Documentation for Conway’s “Game of life” Project

# Introduction and Project Description

Our project implements the hardware and software components of John Conway's "Game of Life".

The system allows the user to configure the state of the field at any moment, start and stop the game simulation, adjust the game rules, and observe the process of cell generations changing.

The project uses CdM-16 based on the Harvard architecture, Logisim, and cdm-devkit.

The project includes the following key components:

* Keyboard for user command input
* Terminal for displaying commands and errors to the user
* CdM-16 Processor for processing user commands
* Video buffer for calculating the current state of the field
* Display for showing the game field

Our version of the game has several distinguishing features:

* Cyclic (toroidal) field
* Customizable game rules (birth and survival)
* Ability to change the field at any moment
* Implementation of user interaction through the keyboard and terminal using commands
* Output of errors and command execution states to the console.

# Hardware Description

## User Input

Input is performed through the "Keyboard" instrument, which allows the user to transmit commands and data to the system.

cl is the clock generator signal that allows the keyboard to send a character, kb\_ctrl is the signal enabling data sending from the decoder.

When a key is pressed, the keyboard generates a character code and transmits a signal indicating readiness to read characters.

The keyboard outputs are provided by: kb\_data, which transmits the entered character per clock cycle, and kb\_ready, which indicates the presence of characters in the keyboard.



## Keyboard Handler

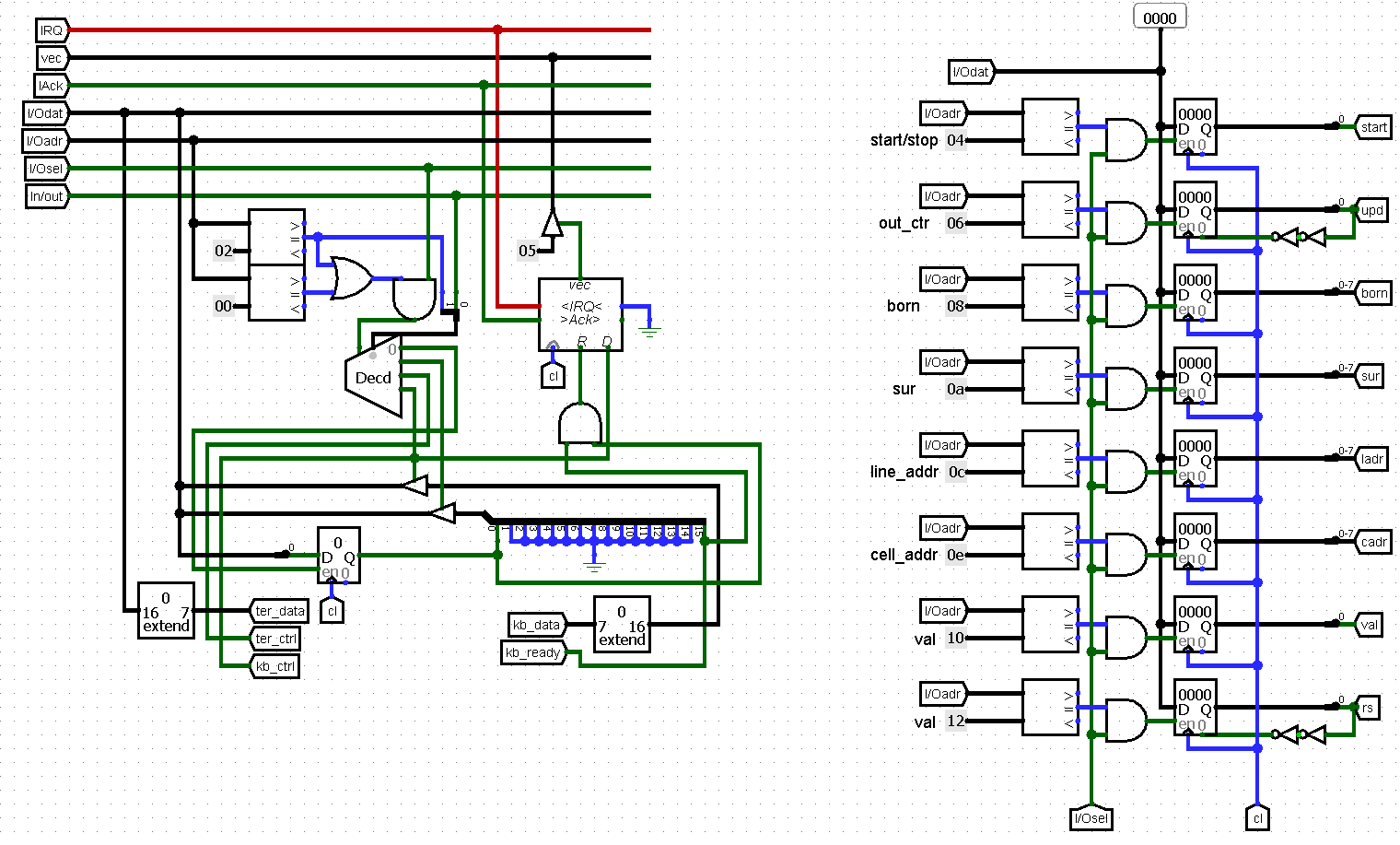
The keyboard handler receives signals from the keyboard, prepares data for sending to the I/Odat bus (and further sending to the ring buffer), and manages the generation of interrupt requests.

When the user enters data, the keyboard sets the character code on kb\_data and the readiness signal kb\_ready. Data from kb\_data is extended to 16 bits and sent through a controlled buffer, activated by the decoder, to the I/Odat bus for reading by the processor.

The kb\_ready signal is sent to the "comb" and the Interrupt Arbiter circuit, from which IRQ signals are generated, an interrupt vector is sent via the vec bus, and the \_kb\_isr interrupt handler is called. To enable interrupt reception, we load a unit into the ISTATE variable (at address 0xff00) via the processor, after which the interrupt enable register is set to 1, allowing the interrupt to be executed.

On the I/Odat bus, data is sent to ter\_data, truncated to 7 bits. The I/Odat bus also sends data to the video buffer inputs if they are in the required memory cells, which are compared with I/Oadr.

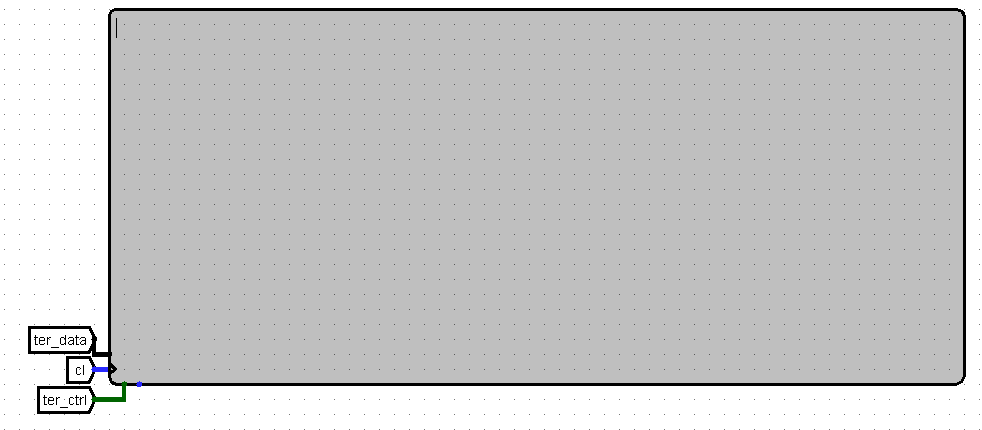
The decoder receives the enable signal from I/Oadr and I/Osel, which is passed through gates, and on the select input: the value in/out for the first bit, and the equality bit of the I/Oadr signal with 0x02 for the second. From the decoder, the values go to controlled buffers, a register, ter\_ctrl, and kb\_ctrl.



## Terminal Operation

The terminal receives values of entered characters via ter\_data from the keyboard handler, the clock generator signal, and the write enable bit ter\_ctrl.

The terminal is responsible for displaying user input, errors, and command execution status.

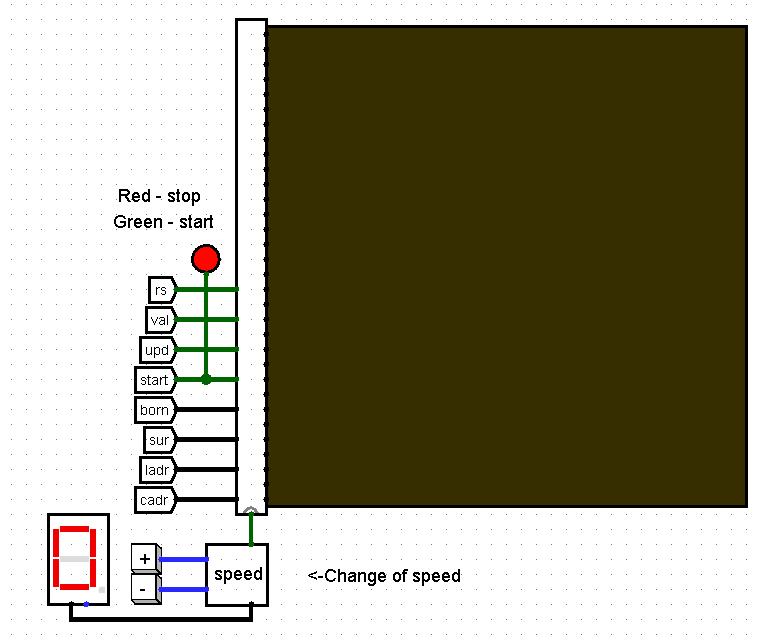


## Video Buffer

The video buffer receives values from the keyboard handler (rs - field clear, val - cell value, upd - forced field update, start/stop - simulation mode on/off, born - cell birth rules, surv - cell survival rules, line\_adr - editing line address, cell\_adr - editing cell address) and the clock generator value, whose frequency can be additionally regulated by the speed\_ctrl circuit.

The video buffer stores the state of the game field and processes simulation rules.

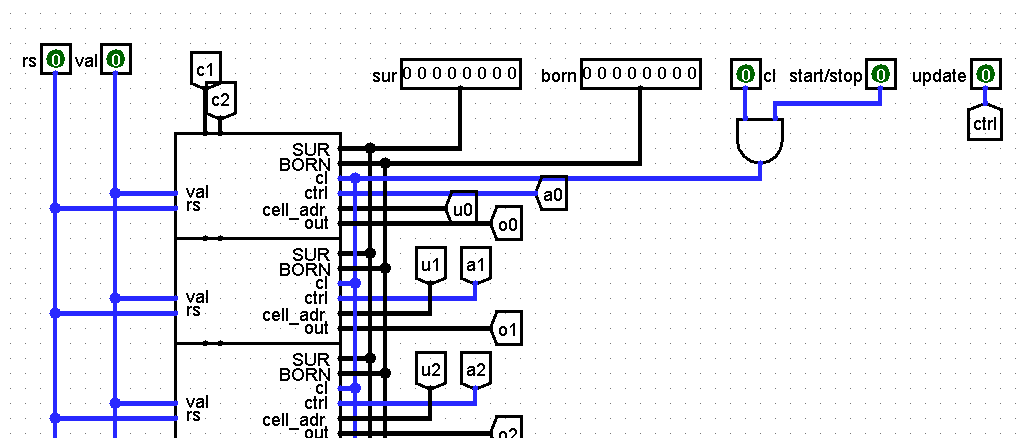
The video buffer is connected to the display and transmits the necessary values to each of its inputs.



# Detailed Description of Video Buffer Operation:

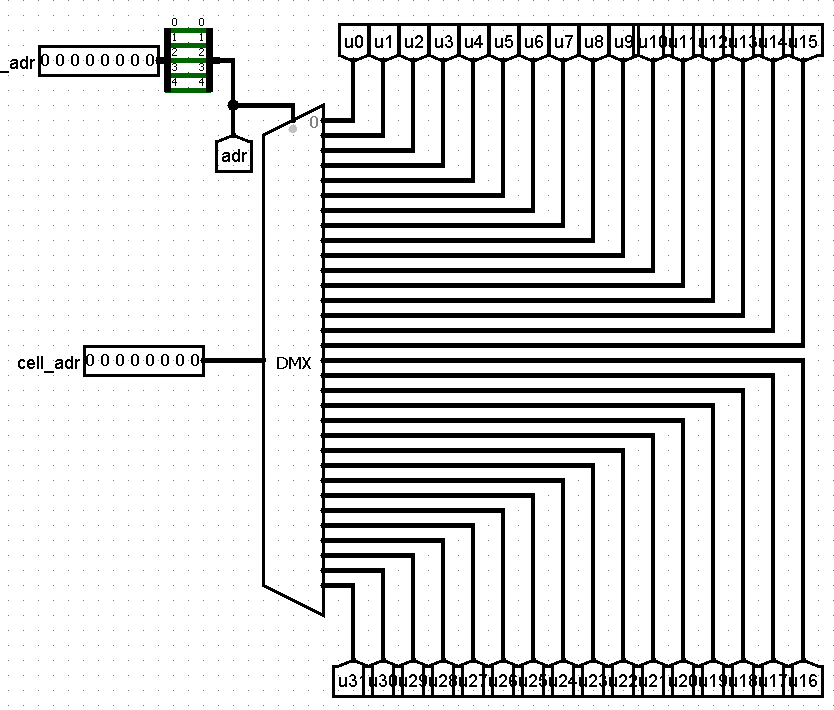
The video buffer receives rs, val, sur, born, cl, start/stop, update, which are transmitted to identical "line" objects, each corresponding to a field row. The objects are responsible for storing and processing the states of cells in one row of the game field. The objects also receive ctrl, cell\_adr values, which are calculated individually for each. All 32 objects are interconnected via up and down to account for cell values in neighboring rows.

КEach "line" object outputs out, corresponding to the values of all 32 cells in all rows.

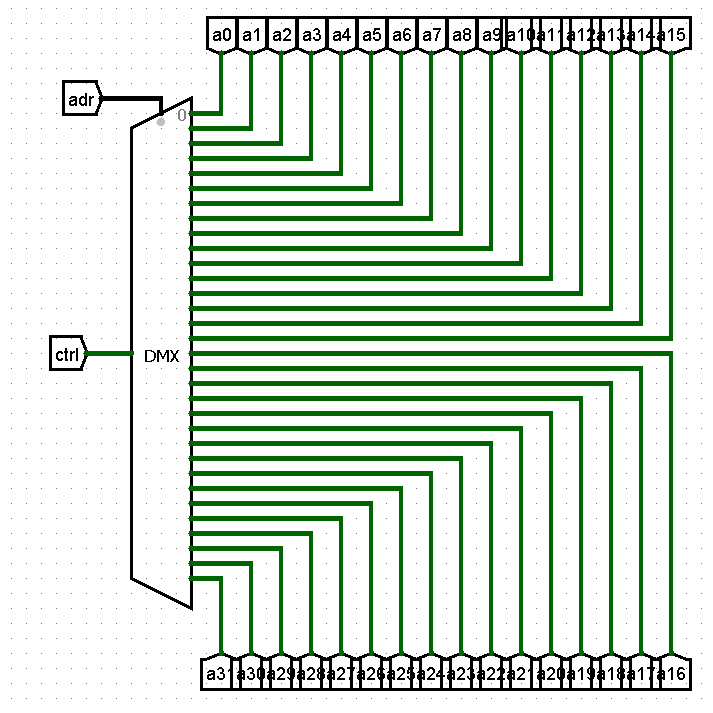


The calculation of ctrl, cell\_adr values is performed using two demultiplexers.

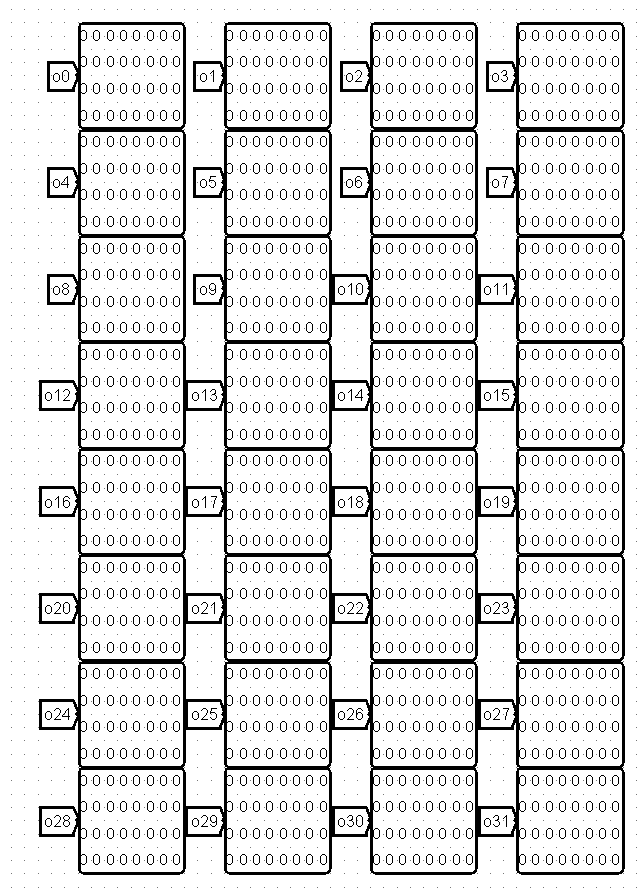
The first demultiplexer receives cell\_adr from the video buffer input and line\_adr and sends the address of a specific cell to the selected row's input.



The second demultiplexer receives line\_adr and update values and outputs the ctrl value for the necessary row for editing.



Each "line" object has an out output that transmits the values of all cells in the row and connects to the corresponding row on the display.

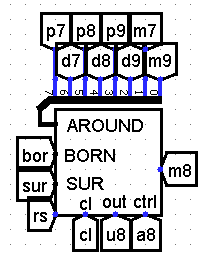


**Description of "line" Object Operation**

The "line" object is responsible for determining cell values in a row of the game field.

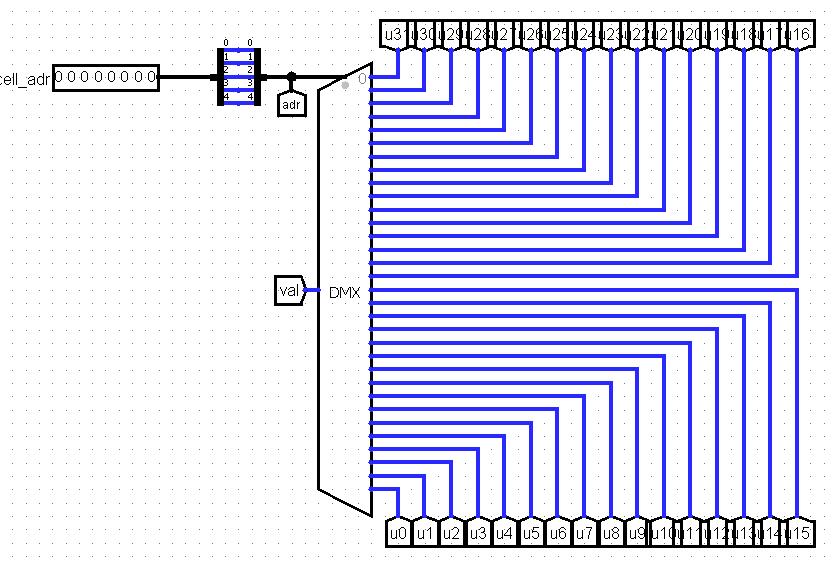
The value of each cell is processed through a "cell" object.

Here it receives the birth and survival rule values (born and surv), the field clear signal (rs), an 8-bit mask from the values of p (three upper cells), d (three lower cells), m (left and right cells), and the values of a (intentional value change) and u (value for cell update) of the given cell. Each cell has a 1-bit output corresponding to the cell's value.

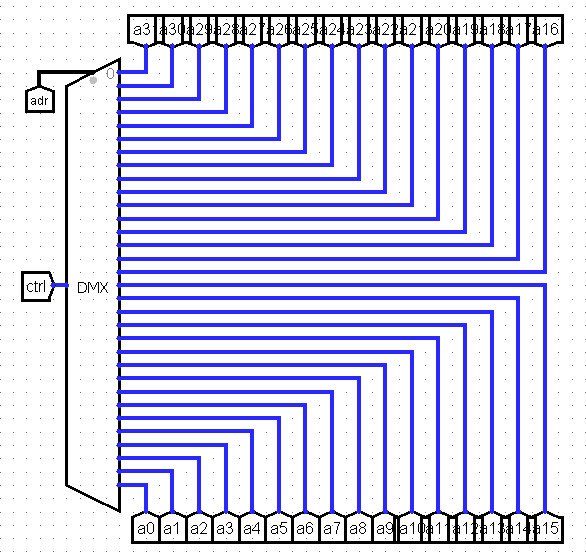


The calculation of a and u values is performed through two demultiplexers.

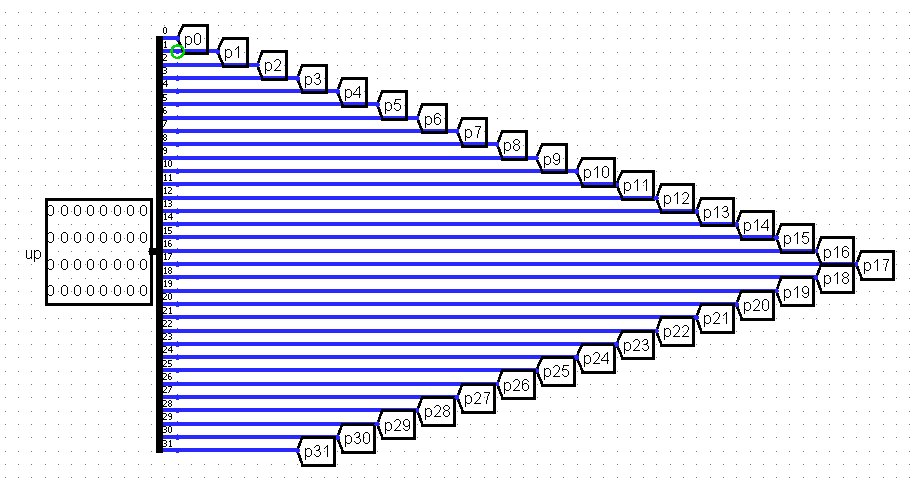
The first demultiplexer receives the cell value, cell\_adr and outputs u values for each cell.



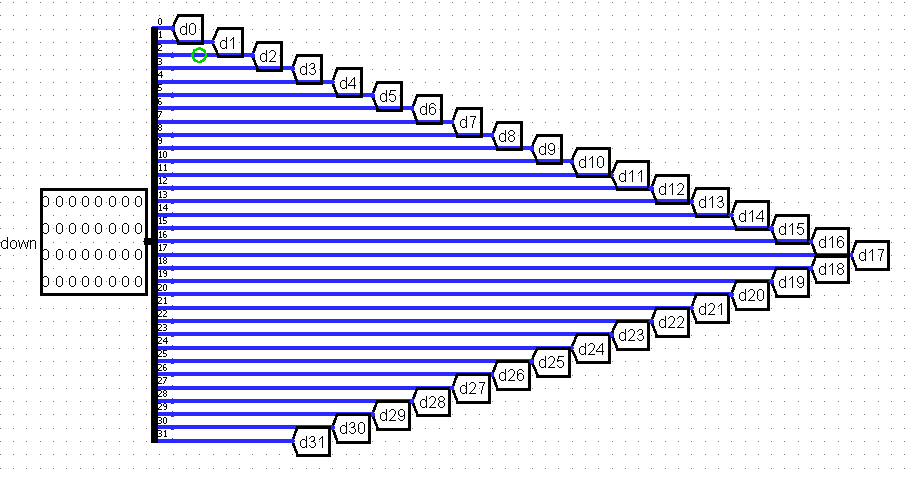
The second demultiplexer receives the cell value and ctrl value and then outputs the a value for each cell.



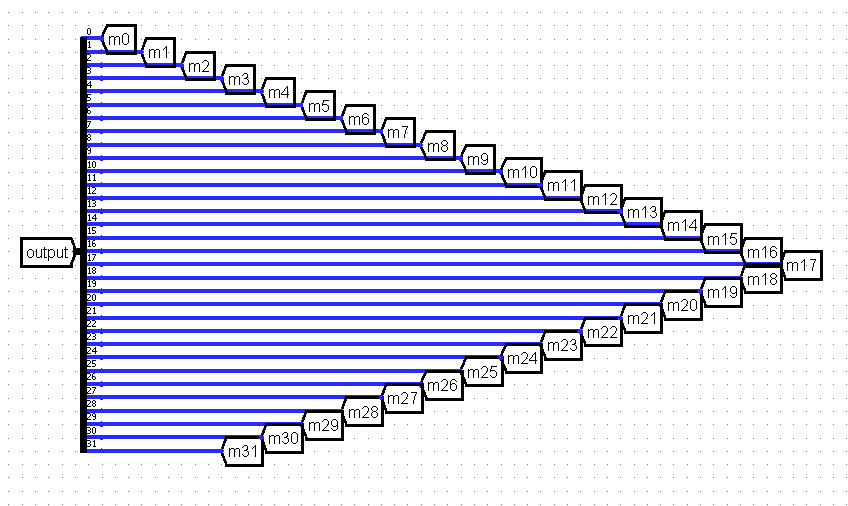
From the up input, data is sent bit by bit to cells with the prefix p.



rom the down input, data is sent bit by bit to cells with the prefix d.



The output is formed from the m values of all cells.

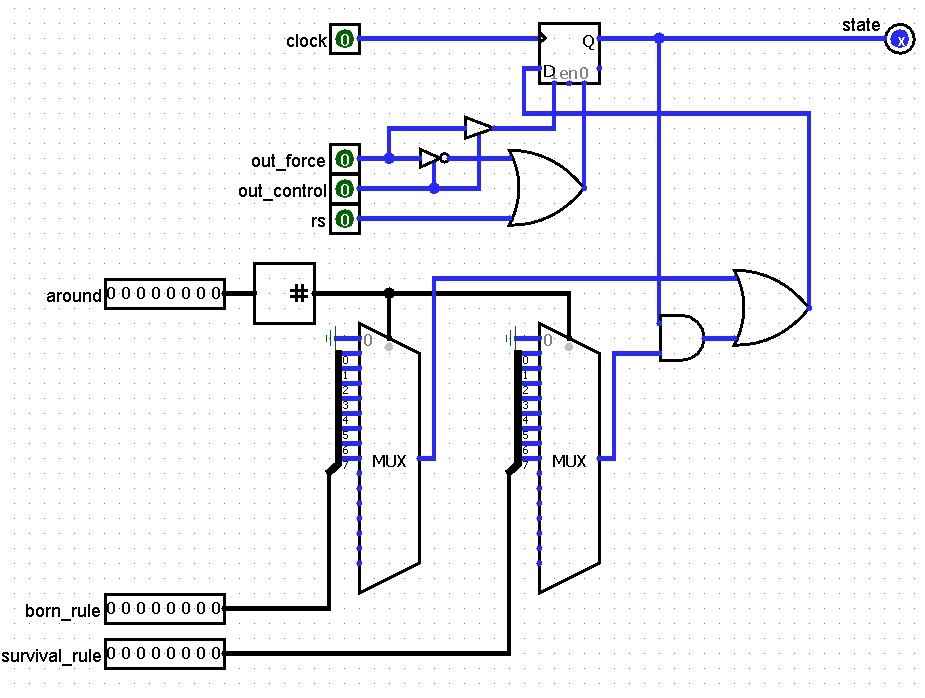


**Description of "cell" Object Operation**

The cell object receives born\_rule and survival\_rule, out\_force (what is sent through the u value), out\_control (what is sent through the a value), rs (field clear bit), and the neighbor mask around. around, passed through a bit adder (number of live neighboring cells), goes to the select inputs of the multiplexers, and born\_rule and survival\_rule go to the data bits.

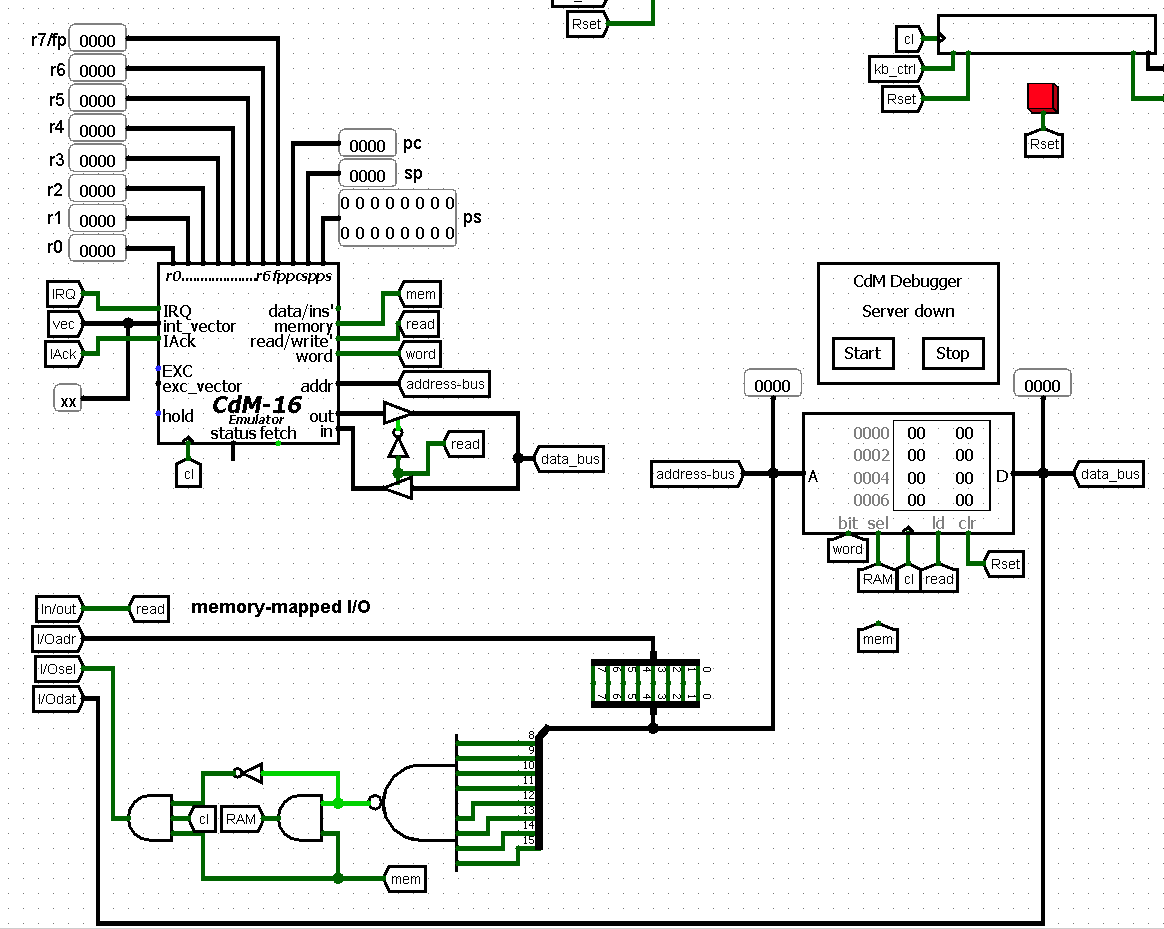
The born and survive outputs correspond to their inputs. These outputs are passed through gates, which then, along with out\_force, out\_control, and rs, go to a D-flip-flop, from which they are then output to state (cell state).

Thus, if out\_control is 1, the cell takes the out\_force value, otherwise the cell takes a value based on the state of neighboring cells and the specified game rules.



## Processor

The CdM-16 processor operates on the von Neumann architecture, meaning it uses a single address space for instructions and data. It reads and writes data via I/Oadr - I/O device address, I/Osel - device select signal, I/Odat - data bus. The processor controls data transfer between the keyboard, terminal, and video buffer via the decoder and buses.

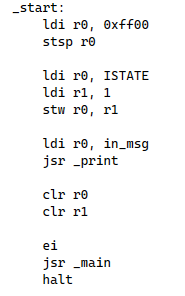


## Software Description

**Entry Point (\_start)**

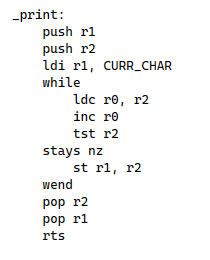
\_start launches the main program code at startup.

When the system is turned on, the stack is initialized, the ISTATE flag responsible for enabling keyboard interrupts is raised, the string "> " (command input prompt) is displayed using the \_print subroutine, and the main program loop \_main is started.



**Subroutine (\_print)**

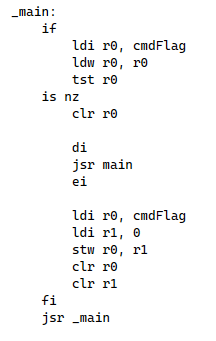
Responsible for sending characters from the keyboard to the terminal.



**Main Loop (\_main)**

\_main is necessary for coordinating command processing.

\_main receives the value of the cmdFlag flag, a non-zero value of which calls the main command parsing function from parser.c.

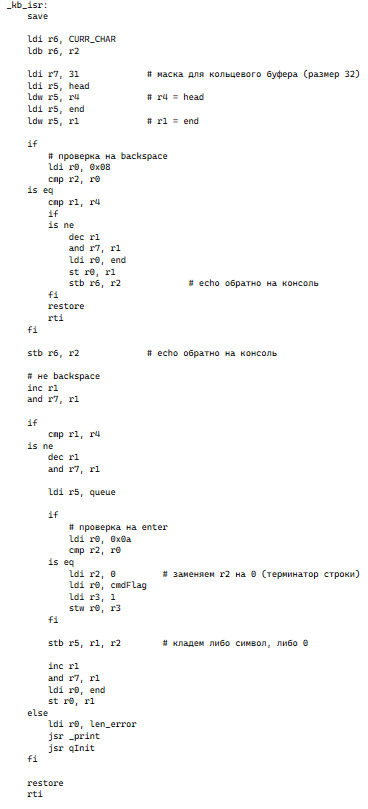


**Interrupt (\_kb\_isr)**

\_kb\_isr is activated when an IRQ signal is received from the Interrupt Arbiter located in the circuit. \_kb\_isr is responsible for reading a character, separately processing Backspace and Enter keys, and placing characters into the software ring buffer, which has a size of 32 bytes.

\_kb\_isr reads the character and starts the corresponding processing algorithm:

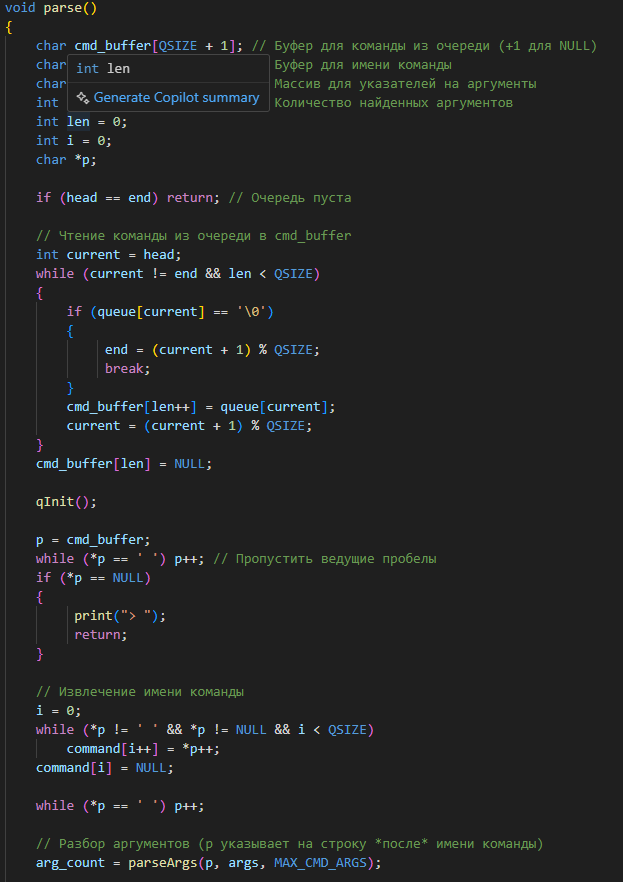
* If it is Backspace (0x08), the buffer is checked for characters. If there are characters in the buffer, the character is sent to the terminal to erase the last entered character on the screen. The pointer to the end of the buffer is shifted back (if the buffer is not empty).
* If it is Enter (0x0a), the character is output to the terminal, and a null terminator is written to the buffer, indicating the end of the string. The cmdFlag is set to 1, signaling \_main that the command is ready for processing.
* If it is another character, the character is output to the terminal. After this, the ring buffer is checked for available space. If it is full, a message about string overflow is displayed. If there is space, the character is placed in the buffer and the end of buffer pointer is incremented by one.



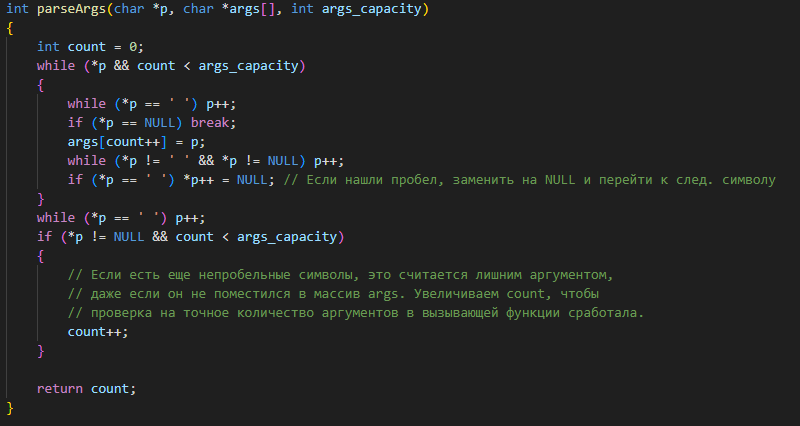
**Command Parsing (parser.c)**

The parser.c file contains the parse() function for processing commands, called from \_main.

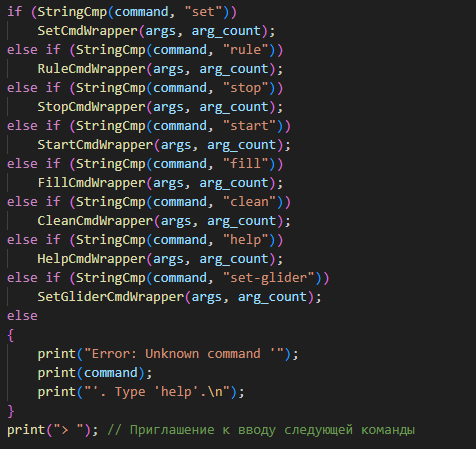
When the function is executed, the entered string is read from the ring buffer into a temporary buffer cmd\_buffer until a null terminator character is found, leading spaces are skipped, and then the first word is extracted from cmd\_buffer and moved to command.



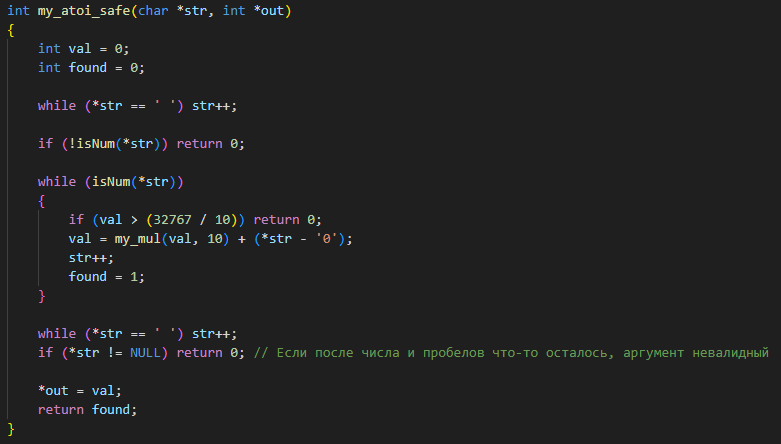
Then the number of arguments is counted using parseArgs(), the string is split into arguments by spaces, and pointers to the beginning of the arguments are saved.



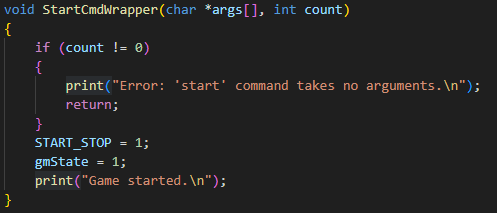
After this, a comparison is made with existing command names for their subsequent calling; if the command does not match the existing set of commands, an error message is returned.



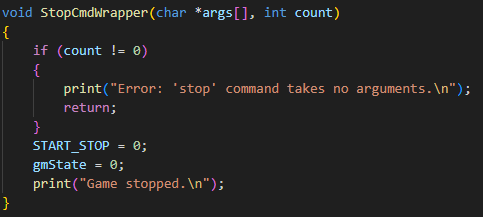
Each command calls its corresponding function, which checks the number of arguments for inconsistency with the required number of arguments of the called program (in which case an error is returned). my\_atoi\_safe() is used to extract arguments, which also checks the correctness of the entered arguments.



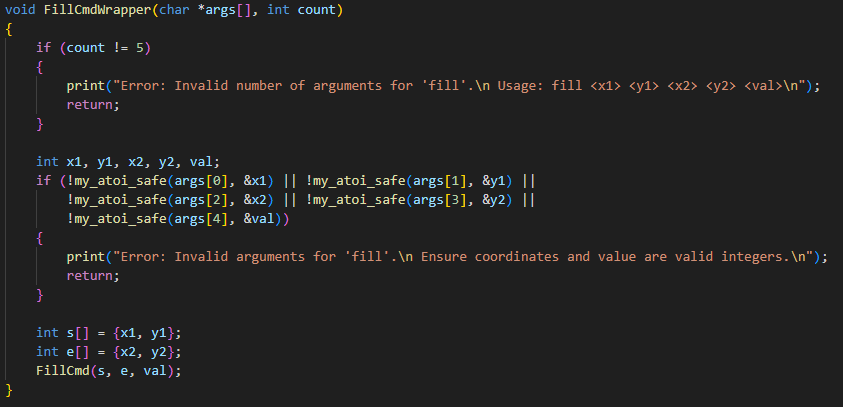
The StartCmdWrapper function sets the START\_STOP flag to 1 to start the game and the gmState variable for temporary game pause when using other functions and subsequent game resumption.

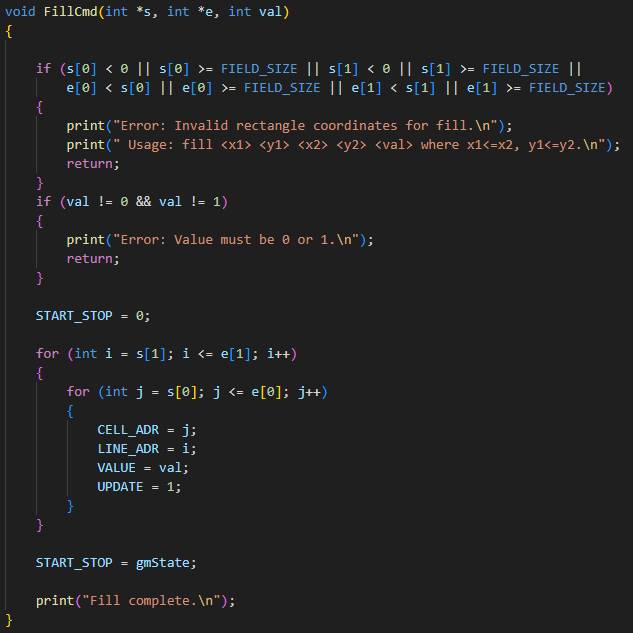


The StopCmdWrapper function does the opposite of the StartCmdWrapper function.

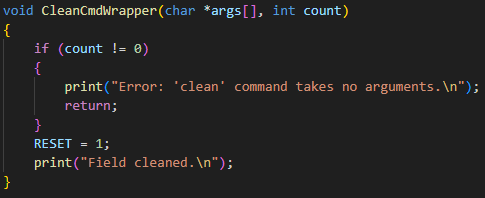


The FillCmdWrapper function calls FillCmd and passes it two arrays corresponding to the coordinates of the top-left and bottom-right corners of the rectangle, as well as the fill value. The FillCmd function transmits the same values to the cells in the user-provided rectangle, temporarily pausing the game.

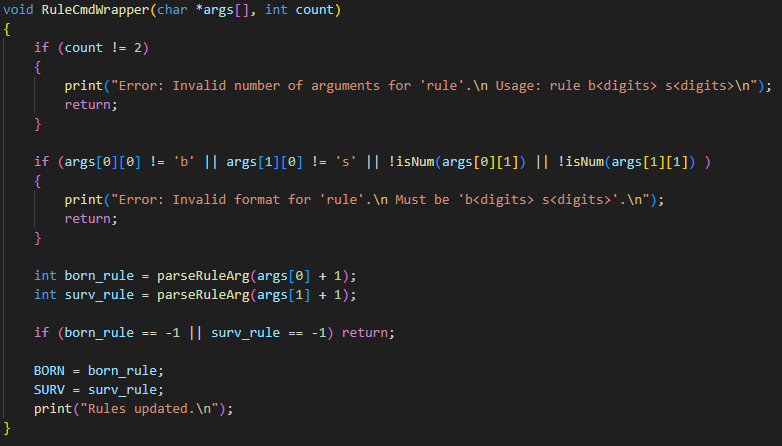




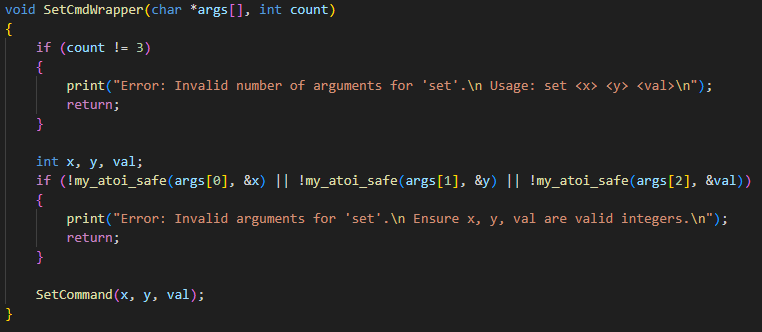
The CleanCmdWrapper function sets the RESET flag to 1 for a complete field clear.

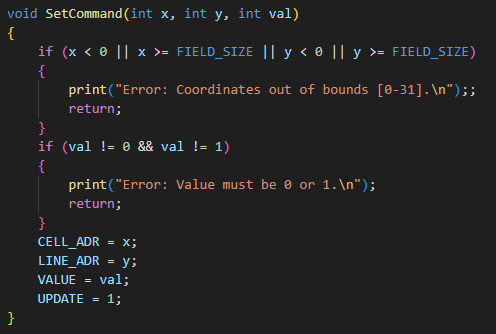


The RuleCmdWrapper function converts a string of digits into the corresponding bitmask for BORN/SURV, processing repeated and unordered digits.

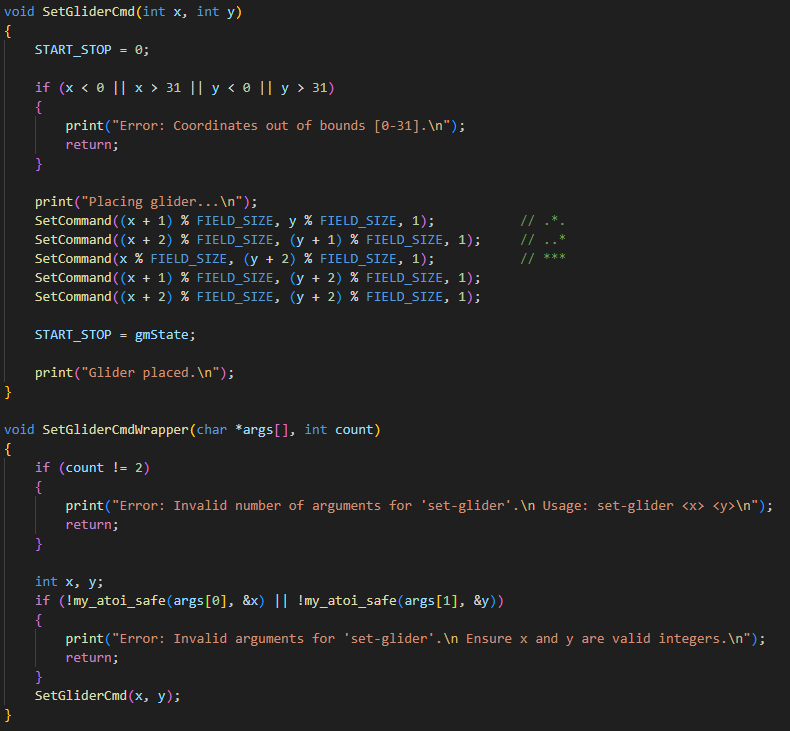


The SetCmdWrapper function calls SetCommand and passes it the coordinates and cell value. The SetCommand function sets the passed value to the cell at the given coordinate.

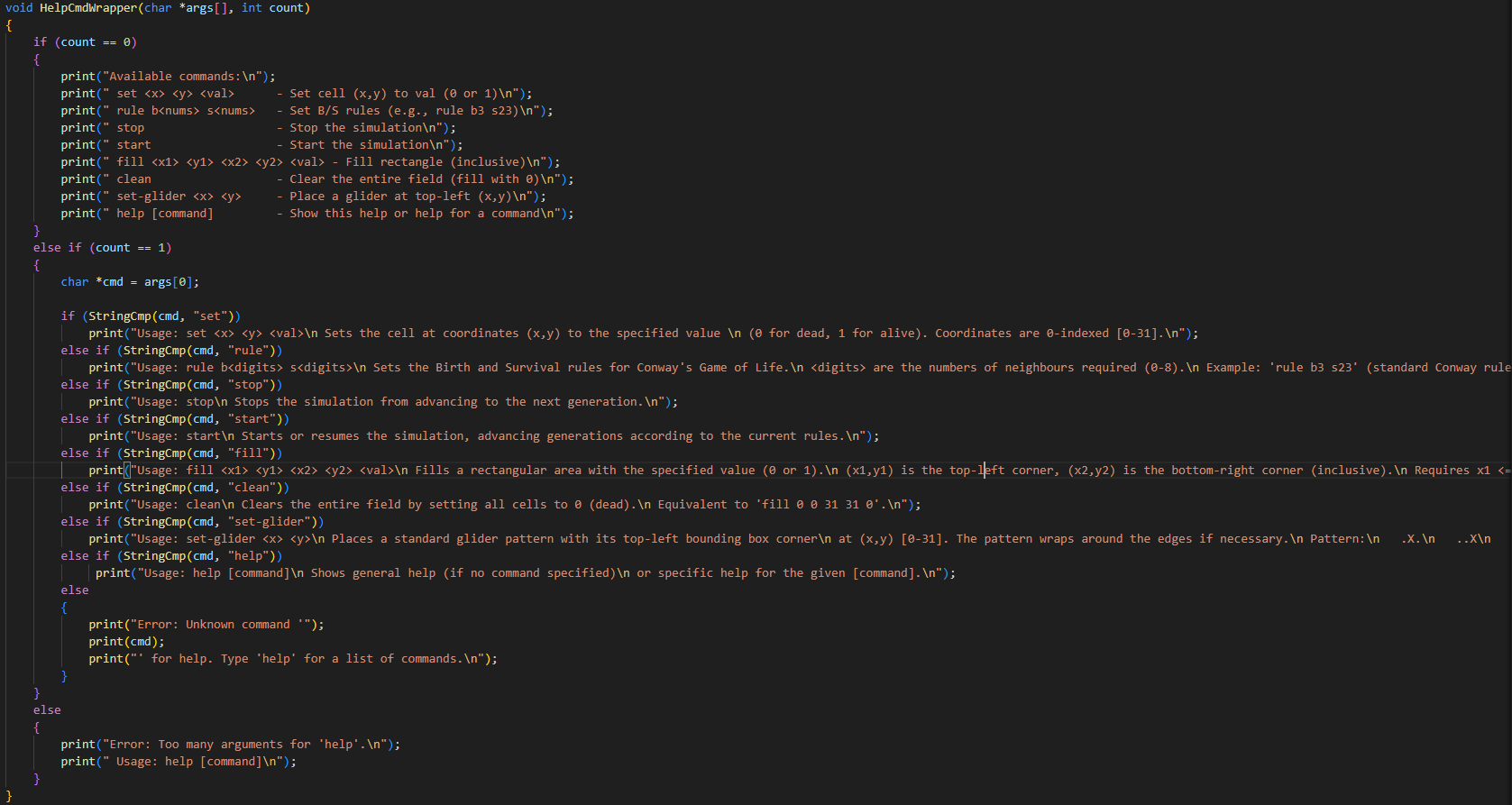




The SetGliderCmdWrapper function calls SetGliderCmd, passing the coordinates of the top-left corner of the glider. The SetGliderCmd function calls SetCommand filling the necessary cells to create a glider, temporarily pausing the game.



The HelpCmdWrapper function prints general help or help for a specific command depending on the arguments provided by the user.



**I**nteraction with the hardware is performed through registers mapped in memory. The video buffer receives CELL\_ADR, LINE\_ADR, VALUE, UPDATE, START\_STOP, BORN, SURV, RESET to control the field state and game rules, and CURR\_CHAR is responsible for receiving characters from the keyboard and sending characters for printing to the terminal.

# User Guide

1. Open the main.circ file in Logisim.
2. Select the maximum clock frequency.
3. Enable clock ticks.
4. Load the program.img image into memory.
5. You can enter commands via the keyboard. If you want to see the list of commands, enter help. For help on a specific command, enter help <command\_name>.

## List of Commands and Descriptions

1. set <x> <y> <val> — set the value of the cell at (x, y).
2. rule <born> <sur> — set the game rules: birth and survival.
3. stop — stop the game.
4. start — start the game.
5. fill <x1> <y1> <x2> <y2> <val> — fill the rectangle from (x1, y1) to (x2, y2) with <val>.
6. clean — clear the entire field.
7. help — display command descriptions; help [command] — display description for a specific command.
8. set-glider <x> <y> — draw a working glider at coordinates (x, y)

**Command Usage Examples**

1. set 2 3 1 — fill the cell with coordinates (2, 3).
2. fill 12 23 23 34 1 start — fill the rectangle with top-left corner (12, 23) and bottom-right corner (22, 33) (Note: Coordinates are inclusive, so x2=23, y2=34 likely means up to x=22, y=33 based on typical conventions, but translating literally), then start the game.
3. fill 0 0 2 6 1; fill 3 0 2 9 1; fill 3 7 9 9 1; fill 0 7 6 9 1; start — create "Kok's Galaxy" and start the game.



1. rule b3 s23 — standard "Game of Life" rules.

".