

The game of Noughts and Crosses

22.05.2021

**Digital Platforms course project**  
  
by Samoseyko Petr   
and Kostin Andrey

# Overlook

The project presented is an emulation of a gamepad for playing the game of Noughts and Crosses (or Tic-Tac-Toe) versus a simple AI, powered by CdM-8 mark 4 processor unit. The circuit itself (including the processor) is emulated via Logisim, and the code for it is written in CdM-8 assembly language.

# Rules of the game and gameplay

The rules applied are the most common for playing tic-tac-toe:

1. The player makes a move, playing a Cross on the gamepad (pressing the appropriate button near the cell).
2. If the game is not over, the computer responds by playing a Nought in a free square.
3. Items 1-2 repeat until the game is over, namely:
4. AI or player managed to set up a full line of the symbols that they are playing with, or
5. there are no free spaces to play on the gamepad.

Here’s how to play the game using the gamepad provided.

1. Run the modeling in Logisim
2. To play your first move, press a button near the chosen cell
3. Wait for the circuit to load your cross and nought played by computer into the field (approx. 40 seconds per 1 player move on tact frequency 4.1 kHz)
4. Keep playing your moves as in 2. - 3. until the game is resultative

# Package contents

The project consists of the following:

1. main.circ - main Logisim circuit with Assembly code loaded
2. Assembly code in text and image formats (.asm and .img files respectively)
3. CdM-8 mark 4 circuit, which is used as a library in main
4. This documentation
5. Another variant of SDR chip (another\_sdr.circ)
6. Readme file with general information

# Working principles description

## **Hardware**

The Logisim circuit consists of the following:

1. **CdM-8 mk. 4 processor unit with Harward architecture**
2. **I/O bus, connecting the gamepad and the processor**
3. **The gamepad itself**

I will not talk about the processor itself or the implementation of Harward architecture that was used, as it is not the point of this documentation. Our main concern is the gamepad and its contents.

I am not going to provide pictures for all the subcircuits as well, as all of them are easily accessible in a .circ file

### 0. Overall information

The connection between processor and gamepad is established via an 8-bit IO bus, which sends data from the IO address to the gamepad and vice versa. Gamepad receives data in format SSXXYYII (state of the game; x-coordinate, y-coordinate, symbol ID of next symbol to be displayed), and sends data to processor in format R000XXYY, where R is a readiness bit (flag) and final 4 bits are coordinates of the last button press.

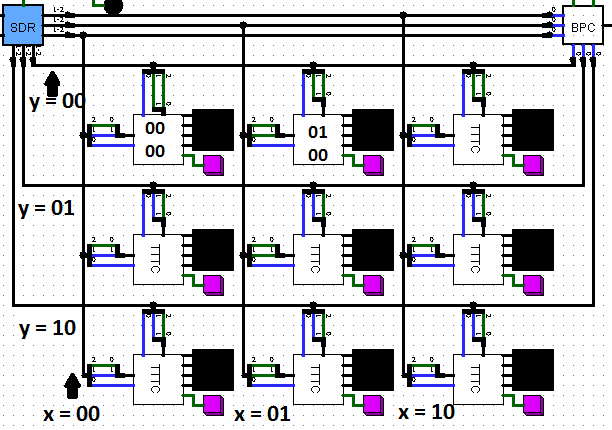
The gamepad uses 4 subcircuits, which are described below.

### 1. TTTC (Tic-Tac-Toe chip)

This part of the circuit is relatively simple. It actually just combines two different functions in one chip thus making the scheme easier to traverse: it is responsible for latching the symbol to be displayed by a 4x4 LED matrix (getting commands from SDR) and providing appropriate outputs from button to BPC.

There are 9 such chips, one for each cell that a symbol can be put in. It has 4 outputs for each row in the matrix and 2 outputs to the 3-bit vertical and horizontal buses to detect a button press. It also has 3 inputs: 1 from the button and 2 from the buses.

The bus usage is easily justified: each cell has a unique combination of the 2 buses, so asserting some signal to only 1 horizontal and 1 vertical bus will only make it possible to be read by a single predetermined chip (or *from* a single chip). These buses also give us the coordinates system: if we provide each bus with numbers 0..2 (or 00 to 10 in binary form), each cell can be given a unique 4-bit address in scope from 0000 to 1010, which is a concatenation of its coordinates (0111 and 0011 are not in use due to 11 not being in 00..10). This addresses are used in circuits and processor. First coordinate (horizontal) will be called X, and second (vertical) - Y in future for better understanding.



**Buses and coordinates example scheme**

But enough about the buses. Chip itself has signals *hin* and *vin* asserted from bits 0 and 1 of the buses. They are 2-bit signals, in which the following is coded:

**bit 0: a part of a symbol code (bit 0 of symbol is bit 0 of *vin*, and bit 1 of symbol is bit 0 of *hin*);**

**bit 1: indicator of whether the current symbol is to be read by the cell**

Whereas bits 0 are asserted on all the buses depending on the symbol ID entering SDR, bits 1 are only set to “1” on single vertical and single horizontal bus, so that actually only one chip changes its inner state. Both *hin* and *vin* drive a single 2-bit register, and the ROM unit decides which symbol to output to the matrix depending on the register state:

**00 - empty space,**

**01 - nought,**

**10 - cross,**

**11 - symbol “C” *(which should never occur during the game and means some error in the program).***

The second part of the scheme is very small and simple: if a button press occurs, *hout* and *vout* signals are set to 0; otherwise the signal on them is floating. *hout* and *vout* are asserted to bit 2 on the bus, which is untouched by SDR and is only read by BPC.

2. BPC (Button-Press Capture)

Our next chip is the one that provides output from gamepad to processor. It gets the signals from *hout*s of all TTTCs (bit 2 of all buses, 6 1-bit signals overall) and determines whether any button was pressed. If so, it latches the address of the chosen cell into a 4-bit register and raises a readiness flag. This flag is reset when the address is read by the processor and is used to ignore the input if no button was actually pressed. The fact of reading is determined by the *reset* input signal and its status from the previous processor tact, which is stored in a 1-bit register.

### 3. SDR (Symbol Display Router)

This subcircuit is used to drive outputs to bits 0 and 1 of each bus (6 signals overall) for them to be read by TTTC. It receives the following signals:

1. **2-bit *symID*,** as described in TTTC specification (bit 0 for each bus respectively), which is received in bits 6..7 from processor
2. **4-bit *XY*,** check the TTTC spec as well (it determines which 2 buses are used), which is received in bits 2..5 from processor
3. **1-bit *clock*,** which tells each TTTC whether it has to apply the input from SDR (it determines bit 1 for all buses at the same time)

Its overall behaviour is best described in the “Project A Notes” document by A. Shafarenko and S. Hunt. However our scheme is a little bit changed to be easier to understand. You may find modified specifications below. Such behaviour is easily reached by using 6 comparators and 6 transistors, 1 for each output signal.

#### xcoord = XY.split(0:1)

#### ycoord = XY.split(2:3)

#### H0.split(0) = symID.split(1)

#### H0.split(1) = if xcoord is 0b00 then clock else float

#### H1.split(0) = symID.split(1)

#### H1.split(1) = if xcoord is 0b01 then clock else float

#### H2.split(0) = symID.split(1)

#### H2.split(1) = if xcoord is 0b10 then clock else float

#### V0.split(0) = symID.split(0)

#### V0.split(1) = if ycoord is 0b00 then clock else float

#### V1.split(0) = symID.split(0)

#### V1.split(1) = if ycoord is 0b01 then clock else float

#### V2.split(0) = symID.split(0)

#### V2.split(1) = if ycoord is 0b10 then clock else float

### 

### 

### 

### 4. GSDD (Game State Display Driver)

This part of the scheme is driven by the first 2 bits coming from the processor. They tell the chip in which state the game is right now:

**00 - player win**

**01 - computer win**

**10 - game still in progress**

**11 - game drawn**

When a clock signal arrives, the internal register changes its value to the respective number received from the processor. The register signal goes through 3 comparators. They determine if any of the 3 LEDs should be lit for the player to see - green for win, red for loss and orange in a case of draw. If a game is in progress, no LED is lit.

## **Software**

### 0. Memory planning

To begin with, let us consider the memory usage. To get the board state at any moment, a 4x3 array is stored in the data memory *(0x00..0x0a),* which will be called just a field array in future: we do not need a proper label to access it, so it is unnamed in code. It is followed by counter *cnt,* which is used to determine whether there is a tie or not, and array *mem* [8] to count the sum of elements in each row, column and diagonal to identify a winner. For each element in mem, we store 3 elements in *table* [24] - a set of constants, defining which cell addresses belong to each row/column/diagonal. It is stored in code memory *(0xca and forwards)*, as it is more convenient to store constants there. *table* has order as below:

#### table: # each triplet below represents a line of three cells

#### dc 0,1,2 # horizontal lines

#### dc 4,5,6

#### dc 8,9,10

#### dc 0,4,8 # vertical lines

#### dc 1,5,9

#### dc 2,6,10

#### dc 0,5,10 # diagonal lines

#### dc 8,5,2

All the memory is planned after the code section.

The program itself consists of pre-execution settings, which adjust data memory for the AI loop to work correctly and the GSDD not to show any result, the put-in subroutine and the main loop. We will review each part in the order of execution.

### 1. Pre-execution

As GSDD receives 00 as its first value, it will display the green light (“human wins”) from the beginning and until the actual end of the game. For it not to do this, we are sending 0b10000000 (which means “game in progress, write blank space at x=00, y=00” and does nothing to the gamepad itself, as all the fields are empty before execution) to our IO address at data memory, namely 0xf3.

Then we should adjust the stack-pointer to be anything but 0xf\*, because these addresses are reserved for input-output. We decided to put it in the middle - address 0x80. As data memory is not used too much and we do not need a huge stack as well, this works just fine.

The last step before actual code begins: our AI is not particularly smart (more about it later). For it not to choose invalid numbers as addresses for its moves - 0x03 and 0x07 - we should put something besides 0 in them, for example, 1.

### 2. Main loop

After all the necessary settings are made, we are ready to go into the main loop of our program. Its logic works as follows:

1. Read the value from IO address to r1
2. If ready flag is not set (r1 - 0b10000000 < 0), go back to a (that means that no new button press occurred), otherwise go to c
3. If a button pressed corresponds to an occupied cell - ignore and go to a, otherwise put address where press occured in r2
4. set r1 = 2 (cross) and execute subroutine (now we only need to know that it takes symID from r1, address from r2, determines the result and sends a respective package to the gamepad, as well as to field array)
5. AI part: using while and 2 registers, cycle through addresses 0x00..0x0a in data memory and find the first empty cell. This is an address to put nought to, store it in r2
6. execute subroutine with r1 = 1
7. return to a

This loop is basically endless and is only broken by a subroutine when it detects any result.

### 3. Subroutine

This part is the most long-executed and complicated of all the software. It is supposed that before the execution there is a symID in r1 and an address to put it into in r2. The task for the subroutine is:

1. Put the symbol in field array
2. Change *mem* and *cnt* according to the state of array
3. See if after the symbol appearing the game has ended, write the result in r3
4. Sum address (moved to bits 2..5), result and symID, send the resulting package to gamepad
5. If the game is over, break the main loop, halt; else return to the place it was called from.

Let us look through each point of this list.

Putting the symbol in the field array is an easy task - we only need a single *st* instruction for that.

Changing *cnt* is quite simple as well - we increment *cnt* if r1 = 2. We know that a player is going to make 5 moves at most, so after a 5th cross being played and no result on board we declare a draw.

We will need all 4 registers not carrying useful information in the next part, so we push r1 and r2 to the stack

After that we should change *mem* to represent the state of the board after the symbol was played. We decided that the easiest way to do this will be to add 4 to *mem [i]* for each cross in a corresponding row and 1 for each nought - this way we get 3 or 12 when the game is won by someone, and result in scope 0..2&&4..11 otherwise. Doing that is pretty complicated. At the beginning of that part, we have r0 = 0 - how many elements have been checked already; r1 = 8 - the constant to which we compare r0 in the while loop; r3 = 0 - sum to add to the next mem. Then we execute following in a while loop with condition of (r0 < r1):

1. Load *table* address to r1
2. Add r0 to r1 3 times to get an address of row at the *table* to look at  
   (We get *table [3\*r0]* where r0 changes from 0 to 7, step = 1)
3. push r0, as we need one more register and r0 is not used in the next part
4. Load r1+3 to r0 (*table [r1]..table [r0]* is i-th row, its sum needs to be written in mem[i])
5. While loop (r0 > r1):
   1. Load next address to r2, load the symbol in this address to r2
   2. If r2 is not a blank (> 0), increment r3
   3. If r2 is cross (> 1), increment r3 three more times
   4. Increment r1, loop end
6. pop r0 (now it has how many of *mem [i]* were already changed)
7. store r3 in the next *mem [i]*
8. set for next iteration (r3 = 0, r1 = 8, r0++), loop end

When this loop finishes, we have to check whether the game is resultative yet. It is done in the following way (r3 is a result of the game):

1. If any element of *mem* is equal to 3, set r3 to 64, r0 to 0xf3 (IO address) and break
2. Else if any element of *mem* is 12, set r3 to 0 (player wins), r0 to 0xf3, break
3. If *cnt* = 5, set r3 to 192 (a drawn game), r0 - 0xf3, break.
4. If no draw or win occured (not broken until that point), set r3 to 128, r0 to 0xf3.

To finish the completion of subroutine, we have to pop r1 and r2 (address and symID), add r1 and r3 to r2 to then store this value to our IO address.

After that, if r3 (result of game) differs from 128 (which is “game continues”), we halt the program, breaking the main loop.

If no breaks occurred, we determine to which point in the program we have to return (this is still a subroutine!) by reading into symID in r1, as we run subroutine with each symID only once in the main loop.

**THIS CONCLUDES THE DOCUMENTATION**

**GitHub project link:** [**https://github.com/drewerr/Tic-Tac-Toe**](https://github.com/drewerr/Tic-Tac-Toe)