NSU FIT | Computing Platforms

Co-design Group Project

“The Game of Noughts and Crosses”

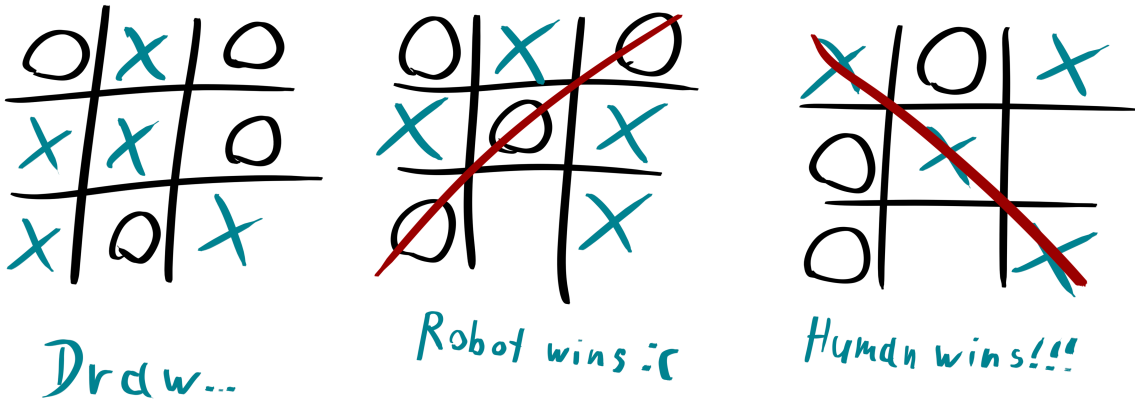
Lylova Sofia, Smolyakov Pavel, Badin Ivan

2022

I. Rules of the game of noughts and crosses

The game involves two players. They are given an empty field 3x3. One player puts crosses in the cells of the field, the other puts noughts. It is forbidden to put a cross or a nought in a non-empty cell. The first one who lines up 3 of his figures (crosses or noughts) vertically, horizontally or diagonally wins. If the field is filled, but none of the players won, then a draw is declared.

In our implementation of the game, the human goes first. His figures are crosses. Accordingly, the computer goes second, and its figures are noughts.



II. Hardware

The hardware employed for this project consists of a full-core split-memory CdM-8 system with an I/O bus, to which a 9 cell game pad is connected. The game pad uses two I/O addresses.

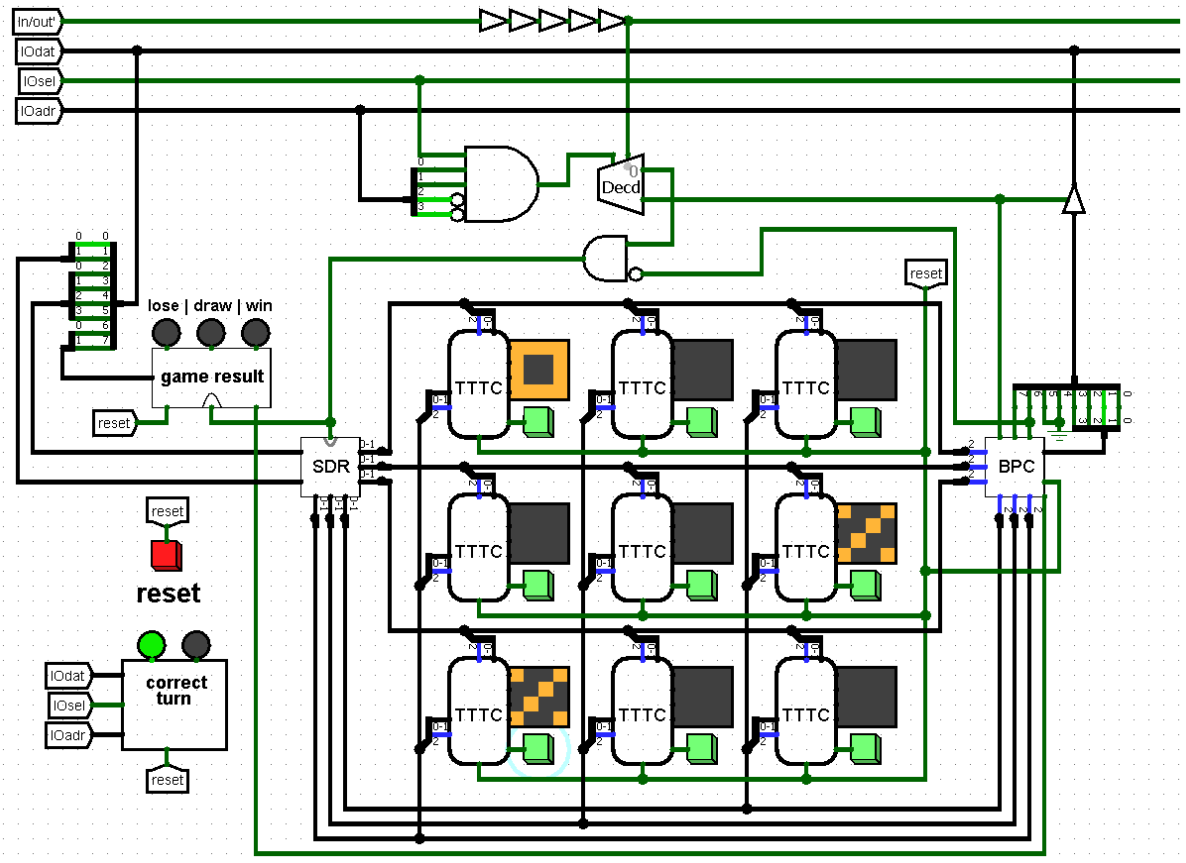
Game pad

Below is our implementation of the game pad. It consists of 9 4x4 LED matrix, 9 green buttons, and one red reset button. One matrix and a green button correspond to one cell of the field. Game pad uses three horizontal 3-bit buses (Hor_bus0, Hor_bus1, Hor_bus2) and three vertical 3-bit buses (Ver_bus0, Ver_bus1, Ver_bus2). Each cell connected to one horizontal bus and one vertical bus.

Let's number the columns of the field from left to right starting from 0, and the rows from top to bottom starting from 0. Each cell is defined by two coordinates: the x-coordinate is the cell column number and the y-coordinate is the cell row number. The address corresponding to each cell is a concatenation of the binary representation of x-coordinates and y-coordinates.

In the picture below, the nought has address 0b0000, the right cross has address 0b1001, the left cross has address 0b0010.

Bits 0 and 1 of each bus are used for output from the CPU to the attached cells and bit 2 is used for input from the attached cells to the CPU.



Individual cells

Each cell is an LED matrix for displaying the figure and a button for selecting this cell. These are controlled by a *TTT Cell* (labeled TTTC) chip which is connected to the vertical and horizontal buses corresponding to this cell. Each cell has its own TTTC chip.

- On the North side of the TTTC chip we have a 1-bit pin labelled Hor_out, and a 2-bit pin labelled Hor_in. Hor_out connected to wire 2 of horizontal bus. Hor_ in connected to wire 0 and 1 of horizontal bus.
- On the West side of the TTTC chip we have a 1-bit pin labelled *Ver_out*, and a 2-bit pin labelled *Ver_in*. These pins are connected to a vertical bus, similar to pins *Hor_in* and *Hor_out*.
- On the South side of the TTTC chip we have a 1-bit pin labelled *reset*. This pin is connected to the reset button, which is responsible for resetting the game.

- On the East side of the TTTC chip we have four 4-bit pins labelled *row0*, *row1*, *row2*, *row3* and a 1-bit pin labelled *btn*. Pins *row0*, *row1*, *row2*, *row3* connected to the LED matrix. Pin *btn* connected to button, which is responsible for the player's choice of this cell.

The specification for a TTTC chip appears in the Appendix.

Game pad management

The game pad is controlled by two chips. These two chips are connected to all vertical and horizontal buses. One chip is responsible for saving button presses. The other chip is responsible for transferring the symbol to the desired cell of the field.

Button input

To save button presses, we have the *Button Press Capture* (labelled BPC) chip. Its job is to save ID of the cell in which the button was last pressed, and to save whether the restart button was pressed. This information may then be read by the processor.

BPC chip is connected to bit 2 of horizontal and vertical buses. Pins on the chip for each bus are labeled by the name of the bus. There are also *button_reset* and *end_of_game* pins on the chip. Pin *button_reset* connected to reset button. Pin *end_of_game* connected to pin *end_of_game* of chip Game State Display Driver. BPC chip has 3 output pins: 1-bit *ready*, 1-bit *reset*, 4-bit *XY*. This data can later be read by the processor.

The specification for a BPC chip appears in the Appendix.

Symbol output

In order for the symbol to be converted into the correct field change, we have the Symbol Display Router (labelled SDR) chip. The symbol ID and coordinate of the cell come from the processor.

This chip has three input pins: 1-bit *good* (on the North side), 4-bit *XY* and 2–bit *Symbol_ID*. This chip is connected to the 0 and 1 bits of each vertical and horizontal bus. The output pins of this chip are labelled according to the buses to which they are connected.

A *symbol_ID* is sent to each cell. Let x, y be the coordinates of the cell in which we want to write the symbol. Then we send a code on the vertical bus x and the horizontal bus y that the desired cell lies on this line. And we send a code to all other buses that there is no cage on these lines. Only in the right cell, the codes for finding the cell go along the vertical and horizontal bus. Thus, the TTTC chip can understand in which cell to apply the change.

The specification for a SDR chip appears in the Appendix.

Packing input and output data in byte

Processor and game pad exchange data over an 8-bit IOdata bunch. The presented tables describe how input and output data are packed.

Input (from processor to game pad)		**Output (from game pad to processor)**	
bits	meaning	bits	meaning
0-1	symbol ID	0-3	cell coordinates
2-5	cell coordinates	4-5	not used
6-7	game result	6	was the restart button pressed

		7	was a new button of some cell pressed after the last reading

Game results output

To get the results of the game , we have a chip Game State Display Driver (labelled game result). It has 2 input pins: 1-bit *reset* and 1-bit *good* (on the South side), 2-bit *result* (on the West side). Pin *result* is connected to 6-7 bits of IOdata bunch. Pin reset is connected to reset button. Also this chip has 4 output pins: 1-bit *loss*, 1-bit *draw*, 1-bit *win*, 1-bit *end_of_game.* Pins *loss*, *draw*, *win* are connected to diodes. The lighting of a certain diode means that the game is over with this result.

The specification for a GSDD chip appears in the Appendix.

Processor's bunches

We have 4 buses that are shared with game pad: 8-bit *IOdata*, 4-bit *IOadr*, 1-bit *IOsel*, 1-bit *In/out*. If it implies data exchange between the processor and the game pad, then *IOsel* rises. We use addresses 0xf3 and 0xf2 for data exchange. *IOadr* is 4 lowest bits of the address at which the processor is currently reading/writing. *IOadr* is used to determine which address (0xf3 or 0xf2) is used. *In/out* is signal indicating whether the processor is reading or writing.

Correctness of the turn

To output whether a turn made by human is correct, we use Correctness Turn Driver (labelled correct turn) chip. It has 4 input pins: 8-bit *IOdata*, 1-bit *IOsel*, 4-bit *IOadr*, 1-bit reset. Pins *IOdata, IOsel, IOadr* are connected to the corresponding bunches. Pin *reset* is connected to the reset button. The chip also has 2 output pins: 1-bit *good* and 1-bit *bad*. Pin *good* is connected to a green diode. Pin *bad* is connected to a red diode. The green diode lights up means that the move was correct. Accordingly, the red diode lights up means that the move was not correct.

The specification for a CTD chip appears in the Appendix.

III. Software

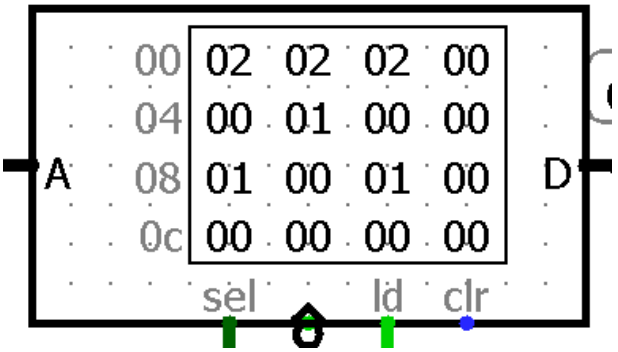
Generally, software part of tic-tac-toe implementation consists of input handler, two output handlers, game state checker, AI player and a main loop, which connects everything together. Description is given below.

Memory planning

Game table contents are stored in the first 11 bytes of RAM. As we differentiate cells by their X and Y coordinates, both of them represented by 2-bit string, we can access any cell in a constant time if cells are located at address 0bXXYY. Table is 3x3, so both coordinates are between 0b00 and 0b10.

The sample project implementation uses only one IO address. It's enough to realize the game's main functionality, but not extensions, so we were forced to use one more.

- Table: 0x00 - 0x0a
- Main IO address (namely *IOAdr*): 0xf3
- IO address for extensions (namely *IOExt*): 0xf2
- Stack: 0xf0 (as our stack grows backwards, elements will be stored in addresses 0xef, 0xee, 0xed and etc.)
- Counter which stores amount of busy cells (namely *celCnt*): 0xe0



Above there is a demonstration of how the game table is stored in the memory. *00* is an empty cell, *01* is a cross and *02* is a nought. As you can see, there's a line of three *02*'s, so this game is won by computer.

Input

Input handler starts a loop which ends when one of nine gametable buttons is being pressed. It waits until bit 7 of *IOAdr* is changed, then it stores bits 0-3 of the input line (X and Y coordinates of a cell in which a human tries to put a cross) in a register *r3*.

Our extension adds functionality to reset the game field. The handler checks if bit 6 of *IOAdr* is set to 1. If it is, the subroutine which returns everything to zero state is called.

Output

The ID of a symbol we want to show on the table is in bits 0-1 of register *r1*, game state is in bits 6-7 of register *r2*, address of the cell is in bits 0-3 of register *r3*. Main output subroutine moves the address of the cell to bits 2-5, gathers together the content of three registers and stores the result at *IOAdr*.

As for our extension which checks the correctness of our turn (looks whether the player tries to put a cross either in a busy or in a free cell), we store 0b10 at *IOExt* if a human is wrong and 0b01 otherwise.

Game state checker

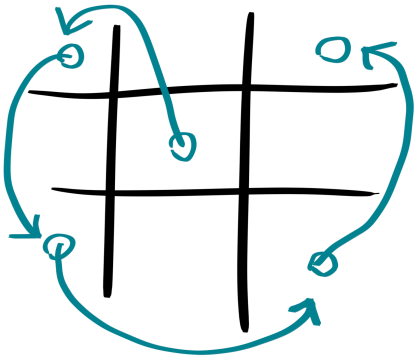
On the whole, our game state checker walks through the table trying to find a line of three. If this task fails, he gives a draw (0b11000000) or no result (0b00000000) depending on the amount of busy cells, otherwise it puts the winner's symbol ID to bits 6-7. Result is stored in register *r2*.

It's simple straightforward strategy, however, on purpose of improving working time, we store the amount of busy cells in *celCnt*, so we don't have to check the whole table every subroutine call. Specifically, when we have less than 5 busy cells, it means that we have both less than three noughts and less than three crosses, and it's needless to determine the winner. If 9 cells are busy and there is no line of three crosses or noughts, then we end the game with a draw result.

Algorithm's time complexity is O(n*k), where n is the number of cells in a line (in this case 3) and k is the number of possible lines to go through (in this case 3 vertical, 3 horizontal and 2 diagonal).

AI player

Unfortunately, our AI is not smart enough to defend against the human player, but he tries to. First of all it puts a nought in a center cell, then - in corners, maximally reducing the amount of strategies the human can use. Computer's route is given below.



Algorithm's time complexity is O(*n*^*2*), where *n* is the number of cells in a line (in this case 3).

Main loop

At all, the main loop structure follows the logical order of actions in a regular tic-tac-toe game.

1. Start input handler
    a. If reset signal is sent, clear everything up
2. Check if turn is correct and start extension output handler
    a. If turn is incorrect, move to the beginning of a loop
3. Start humanPlays subroutine
    a. Increase busy cells counter
    b. Store the cross at the cell where human wants to
    c. Check game state
    d. Start main output handler
4. If state is not draw, start robotPlays subroutine
    a. Increase busy cells counter
    b. Let the AI do his turn
    c. Store the nought at chosen cell
    d. Check game state
    e. Start main output handler
5. Repeat infinitely

IV. Appendix. Chip specifications.

```
=====================================================================
Chip Specification
    Name: TTT Cell (TTTC)
I/O Pins
    In: Ver_in(2), Hor_in(2), button(1), reset(1)
    Out: Ver_out(1), Hor_out(1), row0(4), row1(4), row2(4),
             row3(4)
Internal
    Signals: symbol_ID(2), address(2), bits(16)
    Latches: register(2) current_symbol
    ROMs: symbols(2->16) # A look-up table of 16-bit strings
                         # each with a 2-bit address
ROM contents
    symbols[0] = 0x0000 # bit-string for blank
    symbols[1] = 0x9429 # for nought
    symbols[2] = 0xF99F # for cross
    symbols[3] = 0x1234 # this table entry is not used,

Combinational
    symbol_ID.split(0) = Hor_in.split(0)
    symbol_ID.split(1) = Ver_in.split(0)
    # symbol_ID will be latched into current_symbol
    address = current_symbol
    bits = symbols[address]
    row0 = bits.split(0:3)
    row1 = bits.split(4:7)
    row2 = bits.split(8:11)
    row3 = bits.split(12:15)
    Hor_out = if button then 0 else float
    Ver_out = if button then 0 else float

Behaviour
    if reset then:
        current_symbol := 0
    elif Hor_in.split(1) && Ver_in.split(1):
        current_symbol := symbol_ID

End Chip Specification
=====================================================================
```

```
================================================================
Chip Specification
    Name: Button Press Capture (BPC)
I/O Pins
    In: Hor_bus0(1), Hor_bus1(1), Hor_bus2(1),
            Ver_bus0(1), Ver_bus1(1), Ver_bus2(1),
            reset(1), button_reset(1), end_of_game(1)
    Out: XY(4), clean(1), ready(1)
Internal
    Signals: x_crd(2), y_crd(2), xy_crd(4), cell_press(1),
             btn_pls(1), rst_pls(1), cl_pls(1)
    Latches: register(4) past_push, RS ready_fl, RS clean_fl
Combinational
    y_crd = if (not Hor_bus0) then 0b00
            else if (not Hor_bus1) then 0b01
            else if (not Hor_bus2) then 0b10
            else weak
    x_crd = if (not Ver_bus0) then 0b00
            else if (not Ver_bus1) then 0b01
            else if (not Ver_bus2) then 0b10
            else weak

    cell_press = not(V0 && V1 && V2) && not(H0 && H1 && H2) &&
                 not(end_of_game)

    xy_crd.split(0:1) = y_crd
    xy_crd.split(2:3) = x_crd

    XY = past_push
    ready = ready_fl
    clean = clean_fl
Behaviour
    on rising edge of cell_press:
        pulse btn_pls
    on falling edge of reset do:
        pulse rst_pls
    on rising edge of button_reset:
        pulse cl_pls
    if cell_press then:
        past_push := xy_crd
    on btn_pls do:
        set ready_fl
    on rst_pls do:
        reset redy_fl
        reset clean_fl
    if button_reset then:
        reset ready_fl
    on cl_pls do:
        set clean_fl
End Chip Specification
================================================================
```

```
================================================================
Chip Specification
     Name: Symbol Display Router (SDR)

I/O Pins
    In: symbol_ID(2), xy_crd(4), sel(1)
    Out: Hor_bus0(2), Hor_bus1(2), Hor_bus2(2),
             Ver_bus0(2), Ver_bus1(2), Ver_bus2(2)

Internal
    Signals: x_crd(2), y_crd(2)

Combinational
    x_crd = xy_crd.split(0:1)
    y_crd = xy_crd.split(2:3)

    Hor_bus0.split(0) = if x_crd is 0b00 then symbol_ID.split(0)
                        else float
    Hor_bus0.split(1) = if x_crd is 0b00 then sel else float
    Hor_bus1.split(0) = if x_crd is 0b01 then symbol_ID.split(0)
                        else float
    Hor_bus1.split(1) = if x_crd is 0b01 then sel else float
    Hor_bus2.split(0) = if x_crd is 0b10 then symbol_ID.split(0)
                        else float
    Hor_bus2.split(1) = if x_crd is 0b10 then sel else float

    Ver_bus0.split(0) = if y_crd is 0b00 then symbol_ID.split(1)
                        else float
    Ver_bus0.split(1) = if y_crd is 0b00 then sel else float
    Ver_bus1.split(0) = if y_crd is 0b01 then symbol_ID.split(1)
                        else float
    Ver_bus1.split(1) = if y_crd is 0b01 then sel else float
    Ver_bus2.split(0) = if y_crd is 0b10 then symbol_ID.split(1)
                        else float
    Ver_bus2.split(1) = if y_crd is 0b10 then sel else float

End Chip Specification
================================================================
```

```
================================================================
Chip Specification
    Name: Game State Display Driver (GSDD)

I/O Pins
    In: result(2), sel(1), reset(1)
    Out: win(1), loss(1), draw(1), end_of_game(1)

Internal
    Latches: register(2) cur_st

Combinational
    loss = if cur_st is 0b10 then 1 else 0
    win = if cur_st is 0b01 then 1 else 0
    draw = if cur_st is 0b11 then 1 else 0
    end_of_game = if cur_st is 0b00 then 0 else 1

Behaviour
    if sel do:
        cur_st := score
    if reset do:
        cur_st := 0b00

End Chip Specification
================================================================
```

```
================================================================
Chip Specification
    Name: Correctness Turn Driver (CTD)

I/O Pins
    In: IOdata(8), IOadr(4), IOsel(1), reset(1)
    Out: good(1), bad(1)

Internal
    Signals: sel(1), data_turn(2)
    Latches: register(2) turn

Combinational
    good = if turn is 0b01 then 1 else 0
    bad = if turn is 0b10 then 1 else 0
    sel = if (IOsel is 0b1 and IOadr is 0b0010) then 1 else 0
    data_turn = IOdata.split(0:1)

Behaviour
    if sel do:
        turn := data_turn
    if reset do:
        turn := 0b00

End Chip Specification
================================================================
```