Novosibirsk State University

Project

“Conway’s game of life”

Made by:

Tokarev M.K.

Kuchenkov S.A.

Kozlovskaya K. V.

2025

# Introduction and Description

This project implements the hardware and software parts of John Conway's "Game of Life".

The system allows the user to configure the state of the field, start and stop the game, adjust the field's rules at any point during the simulation, and observe the process of cell generation changes.

The project uses:

* CdM-16 with a von Neumann architecture;
* Logisim and cdm-devkit.

The project includes key components such as:

* A keyboard for entering user commands;
* A terminal to display commands and errors to the user;
* A CdM-16 processor for processing user commands;
* A video buffer for calculating the next state of the field;
* An LED matrix for showing the game field (32x32).

This implementation of the game has several distinctive features:

* A looped (toroidal) field;
* Customizable game rules (birth and survival);
* The ability to change the field at any moment;
* User interaction implemented via keyboard and terminal using commands;
* Output of errors and command execution statuses to the terminal.

# Description of Operation

# Hardware Part

## User Input

Input is done through the “Keyboard” tool, which allows the user to send commands and data to the system.

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

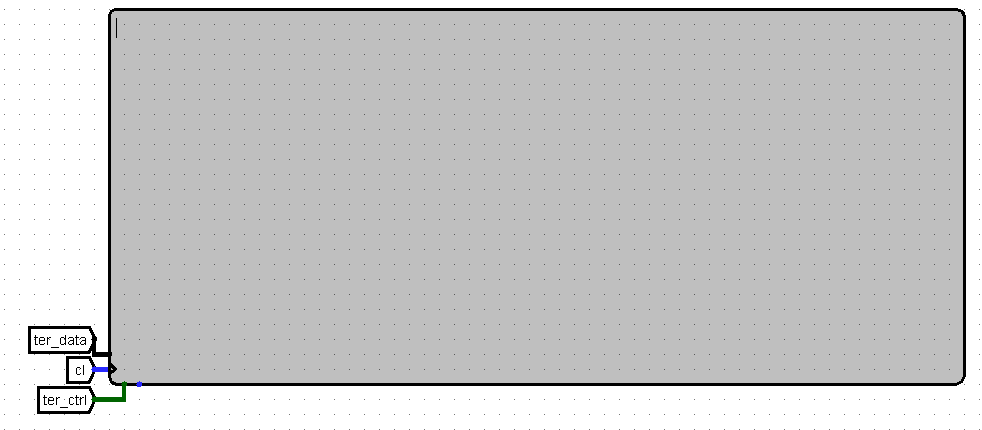
When a key is pressed, the keyboard generates a character code and signals that characters are ready to be read.

Output from the keyboard is provided by kb\_data which transmits the entered character per clock cycle and kb\_ready which indicates the presence of characters in the keyboard buffer.



The character is processed by the processor and sent to the terminal.

The terminal receives input characters via ter\_data from the keyboard handler, the clock signal, and the write-enable bit ter\_ctrl.



**Keyboard and Terminal Handler**

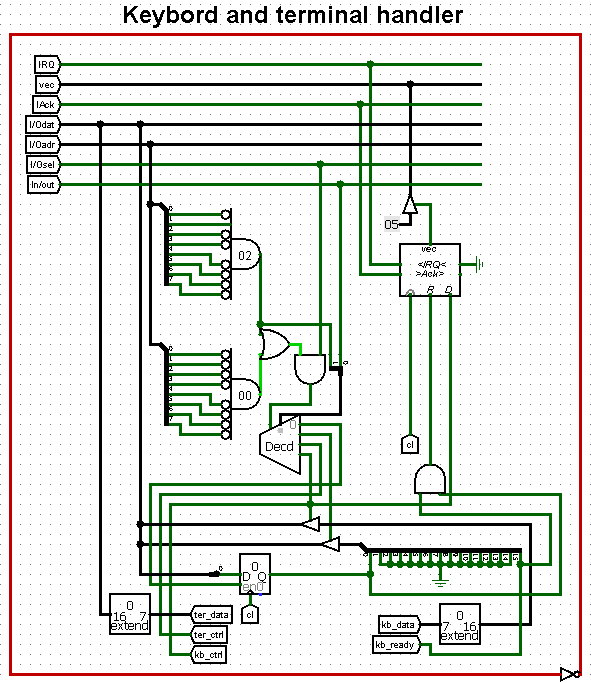
The keyboard handler receives signals from the keyboard, prepares data for the I/Odat bus (and further to the circular buffer), and manages interrupt requests.

When the user inputs data, the keyboard sets the character code on kb\_data and the ready signal kb\_ready. The data from kb\_data is extended to 16 bits and sent via a controlled buffer, activated by the decoder, to the I/Odat bus for the processor to read.

The kb\_ready signal is sent to the Interrupt Arbiter, which generates IRQ signals, sends an interrupt vector via the vec bus (vector number 5), and invokes the interrupt handler \_kb\_isr.

To enable interrupts, the program loads the value 1 into the variable ISTATE (at address 0xff00), after which the interrupt enable register is set to 1, allowing the interrupt to be executed.

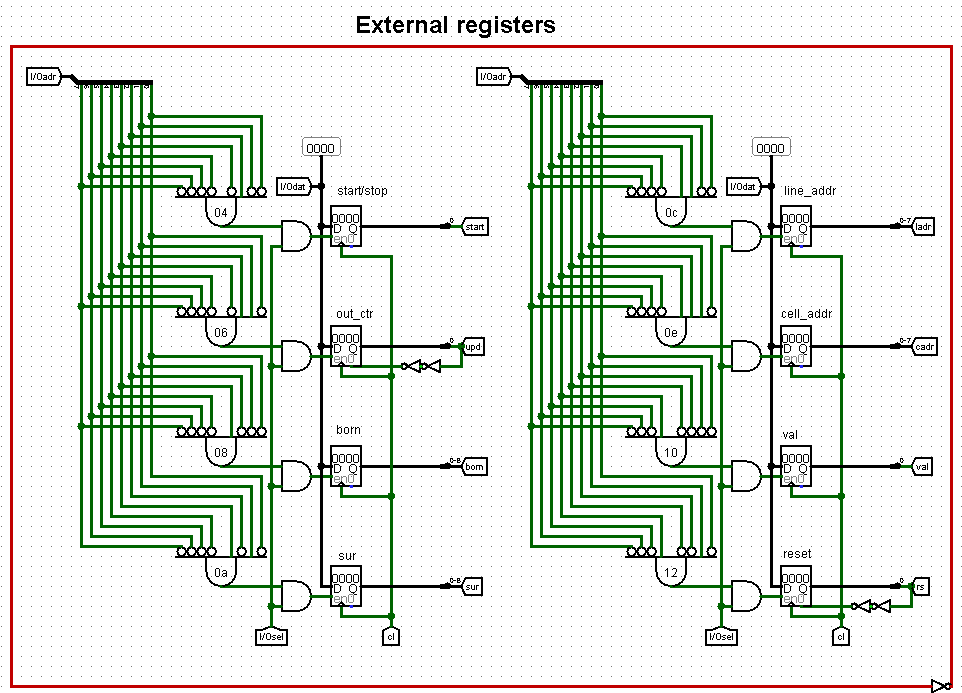
On the I/Odat bus, the data is sent to ter\_data, cut to 7 bits, to display the entered character on the terminal. The decoder receives at its enable input signals from I/Oadr and I/Osel, which passes through logic gates, and its select input: the first bit receives the in/out value, and the second bit receives a value indicating whether the I/Oadr equals 0x02. The decoder outputs are directed to controlled buffers, registers, ter\_ctrl, and kb\_ctrl.



**External Registers**

The I/Odat bus also sends data to external registers if they are located in the required memory addresses, which are compared with I/Oadr. All external registers are in the memory-mapped I/O region at the following addresses:

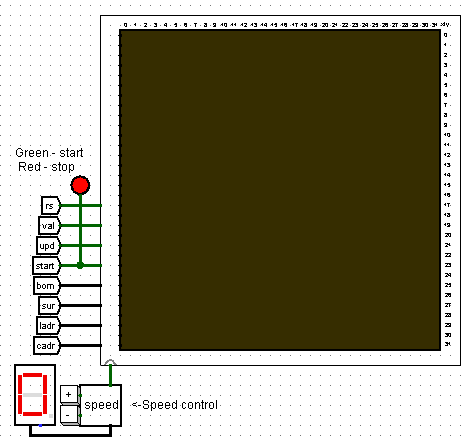
* 0x04 – Start/stop value (1 is start, 0 is stop);
* 0x06 – Asynchronous cell state update;
* 0x08 – Current born rule;
* 0x0a – Current survival rules;
* 0x0c – Line address for cell update;
* 0x0e - Cell address in line for cell update;
* 0x10 – Value for cell update;
* 0x12 – Reset bit.



## Video Buffer

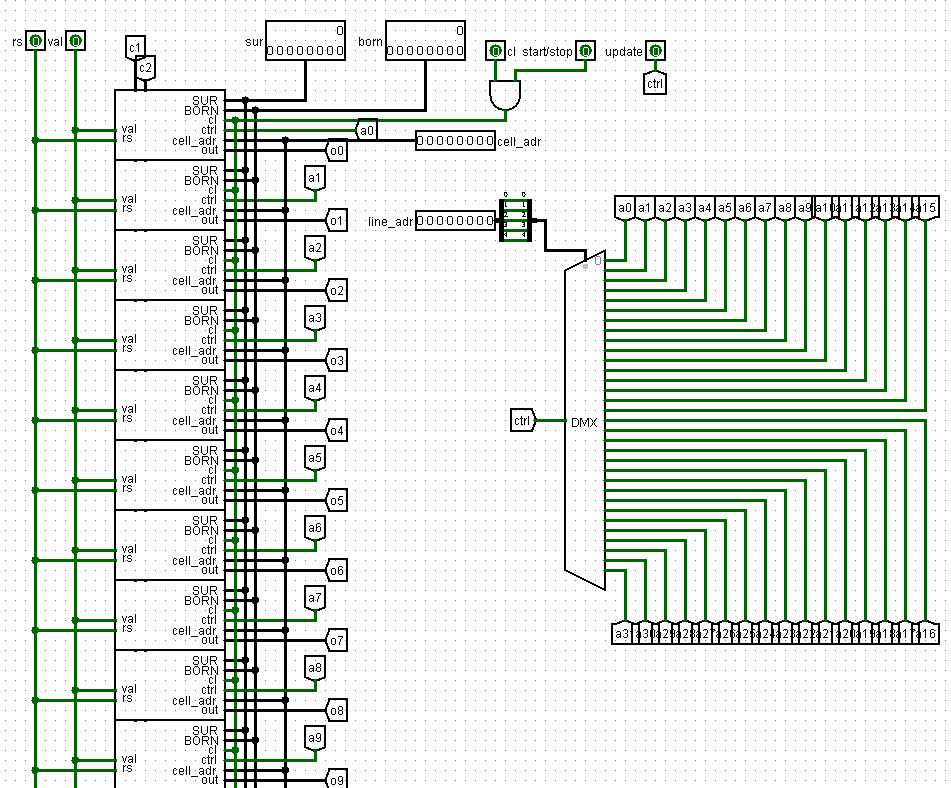
The video buffer receives all values from the external registers and the clock signal, the frequency of which can be adjusted by the speed\_ctrl circuit.

The video buffer stores the current state of the game field, computes the next state, and is connected to the LED matrix, transmitting the necessary values to each of its inputs.

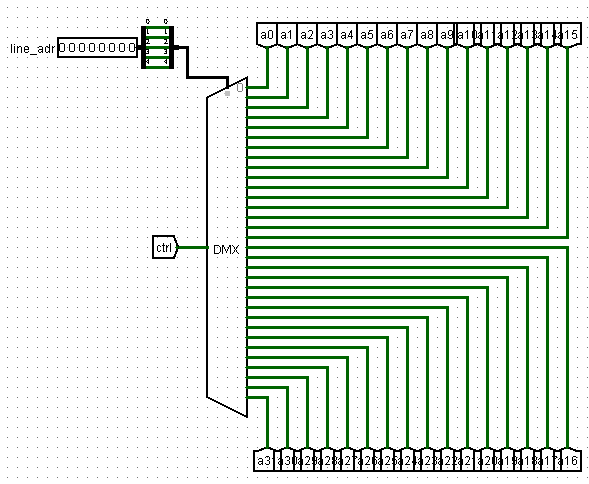


The input values for the video buffer are processed by identical "line" subcircuits, each corresponding to a row of the field. These subcircuits are responsible for storing and processing the states of cells in one row of the game field. All 32 subcircuits are interconnected via up and down signals to account for cell values in adjacent rows.

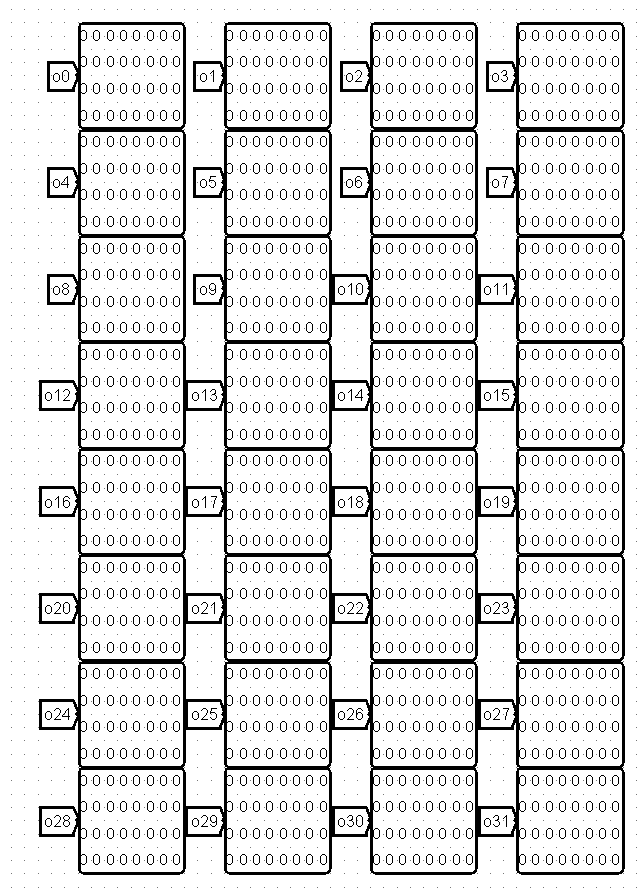
Each "line" subcircuit outputs out — a 32-bit array representing the values of all 32 cells in the row.



The calculation of ctrl and cell\_adr is performed by a demultiplexer. The select input receives line\_adr, and the output sends the update signal to the corresponding row for editing.



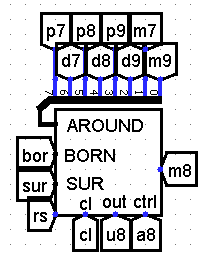
Each "line" subcircuit has an out output that transmits the values of all cells in the row and connects to the corresponding row on the LED matrix.



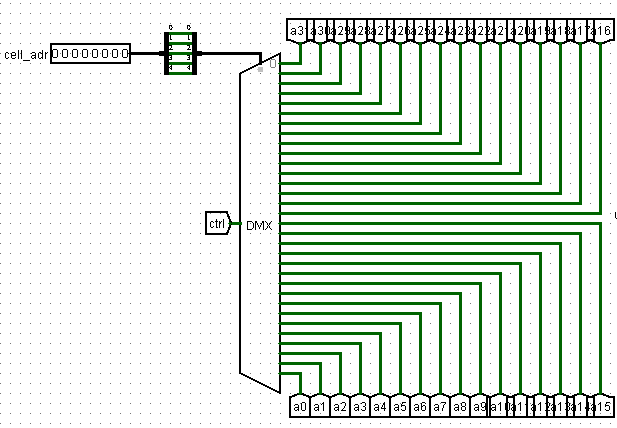
## Description of the "line" subcircuit

The "line" subcircuit determines the cell values in a row of the game field.

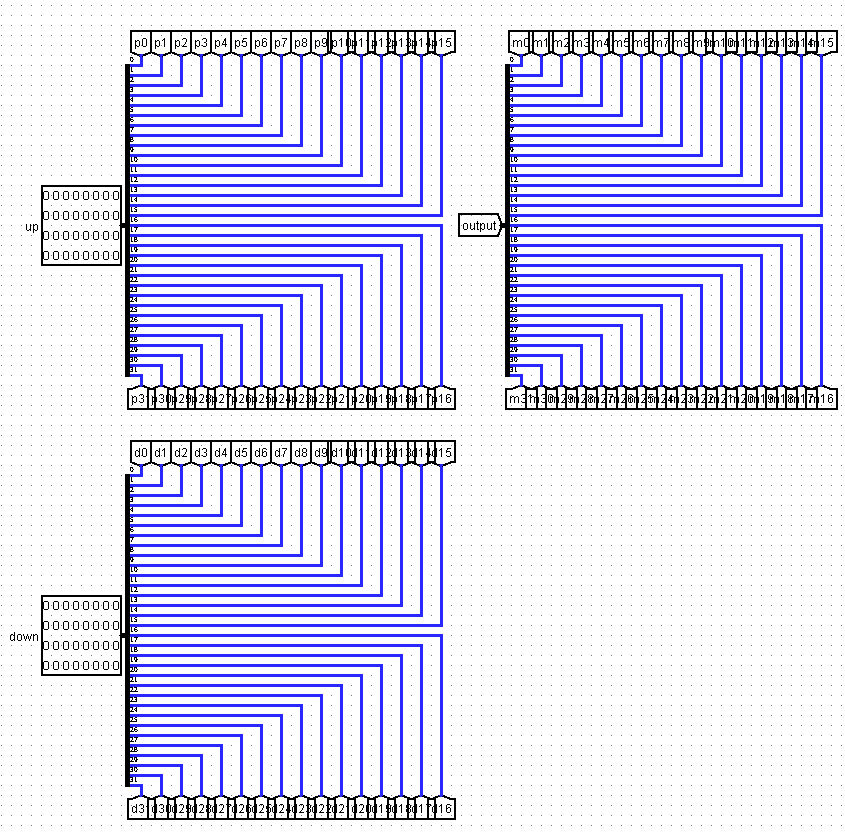
The value of each cell is processed by the "cell" subcircuit. Here, it receives the birth and survival rules (born and sur), the field reset signal (rs), an 8-bit mask of the three upper cells, three lower cells, and the left and right cells, the value for forced cell modification, and the value for updating the state of the cell. Each cell has a 1-bit output corresponding to its state.



The calculation of forced cell editing is performed by a demultiplexer. The select input receives the cell address, sending the update signal to the correct cell.



The "line" subcircuit also includes several splitters for correctly processing neighbors for each cell.

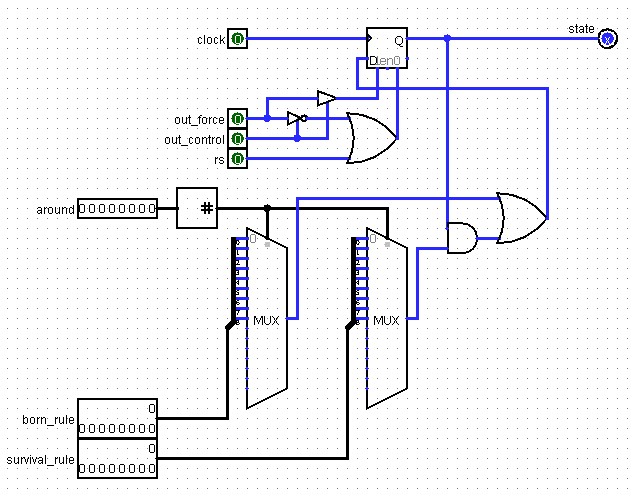


## Description of the "cell" subcircuit

The "cell" subcircuit receives born\_rule, survival\_rule, out\_force, out\_control, rs (the field reset bit), and the neighbor mask around.

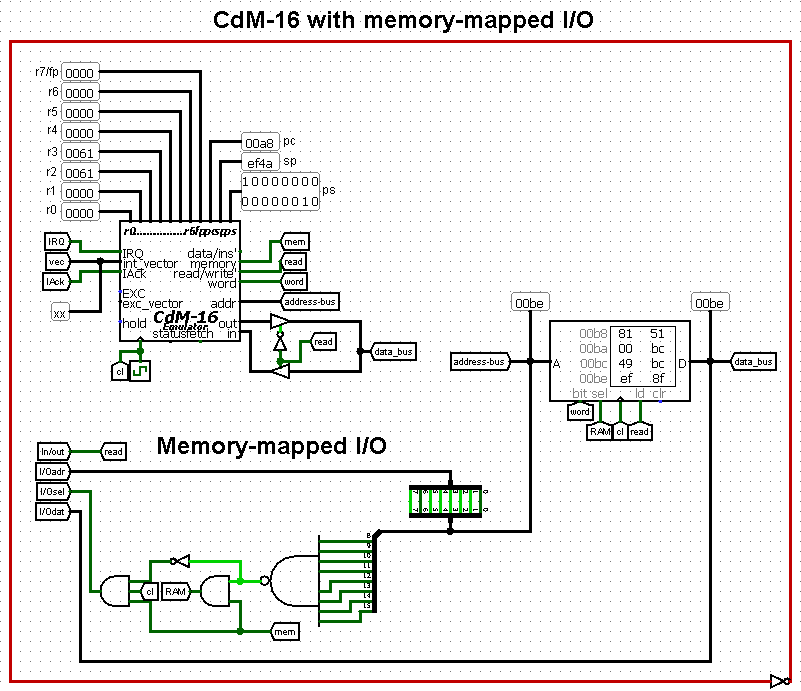
around is passed through a bit adder to count alive neighbors. This value goes to the select inputs of multiplexers, while born\_rule and survival\_rule go to the data bits. This determines whether the cell is alive in the current simulation cycle. These outputs pass through logic gates and enter a D flip-flop, from which they are then output as state (the cell's state).

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



## Processor

The CdM-16 processor operates on a Von Neumann architecture, meaning it uses a single address space for instructions and data. Additionally, a portion of the address space is allocated for memory-mapped I/O (addresses 0xff00 – 0xffff). It reads and writes data via I/Oadr (the I/O device address), I/Osel (the device selection signal), and I/Odat (the data bus). The processor manages data transfer between the keyboard, terminal, and video buffer.



# Software Description

## Entry Point *(\_start*)

\_start launches the main program code upon startup.

Upon system startup, the stack is initialized, and the *ISTATE* flag is raised, which is responsible for enabling keyboard interrupts. The prompt “> “ is displayed using the *\_print* subroutine, and the main program loop *\_main* is started.

|  |
| --- |
| \_start:  ldi r0, 0xff00  stsp r0  ldi r0, ISTATE  ldi r1, 1  stw r0, r1  ldi r0, in\_msg  jsr \_print  clr r0  clr r1  ei  jsr \_main  halt |

## Subroutine *\_print*

Responsible for sending keyboard characters to the terminal.

|  |
| --- |
| \_print:  push r1  push r2  ldi r1, CURR\_CHAR  while  ldc r0, r2  inc r0  tst r2  stays nz  st r1, r2  wend  pop r2  pop r1  rts |

## Main Loop (*\_main*)

*\_main* coordinates command processing.

*\_main* checks the value of the *cmdFlag* flag. A non-zero value invokes the command parsing function *main* from **parser.c**.

|  |
| --- |
| \_main:  if  ldi r0, cmdFlag  ldw r0, r0  tst r0  is nz  clr r0  di  jsr main  ei  ldi r0, cmdFlag  ldi r1, 0  stw r0, r1  clr r0  clr r1  fi  jsr \_main |

## Interrupt (*\_kb\_isr*)

*\_kb\_isr* is triggered when an **IRQ** signal is received from the **Interrupt Arbiter** and interrupt vector number 5. *\_kb\_isr* handles reading characters, separately processing Backspace and Enter keys, and placing characters into a 32-byte software circlular buffer.

*\_kb\_isr* reads a character and initiates the corresponding processing algorithm:

* If it is Backspace (0x08), the buffer is checked for characters. If there are characters, the character is sent to the terminal to erase the last entered character on the screen. The buffer end pointer is decremented (if the buffer is not empty).
* If it is Enter (0x0a), the character is sent to the terminal, and a terminating null is written to the buffer, marking the end of the line. The *cmdFlag* is set to 1, signaling *\_main* that the command is ready for processing.
* For other characters, the character is sent to the terminal. The buffer is checked for space. If full, an overflow message is displayed. Otherwise, the character is placed in the buffer, and the end pointer is incremented.

|  |
| --- |
| \_kb\_isr:  save  ldi r6, CURR\_CHAR  ldb r6, r2  ldi r7, 31 # mask for the ring buffer (size 32)  ldi r5, head  ldw r5, r4 # r4 = head  ldi r5, end  ldw r5, r1 # r1 = end  if  #check for backspace  ldi r0, 0x08  cmp r2, r0  is eq  cmp r1, r4  if  is ne  dec r1  and r7, r1  ldi r0, end  st r0, r1  stb r6, r2 # echo back to the console  fi  restore  rti  fi  stb r6, r2 # echo back to the console  # не backspace  inc r1  and r7, r1  if  cmp r1, r4  is ne  dec r1  and r7, r1  ldi r5, queue  if  # проверка на enter  ldi r0, 0x0a  cmp r2, r0  is eq  ldi r2, 0 # replace r2 with 0 (line terminator)  ldi r0, cmdFlag  ldi r3, 1  stw r0, r3  fi  stb r5, r1, r2 # put either a character or a 0  inc r1  and r7, r1  ldi r0, end  st r0, r1  else  ldi r0, len\_error  jsr \_print  jsr qInit  fi  restore  rti |

## Command Parsing (parser.c)

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

When the function executes, the entered string is read from the circular buffer into a temporary buffer called *cmd\_buffer* until a terminating null is found. Leading spaces are skipped, and the first word is extracted from *cmd\_buffer* into *command*.

|  |
| --- |
| void parse**()**  **{**  char cmd\_buffer**[**QSIZE **+** 1**];** // Buffer for commands from queue (+1 for NULL)  char command**[**QSIZE **+** 1**];** // Buffer for the command name  char **\***args**[**MAX\_CMD\_ARGS**];** // Array for pointers to arguments  int arg\_count **=** 0**;** // Number of arguments found  int len **=** 0**;**  int i **=** 0**;**  char **\***p**;**  **if** **(**head **==** end**)** **return;** // Queue is empty  // Reading a command from a queue in cmd\_buffer  int current **=** head**;**  **while** **(**current **!=** end **&&** len **<** QSIZE**)**  **{**  **if** **(**queue**[**current**]** **==** '\0'**)**  **{**  end **=** **(**current **+** 1**)** **%** QSIZE**;**  **break;**  **}**  cmd\_buffer**[**len**++]** **=** queue**[**current**];**  current **=** **(**current **+** 1**)** **%** QSIZE**;**  **}**  cmd\_buffer**[**len**]** **=** **NULL;**  qInit**();**  p **=** cmd\_buffer**;**  **while** **(\***p **==** ' '**)** p**++;** // Skip leading spaces  **if** **(\***p **==** **NULL)**  **{**  print**(**"> "**);**  **return;**  **}**  // Extracting the command name  i **=** 0**;**  **while** **(\***p **!=** ' ' **&&** **\***p **!=** **NULL** **&&** i **<** QSIZE**)**  command**[**i**++]** **=** **\***p**++;**  command**[**i**]** **=** **NULL;**  **while** **(\***p **==** ' '**)** p**++;**  // Parsing arguments (p points to the line after the command name)  arg\_count **=** parseArgs**(**p**,** args**,** MAX\_CMD\_ARGS**);**  ...  **}** |

The number of arguments is counted using *parseArgs()*, which splits the string into arguments by spaces and stores pointers to the beginnings of the arguments.

|  |
| --- |
| int parseArgs**(**char **\***p**,** char **\***args**[],** int args\_capacity**)**  **{**  int count **=** 0**;**  **while** **(\***p **&&** count **<** args\_capacity**)**  **{**  **while** **(\***p **==** ' '**)** p**++;**  **if** **(\***p **==** **NULL)** **break;**  args**[**count**++]** **=** p**;**  **while** **(\***p **!=** ' ' **&&** **\***p **!=** **NULL)** p**++;**  **if** **(\***p **==** ' '**)** **\***p**++** **=** **NULL;** // If found a space, replace it with NULL and go to the next symbol  **}**  **while** **(\***p **==** ' '**)** p**++;**  **if** **(\***p **!=** **NULL** **&&** count **<** args\_capacity**)**  **{**  // If there are still non-whitespace characters, this is considered an extra argument,  // even if it does not fit into the args array. We increase count so that checking  // for the exact number of arguments in the calling function works.  count**++;**  **}**  **return** count**;**  **}** |

After this, the extracted command is compared with the existing command names for subsequent invocation. If the command does not match any of the existing commands, an error message is returned.

|  |
| --- |
| **...**  **if** **(**StringCmp**(**command**,** "set"**))**  SetCmdWrapper**(**args**,** arg\_count**);**  **else** **if** **(**StringCmp**(**command**,** "rule"**))**  RuleCmdWrapper**(**args**,** arg\_count**);**  **else** **if** **(**StringCmp**(**command**,** "stop"**))**  StopCmdWrapper**(**args**,** arg\_count**);**  **else** **if** **(**StringCmp**(**command**,** "start"**))**  StartCmdWrapper**(**args**,** arg\_count**);**  **else** **if** **(**StringCmp**(**command**,** "fill"**))**  FillCmdWrapper**(**args**,** arg\_count**);**  **else** **if** **(**StringCmp**(**command**,** "clean"**))**  CleanCmdWrapper**(**args**,** arg\_count**);**  **else** **if** **(**StringCmp**(**command**,** "help"**))**  HelpCmdWrapper**(**args**,** arg\_count**);**  **else** **if** **(**StringCmp**(**command**,** "set-glider"**))**  SetGliderCmdWrapper**(**args**,** arg\_count**);**  **else** **if** **(**StringCmp**(**command**,** "set-bee-queen"**))**  SetBeeQueenCmdWrapper**(**args**,** arg\_count**);**  **else** **if** **(**StringCmp**(**command**,** "set-lwss"**))**  SetLWSSCmdWrapper**(**args**,** arg\_count**);**  **else** **if** **(**StringCmp**(**command**,** "set-hwss"**))**  SetHWSSCmdWrapper**(**args**,** arg\_count**);**  **else**  **{**  print**(**"Error: Unknown command '"**);**  print**(**command**);**  print**(**"'. Type 'help'.\n"**);**  **}**  print**(**"> "**);** // Prompt to enter the following command  **}** |
|  |

Each command calls its corresponding handler function, which checks the number of arguments for a mismatch with the required number of arguments for the called program (in this case, an error is returned). To extract numerical arguments, *my\_atoi\_safe()* is used, which also verifies the correctness of the entered arguments.

|  |
| --- |
| int my\_atoi\_safe**(**char **\***str**,** int **\***out**)**  **{**  int val **=** 0**;**  int found **=** 0**;**  **while** **(\***str **==** ' '**)** str**++;**  **if** **(!**isNum**(\***str**))** **return** 0**;**  **while** **(**isNum**(\***str**))**  **{**  **if** **(**val **>** **(**32767 **/** 10**))** **return** 0**;**  val **=** my\_mul**(**val**,** 10**)** **+** **(\***str **-** '0'**);**  str**++;**  found **=** 1**;**  **}**  **while** **(\***str **==** ' '**)** str**++;**  **if** **(\***str **!=** **NULL)** **return** 0**;** // If there is anything left after the number and spaces, the argument is invalid.  **\***out **=** val**;**  **return** found**;**  **}** |

**Краткое описание функций-обработчиков команд**

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

* The *StartCmdWrapper* function sets the *START\_STOP* flag to 1 to start the game and also manages the *gmState* variable for temporary game pauses when other functions are being used, allowing for subsequent game resumption.
* The *StopCmdWrapper* function does the opposite of *StartCmdWrapper*.
* The *FillCmdWrapper* function checks the passed arguments for correctness and calls *FillCmd*, passing it two sets of coordinates: for the top-left and bottom-right corners of the rectangle, as well as the value for filling. The *FillCmd* function assigns identical state values to cells within the user-specified rectangle, temporarily pausing the game.
* The *CleanCmdWrapper* function checks the entered arguments and passes a value of 1 to the *RESET* flag for a complete field clear.
* The *RuleCmdWrapper* function converts a string of digits into a corresponding bitmask for BORN/SURV, handling duplicate and unordered digits.
* The *SetCmdWrapper* function checks the entered arguments and calls *SetCommand*, passing it the cell's coordinates and value. The *SetCommand* function sets the passed value to the cell at the specified coordinates.
* The *SetGliderCmdWrapper* function checks the entered arguments and calls *SetGliderCmd*, passing the coordinates of the glider's top-left corner. The *SetGliderCmd* function calls *SetCommand* to fill the necessary cells for creating a glider, temporarily pausing the game.
* The *SetBeeQueenCmdWrapper* function is analogous to *SetGliderCmdWrapper*, but draws the "bee-queen" pattern.
* The *SetLWSSCmdWrapper* function is analogous to *SetGliderCmdWrapper*, but draws the "light-weight spaceship" pattern.
* The *SetHWSSCmdWrapper* function is analogous to *SetGliderCmdWrapper*, but draws the "heavy-weight spaceship" pattern.
* The *HelpCmdWrapper* function prints general help if no arguments are provided, or help for a specific command if a single argument – the command name – is passed.

# User Manual

1. Open the file main.circ in Logisim.
2. Load the program.img image into memory.
3. Select the maximum clock frequency.
4. Enable the clock.
5. Enter commands via the keyboard. To see the list of commands, type `help`. For help on a specific command, type `help <command name>`.

## List of Commands and Descriptions

1. set <x> <y> <val> — Set the value of cell (x, y).
2. rule <born> <sur> — Set 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 ((x1, y1), (x2, y2)) with the value <val>.
6. clean — Clear the entire field.
7. help — Display command descriptions. help [command] — Display help for a specific command.
8. set-glider <x> <y> — Draw a working glider at coordinates (x, y).
9. set-bee-queen <x> <y> — Draw a "bee-queen" pattern at coordinates (x, y).
10. set-lwss — Draw a "light-weight spaceship" at coordinates (x, y).
11. set-hwss — Draw a "heavy-weight spaceship" at coordinates (x, y).