FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION OF HIGHER EDUCATION "NOVOSIBIRSK NATIONAL RESEARCH STATE UNIVERSITY"

DEPARTMENT OF INFORMATION TECHNOLOGIES

Group Project A

on the discipline "Digital platforms"

“The Game of Noughts and Crosses”

The work was done by:
1st-year students
of the Department
of Information Technology
group 21216
Makhov Nikolay
Pogibelnaya Olga
Tatarintsev Vladislav

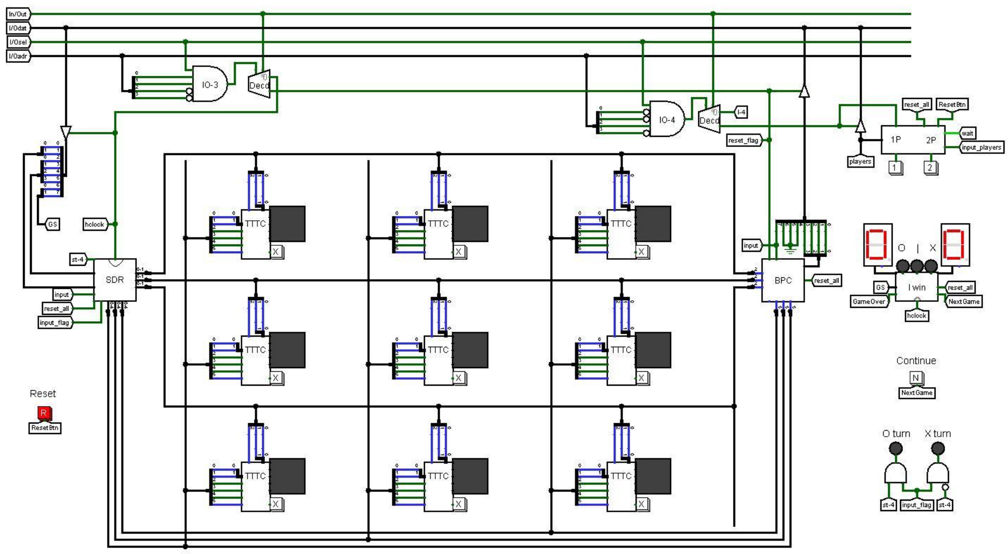
Novosibirsk, 2022

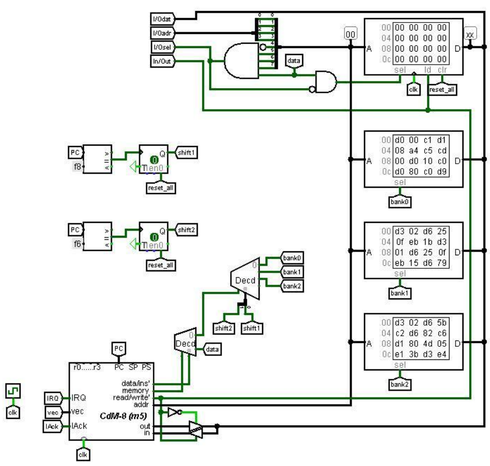
Table of contents

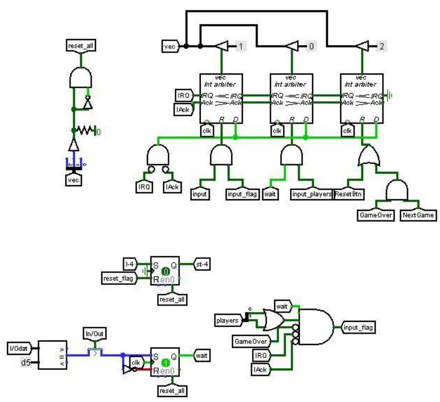
Introduction

Tic-tac-toe is a logical game between two players on a square field of 3 by 3 cells. Each of the players puts crosses or noughts so that it is possible to draw a line through 3 identical symbols. In our project, the game always starts with crosses. Why did we choose this project?

When we found out that we are on the same team, we distributed responsibilities according to the strengths of each of us. It turned out that Nikolai will be able to develop the hardware part, and Olga and Vlad are ready to write the software part of our work. It seemed to us that tic-tac-toe is a game where AI has more options than in any other games. It is more interesting because we should prescribe smart computer actions, while there is no such variety in other games.

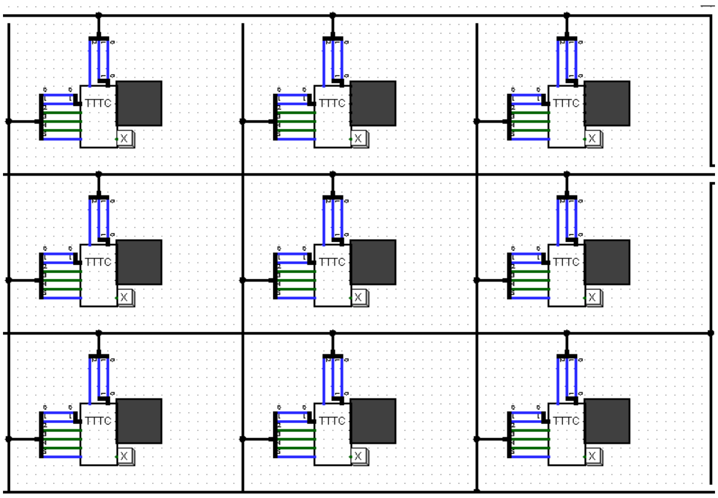


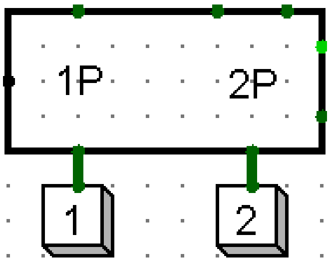




Gamepad

The main part of the interface of our game is the playing field, which consists of nine cells. Each cell is a black screen (matrix) in size 5x5. On these screens during the game, future crosses and noughts will appear.

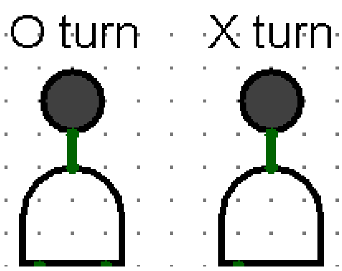


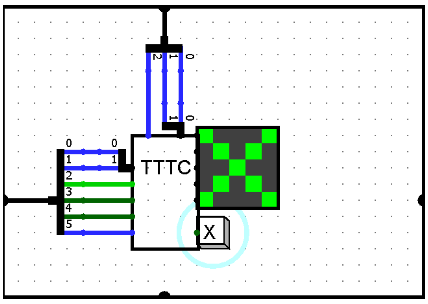


For more interest and variety, we implemented two game modes: a game with a computer and a game for two players.

Before starting the game, the user first needs to choose the game mode.

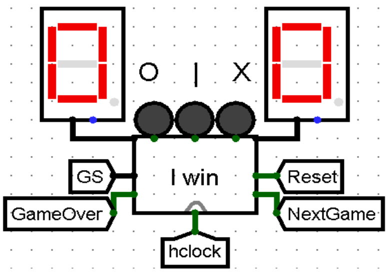
Next, the player must make his move. So that the player never forgets who should move at the moment, there are two special LEDs that are illuminated during the move of a certain symbol.





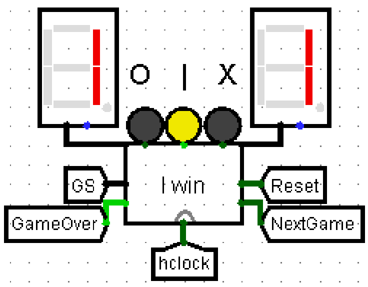
If the LED illuminates the cross, then the player can make his move by choosing the cell of the playing field and pressing the small white button below the screen belonging to this cell. The cross symbol will appear immediately after pressing.

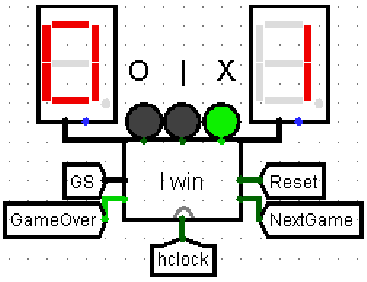
Then the computer (or second player) moves, then the user again, and so on. When there is a row or diagonal where there are three identical symbols, the game will end with the victory of one side and the defeat of the second. If there are no free cells left on the field, the game ends in a draw. To define the winner and keep the score, there is a special indicator that changes values depending on the outcome of the game (located to the right of the playing field).



There are three options for the outcome of the game:

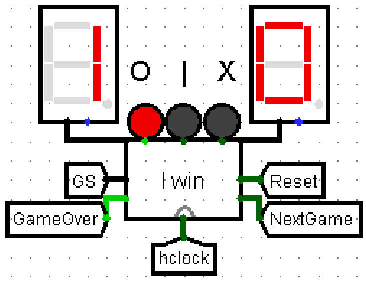
1) *Draw*. In this case, the indicator lights up yellow and each player gets one point.



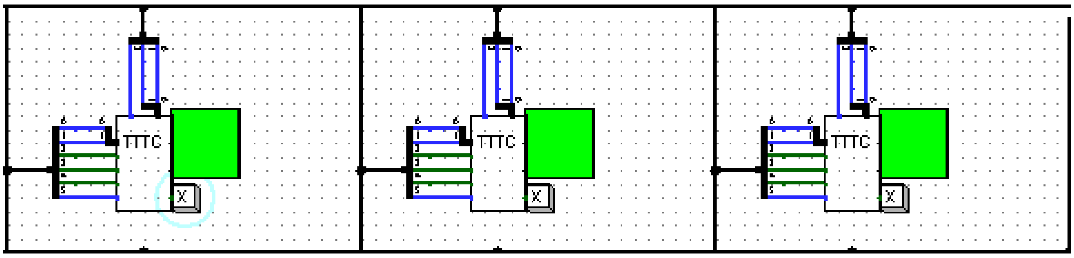


2) *Crosses win*. The indicator lights up green and the user playing as a cross gets one point.

3) *Noughts win*. The indicator lights up red and the computer (or the player playing for the noughts) gets one point.

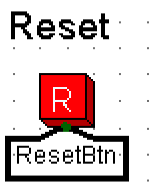


Also, after the victory of either side, the winning row lights up.



For the player to move on to the next game, he needs to press the "*Continue*" button located to the right of the playing field under the score indicator.

If the player wants to reset the score, he must press the red "*Reset*" button located to the left of the playing field. But in this case, he will have to choose the game mode again.

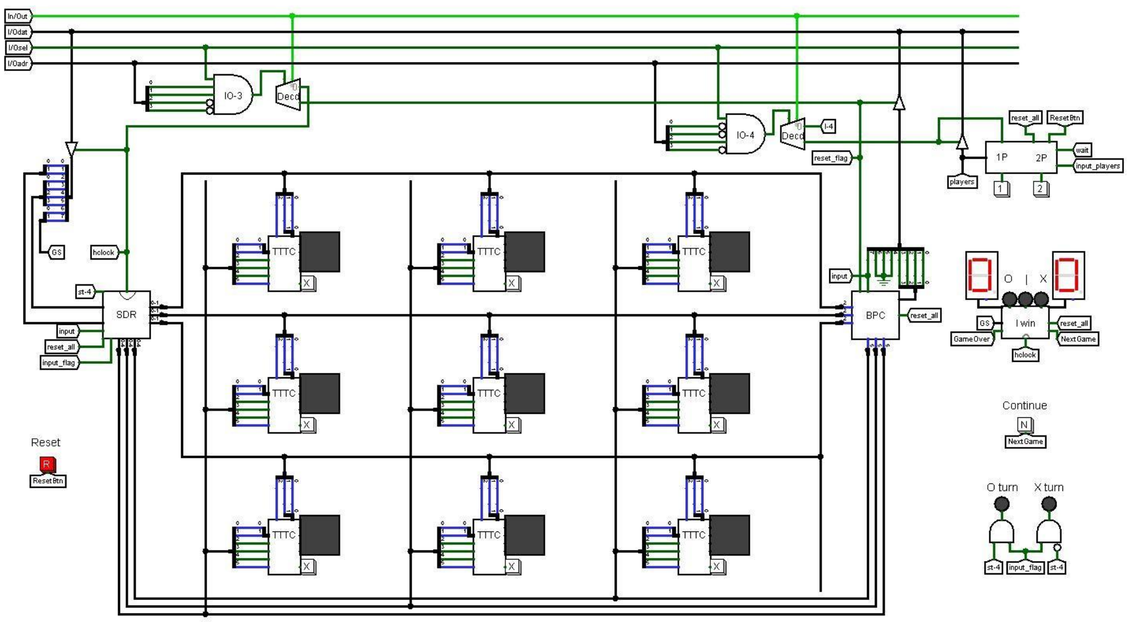


Hardware

Nikolay created this part of our project. The hardware part contains 13 chips (9✕TTTC, 1✕SDDR, 1✕BPC, 1✕GSDD, 1✕CofP), a CDM-8-mark5 processor and 3 interrupt arbitrators.

The main part of the circuit consists of two parts, the gamepad (which was partially described above) and the part in which the processor, interrupt arbitrators and memory banks are located.

Gamepad



Data from the CDM-8-mark 5 processor is transmitted to the eight-bit I/O data bus and each byte stored at address 0xE3 must contain the following bits:

Bits	Meaning
0-1	symbol ID
2-5	cell address
6-7	game status

Nothing is saved at address 0xE4, but the entry at this address is used as a flag indicating that the next character entered by the player will be a nought (used only when playing together).

Symbol ID	Meaning
0b00	an empty cell
0b01	a nought
0b10	a cross (but since the user always puts a cross and pressing the button processed without the participation of the processor, this ID is not used)
0b11	a completely painted cell (used when marking the winning line)

Cell address consists of the x and y coordinates of the cell in the yyxx format. For example, 0b0000 is the upper left cell, and 0b1010 is the lower right, 0b0010 is the upper right. This format is chosen for the convenient location of the playing field in RAM.

Game status	Meaning
0b00	the game has not ended yet
0b01	victory of noughts
0b10	victory of crosses
0b11	draw

The data transmitted from the I/O data bus to the processor when loaded from address 0xE3 contains the following bits:

Bits	Meaning
0-3	coordinates of the pressed button
4-6	nothing
7	flag of button pressing

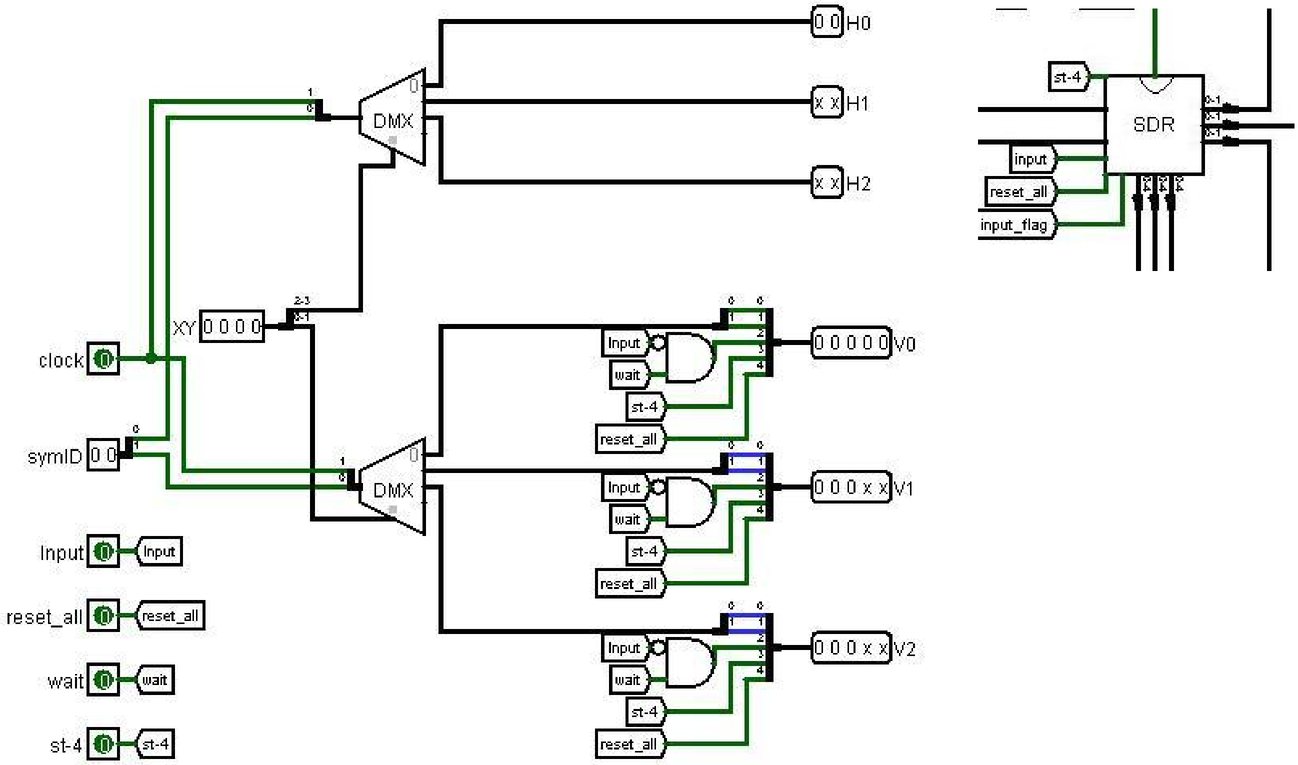
When loading from address 0xE4:

Bits	Meaning
0-1	count of players
2-7	nothing

Description of chips:

Chip	Purpose
SDR	used to transfer the symbol ID to the correct cell
TTTC	they are responsible for setting the characters entered by the player or computer
BPC	transmits to the I/O data the address of the button that the player pressed
GSDD	used to display the score
CofP	sends to the I/O data the number of players that the player has selected

SDR (Symbol Display Router)

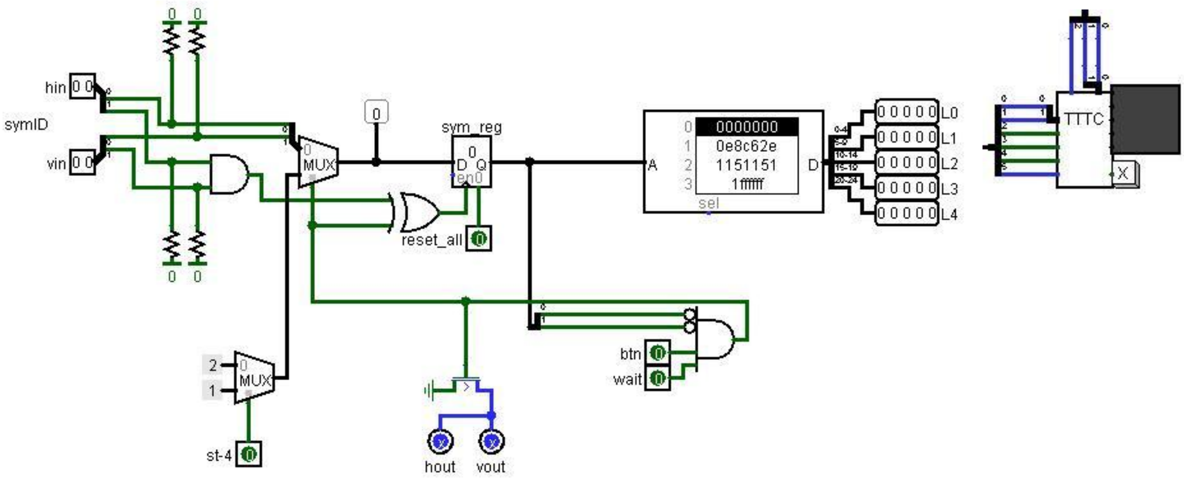


Inputs	Meaning
XY	cell address
symID(2)	symbol ID
clock	high when the computer puts the symbol
input	high when the player has already entered the symbol during his turn
wait	high while waiting for the player's move
st-4	the signal whose rise indicates that the player will now enter a nought
reset_all	pulse when the processor provides an interrupt with vector 2

Outputs	Meaning
H0-H2(2)	horizontal data buses
V0-V2(5)	vertical data buses

Depending on the cell address, it transmits a signal via one of 3 horizontal and one of 3 vertical buses. The zero-bit symbol ID and clock are transmitted over the horizontal bus. The first bit of *symID*, *clock*, *wait* and not *Input*, *st-4*, *reset_all* is transmitted over the vertical bus.

TTTC (Tic-Tac-Toe Cell)



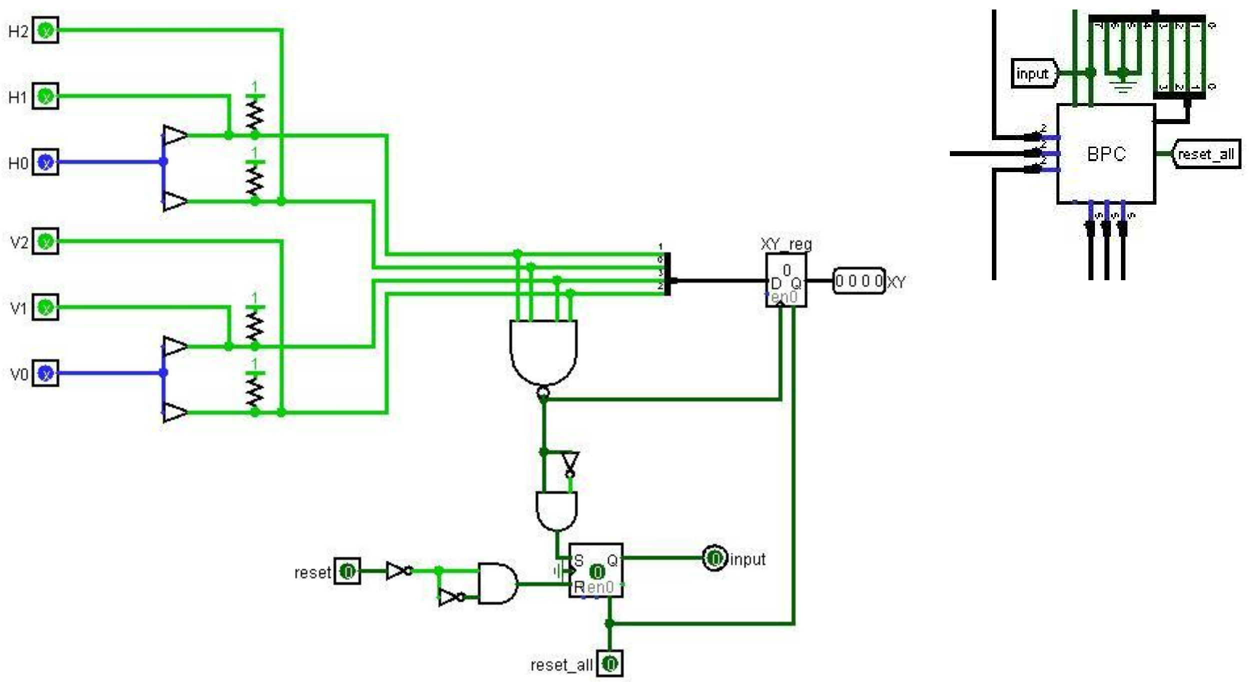
Inputs	Meaning
hin(2)	0 bit of symbol ID and clock
vin(2)	1 bit of symbol ID and clock
st-4	the signal whose rise indicates that the player will now enter a nought
wait	high while waiting for the player's move and the player hasn't moved yet
btn	high when the button is pressed
reset_all	pulse when the processor provides an interrupt with vector 2

Outputs	Meaning
L0-L4(5)	contain rows codes that should be displayed on the LED matrix
hout	horizontal button press bus
vout	vertical button press bus

If the first bit (clock) is high on the vertical and horizontal bus, then the symbol ID (zero bit *hin* and *vin*) is written to the register, which is the address in the ROM in which the input data for the LED matrix is recorded.

If the user pressed the button during his turn and there was nothing in this cell before, then 0 is transmitted to *hout* and *vout* (which are processed by the BPC chip), a nought or a cross is written to the register (depending on the *st-4* flag) and output to the LED matrix.

BPC (Button Press Capture)

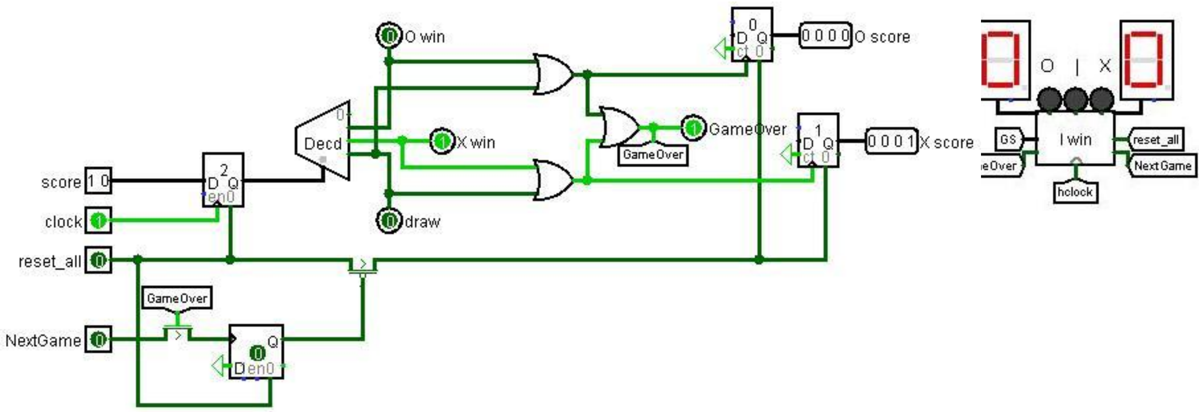


Inputs	Meaning
H0-H2	horizontal button press busses
V0-V2	vertical button press busses
reset	high when the processor reads XY
reset_all	pulse when the processor provides an interrupt with vector 2

Outputs	Meaning
XY(4)	cell address
input	high when the player has already entered the symbol during his turn

When one of the signals on the horizontal and one of the signals on the vertical bus becomes low, the address of the pressed button is transferred to the register and the *input* flag is raised. When the processor reads this address, the *input* flag is lowered.

GSDD (Game State Display Driver)



Inputs	Meaning
score(2)	game status
clock	high when the processor saves at 0xE3
reset_all	pulse when the processor provides an interrupt with vector 2
NextGame	high when the button Continue is pressed

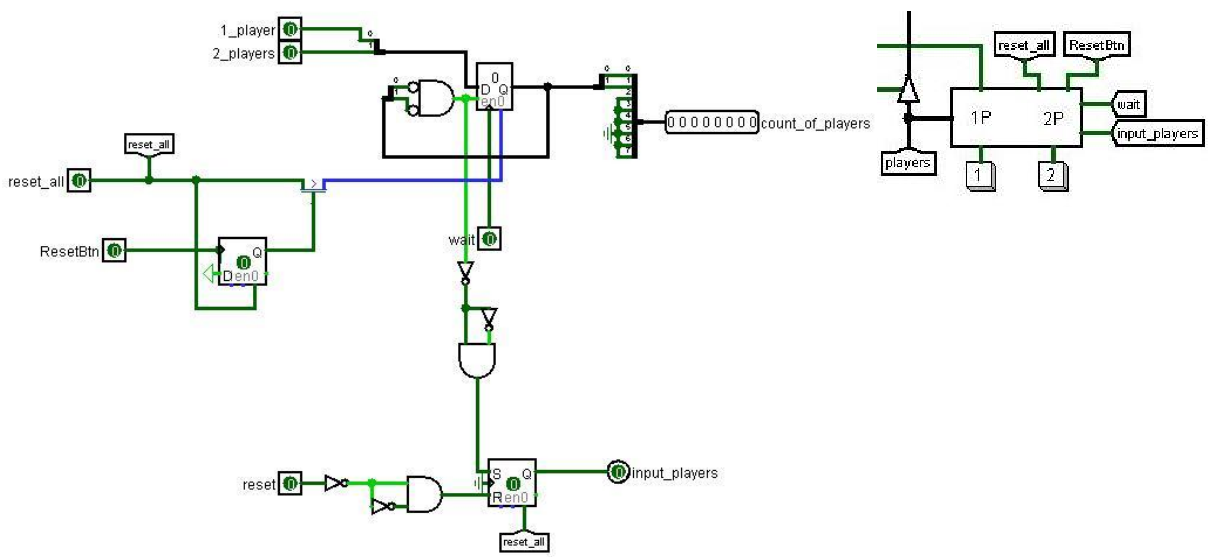
Outputs	Meaning
O win	high when O wins
X win	high when X wins
draw	high when draw
GameOver	high when the game ends
O score(4)	O score
X score(4)	X score

Receives 6-7 bits from the I/O data bus and, depending on them, adds a point to one of the players (in case of a draw, a point is added to both players). Also, when one of the players wins or draws, the corresponding LED lights up.

When the Continue button is pressed, the count is not reset, only the LEDs are extinguished.

And the reset button resets the account and everything else.

CofP (Count of Players)



Inputs	Meaning
1_player	high when the 1P button is pressed
2_players	high when the 2P button is pressed
wait	high when the processor executes the wait command
reset	high when the processor reads the count of players
reset_all	pulse when the processor provides an interrupt with vector 2
ResetBtn	high when the button Reset is pressed

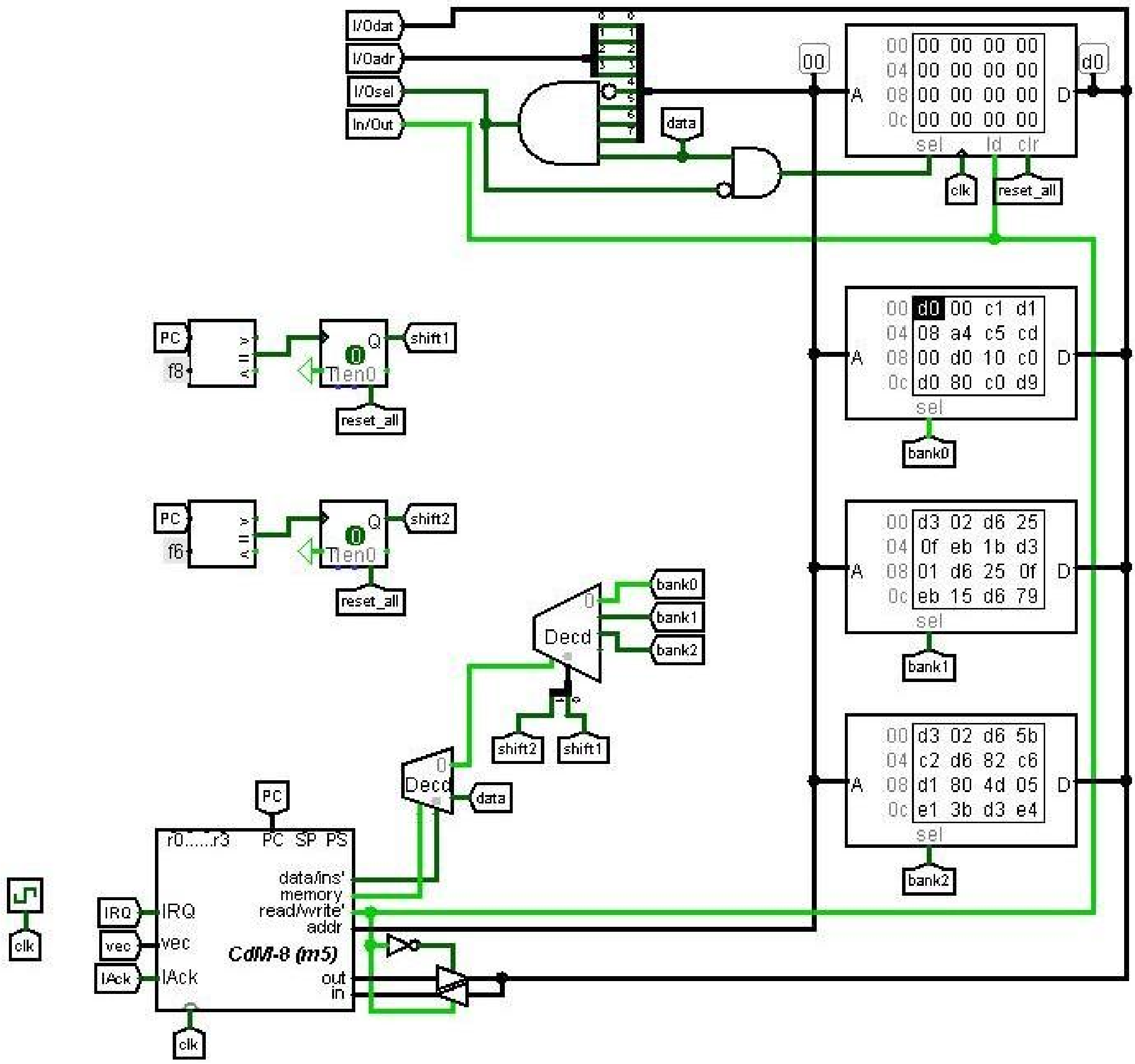
Outputs	Meaning
count_of_players(8)	user-selected number of players
input_players	high when the user has selected a number of players, but the processor has not received them yet

When one of the buttons (*1_player* or *2_players*) is pressed at the beginning of the first game, the code with the number of players is loaded into the register, and then transferred to 0-1 bits of I/O data (read when reading from the address 0xE4). When the register already has information about the number of players, the user will be able to change it only after pressing the Reset button.

0b01 - 1 player
0b10 - 2 players

The number of players is reset only when *ResetBtn* is pressed, and when the game continues, it remains the same.

Banks of data



The architecture of the game on the principle of memory sharing is based on the Harvard architecture (code and data are located separately). 3 memory banks are used.

- bank 0 - Contains the main loop for playing against the computer
- bank1 - Contains AI for the computer
- bank 2 - Contains the main loop for playing together

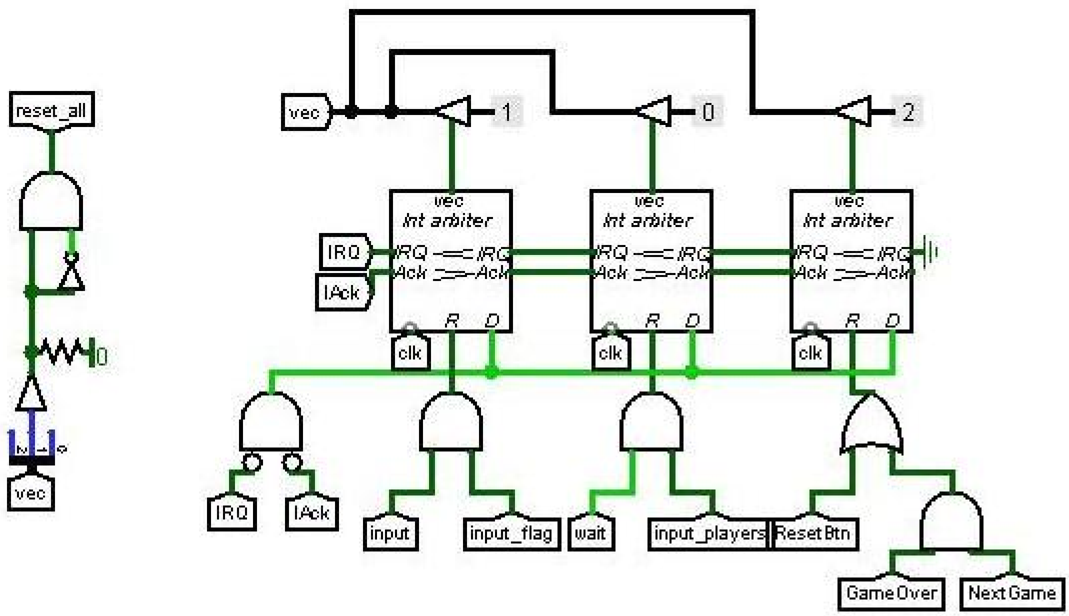
Switching between banks is implemented simply:

If the PC is equal to 0xF8, then there is a switch from bank0 to bank1 and also back.

If the PC is equal to 0xF6, then there is a switch from bank0 to bank2 and also back.

Since 0xE3 and 0xE4 are I/O addresses, writing and reading from RAM at addresses 0xE* is limited.

Interrupt arbiters



The scheme contains 3 interrupt arbitrators:

- Interrupt arbiter with vector 0 is responsible for interrupts when a player presses a button that determines the number of players. The interrupt request is sent only when the wait command is executed and the button is pressed.

- Interrupt arbiter with vector 1 is responsible for interrupts when the player presses the symbol setting button. The interrupt request is sent only when the corresponding button is pressed and high input_flag (wait, the number of players is selected and the game is not over yet.

- Interrupt arbiter with vector 2 is responsible for interrupts when the player presses the button Reset. The interrupt request is sent either when the Reset button is pressed, or at the end of the game when the Continue button is pressed.

When the processor provides an interrupt with vector 2, the reset_all signal pulse occurs, which resets all registers and triggers.

Software

Olga and Vladislav created the software part of our project. Our program consists of three parts: the main part, artificial intelligence, and a program for playing two participants.

Below, as you can see, there is the array of the playing field:

0	1	2	3
4	5	6	7
8	9	10	

The gray color shows the indexes of the cells according to their location on the playing field. Cells 3 and 7 also exist in the array in RAM, but they are not available for filling.

It should be noted that the cell where all input and output data are fed is the cell with the address 0xE3.

Main part

```
inputs>
table: dc 0,1,2
       dc 4,5,6
       dc 8,9,10
       dc 0,4,8
       dc 1,5,9
       dc 2,6,10
       dc 0,5,10
       dc 8,5,2
   end
```

For this part of our program, we needed the following input. This is a *table* constant that contains 8 rows. Each row contains array indexes that denote all possible winning combinations. So, if you want to win, you need to fill one of these rows with the same symbols.

What does the program consist of?

```
asect 0xf0 # vec 0
dc CountPlayers # 0x10
dc 0x80

asect 0xf2 # vec 1
dc GamepadBtn # 0x84
dc 0x80

asect 0xf4 # vec 2
dc 0x00
dc 0x00
```

At the beginning of the program, there are interrupt service routines.

The reset interrupt service routine (vec 2) is located in all memory banks.

The service routine for interrupting buttons on the gamepad (vec 1) is still in the memory bank with the game together.

Then the *putX* function is launched, which is responsible for waiting for the player's action and is also the link between the player's moves and the location of the crosses in the array. Next, the *test_win* function is called. It consists of several cycles. The first cycle is a pass through the array by indexes from the constant "table" and checking for the presence of 3 crosses in a row. If there is such a line in which there are 3 crosses in a row, then the function exits with the code corresponding to the victory - 0b10000000. If the function has not found such a combination, then it goes through the array again and looks for free cells, that is, cells in which NUL[1] lies. If there are any, it returns the continuation code of the game - 0b00000000. Otherwise, it's a draw - 0b11000000. When the execution of the *test_win* function is completed, we return to the main program. Further actions depend on the value we received from *test_win*. If the player wins, the winning row is highlighted by the *rowIllumination* function. If it is a draw, the output value is 0b11111111. It should be added that such a draw code is not tracked in the hardware, but since there is no cell with the code 0b1111 (if converted to decimal, it is 15, and as shown in the table above, there is no such index), this output is skipped and the game ends due to the lack of free cells. Otherwise, the transition to the second memory bank occurs, where the second part of our program is stored - artificial intelligence.

Artificial intelligence

```
inputs>
cells: dc 8, 10, 2, 0, 1, 4, 6, 9
table: dc 0,1,2
       dc 4,5,6
       dc 8,9,10
       dc 0,4,8
       dc 1,5,9
       dc 2,6,10
       dc 0,5,10
       dc 8,5,2
```

I code, with the value 0).

For this part of the program, we will need the following input: this is the previously used *table* and the *cells* constant, which is responsible for the priority of the cells that the AI will select when it needs to put noughts.

The artificial intelligence program starts with the following lines:

```
asect 0xf8
br 0x00
```

They are necessary for switching between memory banks.

Then the *find2inRow* function is called. This function deals with the main work of AI. It looks for lines containing 2 identical characters and the third is NUL (that is, the cell is still free). It works as follows: the character code opposite to the one for which we want to find 2 in a row is loaded into the r3 register. This is done to optimize the search: if a cell is found in any row containing a character with the code from r3, this row is skipped. For example, if the function is looking for 2 crosses in a row to prevent a player's winning, and there is a nought in the row, then this row can be skipped. First, *find2inRow* looks for two noughts in a row, so that if they are found, put the third one and win. If it does not find such a combination, it exits the function. Then it is run again to find two crosses in a row and put the third nought in order not to let the player win. If *find2inRow* finds two identical characters in a row, then it puts a nought in an empty cell in this row in the array by calling the *put0* function. If, after executing the function *find2inRow*, the nought has not been added to the array, then the *testMiddle* function is called, which puts the nought in cell 5, if it is free.

If the middle of the field was occupied, then the *callCorners* function is called with a further call to the *corners* function. These functions use *cells*, analyzing what lies in the array in cells with indexes from this constant, and if there is a NUL, they put a nought in this cell.

After that, the *output* function is called, which puts data with information about the status of the game in the cell with the address 0xE3 (it may be a player's victory, his defeat, draw, or continuation of the game), the array index and the symbol code that is put in the cell with this index. And if this is an AI victory, then the *rowIllumination* function is called, which highlights the cells in the winning row. The last line of the program

```
br 0xf8
```

performs the function of switching from the AI memory bank to the main part memory bank, where the *putX* function is called and everything starts over.

Playing together

```
inputs>
table: dc 0,1,2
       dc 4,5,6
       dc 8,9,10
       dc 0,4,8
       dc 1,5,9
       dc 2,6,10
       dc 0,5,10
       dc 8,5,2
   end
```

In the input data, we need the *table* constant again, since almost all functions of this program use it.

To implement the game mode for two people, we wrote an additional program to recognize the victory of one of the players. Its essence lies in filling an array of characters in RAM and checking it for a victory by one of the parties.

The program starts with lines of code where the interaction between memory banks is implemented. Next, the putX0 function is called to add crosses and noughts to the field array. Initially, it is assumed that it counts information about the cross from the input data cell. After that, the test_win function is called, which, let me remind you, goes through all possible combinations and checks for the presence of 3 identical characters in a row. (a more detailed description of this function can be found in the main part).

Further actions depend on the value that the *test_win* function has written to r3:

- If *test_win* has written the victory code to the r3 register, then the *output* function is called first, which fixes the victory of the crosses with the score, and then *rowIllumination* is called, which also calls the *output* function, only now with the backlight code in 0 and 1 bits.
- If test_win has written the game continuation code to the r3 register, then all actions are repeated from the beginning, but only for noughts: call *putX0*, *test_win*, checking the victory of noughts. If the game is not finished after reading the nought, then the "br 0x00" command is used to return to the beginning of the program. Otherwise, the *output* function is called to fix the victory of the noughts and the *rowIllumination* function is called to highlight the row.
- If there is a draw after the move of the crosses, the *output* function is called, i.e. the processor transmits information about the draw to I/O data.