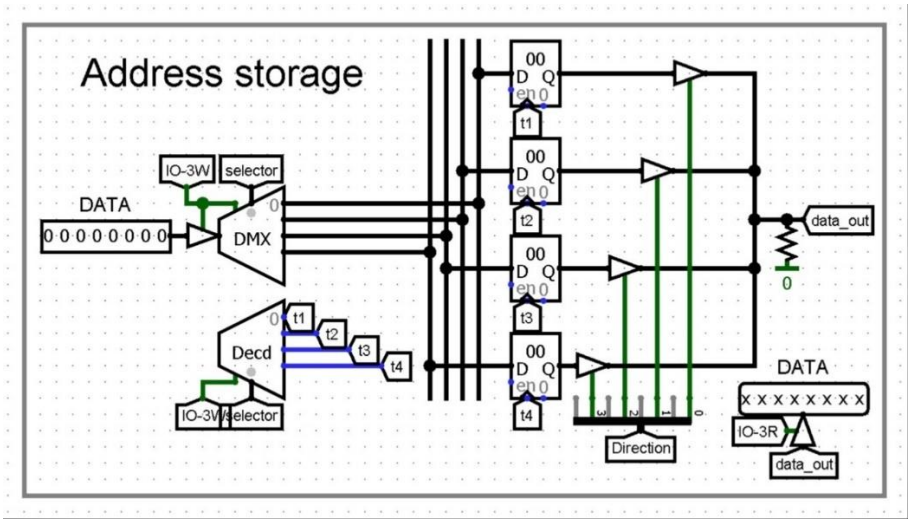


Pic.32 - the "Key controller" module

The "Key controller" module reads the value from the Control port and stores it in the first register. After that, the value from the register is transferred to the "Address storage"[4.2.3] module through the *Direction* tunnel and opens the corresponding controlled buffer for transmitting the address of the motion command to the processor.

After that, the value from the first register is transferred to the second register to check the next action for "movement into itself". The value of the second register is inverted and blocks one of the controlled buffers during the next activations of the "Key controller" module.

4.2.3 "Address storage"



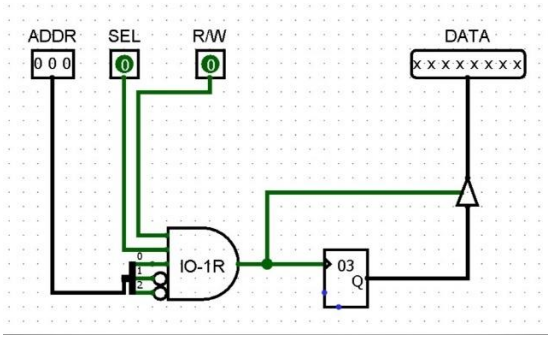
Pic.33 - the "Address storage" module

The "Address storage" module reads the value from the DATA port when the IO-3R[4.2.1.1] interface is active.

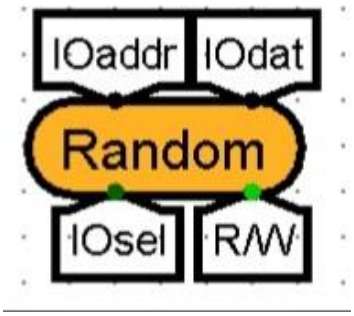
Then this value is stored in the corresponding register. The module operates in 4-clock mode, with further deactivation. The registers store the addresses by which the motion commands are stored in the processor memory.

If the IO-3W[4.2.1.2] interface is activated, the "Address storage" module transmits a value from a single register to the DATA port. To do this, the value from the "Key controller"[4.2.2] module opens one of the controlled buffers.

5 "randomGenChip"



Pic.34 - the "randomGenChip" circuit (Back-end)



Pic.35 - the "randomGenChip" circuit (Front-end)

The "randomGenChip" scheme is used for random generation of numbers. The IO-1R[5.1] interface is used for this.

Input ports:

- Device address port (ADDRESS, 3-bit)
- Port for external device activation signal (SEL, 1-bit)
- Port for signal Reading/Recording (R/W, 1-bit)

Output ports:

- Data port (DATA, 8-bit)

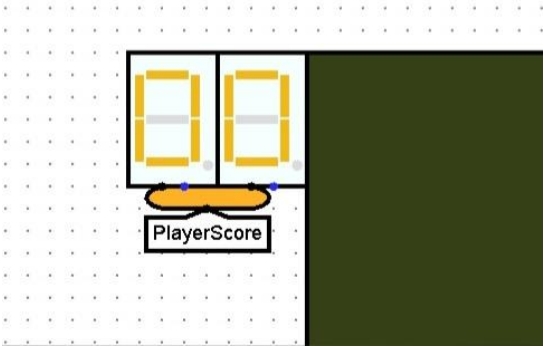
5.1 IO-1R

When the IO-1R interface is activated, the random number generator generates an 8-bit value that is output to the DATA port.

This interface is activated if:

- SEL = 1
- R/W = 1
- ADDRESS = 001

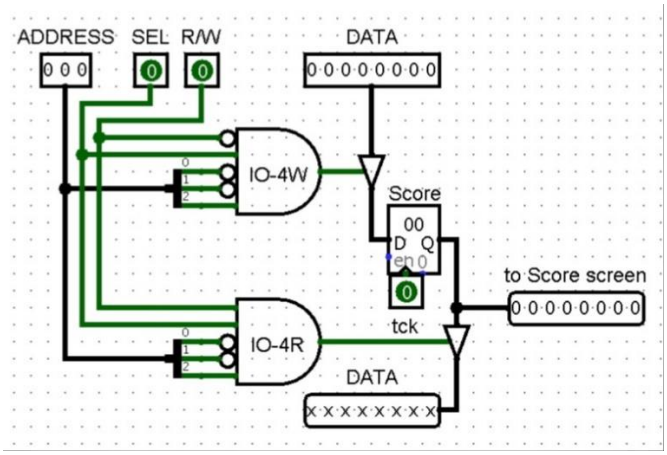
6 Displaying the number of points



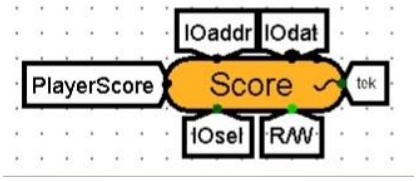
Pic.36 – the hexadecimal indicators

Two hexadecimal indicators are used to display the number of points scored. These indicators are connected to the "scoreChip"[6.1] circuit using the "ScoreDisplayDriver"[6.2] circuit.

6.1 "scoreChip"



Pic.37



Pic.38

Pic.37 - the "scoreChip" circuit (Back-end)

Pic.38 - the "scoreChip" circuit (Front-end)

The "scoreChip" circuit is used to store and display the number of points scored.

Input ports:

- Device address port (ADDRESS, 3-bit)
- Port for external device activation signal (SEL, 1-bit)
- Port for signal Reading/Recording (R/W, 1-bit)
- Data port (DATA, 8-bit)
- Clock generator port (tck/~, 1-bit)

Output ports:

- Data port (DATA, 8-bit)
- Port for connection to the indicator (to Score screen, 8-bits)

The "scoreChip" circuit has two interfaces: IO-4W[6.1.1] and IO-4R[6.1.2].

6.1.1 IO-4W

The IO-4W interface reads the number of points scored from the DATA port and stores this value in a register when the signal from the tck port is lowered. Values from the register are continuously output through the to Score screen port.

This interface is activated if:

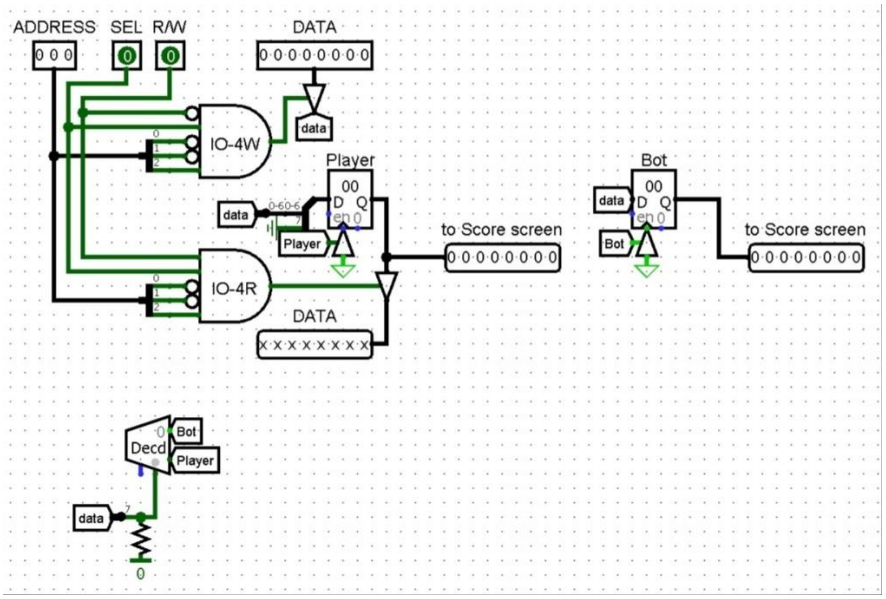
- SEL = 1
- R/W = 0
- ADDRESS = 100

6.1.2 IO-4R * This interface was used in previous versions. Currently, this interface is not used. The IO-4R interface opens a managed buffer and outputs the number of points scored from the register to the DATA port.

This interface is activated if:

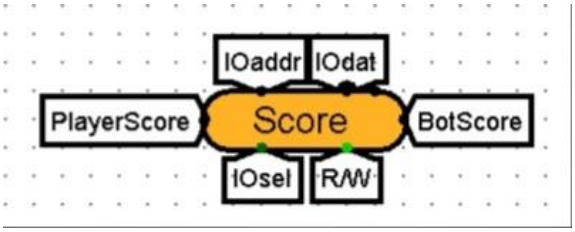
- SEL = 1
- R/W = 1
- ADDRESS = 100

6.1.3 Bot version



Pic.39

Pic.40



Pic.41

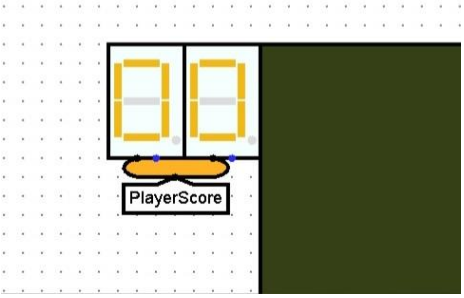
Pic.39 - the "scoreChip" circuit (Back-end, Bot version)

Pic.40 - the hexadecimal indicators (Bot version)

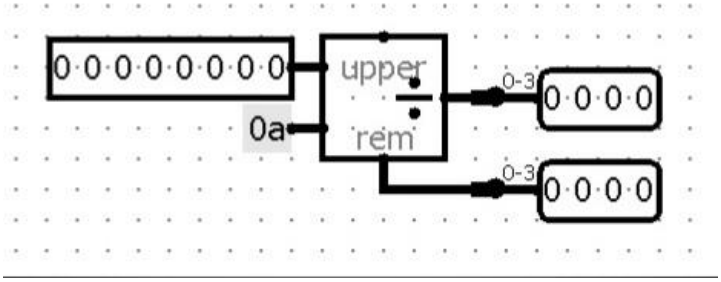
Pic.41 - the "BotEyes" circuit (Front-end, Bot version)

In the Bot version, the "scoreChip"[6.1] circuit has an additional register for storing the points scored by the bot. Also, in the circuit there is another to Score screen port to output values from this register. The seventh bit of data from the DATA input port is used to select the register. The value of this bit opens one of the controlled buffers using the decoder.

6.2 "ScoreDisplayDriver"



Pic.42



Pic.43

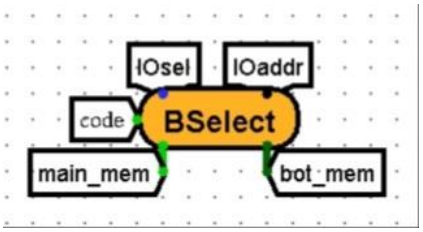
Pic.42 - the "ScoreDisplayDriver" circuit (Front-end)

Pic.43 - the "ScoreDisplayDriver" circuit (Back-end)

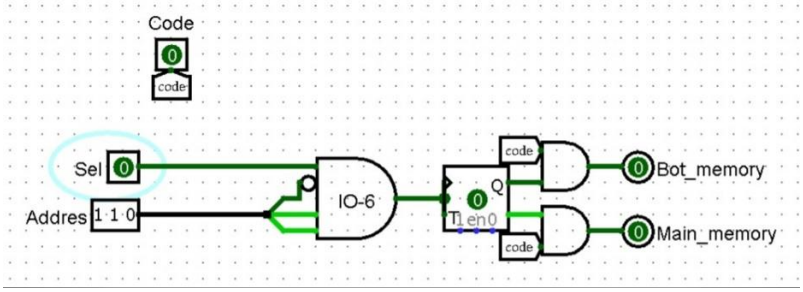
The "ScoreDisplayDriver" scheme outputs a number from the input port in decimal representation through two output ports.

The number from the input port is divided by 10 using the Divisor element. After that, the result of the division (left digit) is output to the upper output port, and the remainder of the division (right digit) is output to the lower output port.

7 "MemorySelector"



Pic.44



Pic.45

Pic.44 - the "MemorySelector" circuit (Front-end)

Pic.45 - the "MemorySelector" circuit (Back-end)

The "MemorySelector" scheme changes the connection to the ROM modules. The IO-6[7.1] interface is used for this.

Input ports:

- Device address port (ADDRESS, 3-bit)
- Port for external device activation signal (SEL, 1-bit)
- Port for the activating the connection to the code signal (Code, 1-bit)

Output ports:

- Main ROM module activation port (Main_memory, 1-bit)
- Bot ROM module activation port (Bot_memory, 1-bit)

7.1 IO-6

During each activation of the IO-6 interface, the T-trigger changes the active ROM module. This interface is activated if:

- SEL = 1
- ADDRESS = 110

8 Literature

- A.Shafarenko, S.P.Hunt. Computing platforms. School of Computer Science University of Hertfordshire, 2015. – 307 p.