Novosibirsk state university

Project

“Conway’s game of life”

Made by:

Tokarev M.K.

Kuchenkov S.A.

Kozlovskaya K.

2025

# Введение и описание

Данный проект реализует аппаратную и программную части "Игры Жизнь" Джона Конвея.

Система позволяет пользователю настроить состояние поля, запускать и останавливать игру, регулировать правила поля в любой момент симуляции, а также наблюдать за процессом смены поколений клеток.

Проект использует:

* CdM-16 на фон-неймановской архитектуре;
* Logisim и cdm-devkit.

Проект включает такие ключевые компоненты, как:

* Клавиатуру для ввода пользовательских команд;
* Терминал для отображения команд и ошибок пользователю;
* Процессор CdM-16 для обработки пользовательских команд;
* Видеобуфер для вычисления следующего состояния поля;
* Дисплей для отображения игрового поля (32\*32).

Данная реализация игры имеет несколько отличительных черт:

* Зацикленное (торическое) поле;
* Настраиваемые правила игры (рождение и выживание);
* Возможность изменять поле в любой момент;
* Реализация взаимодействия с пользователем через клавиатуру и терминал с помощью команд.
* Вывод в консоль ошибок и состояний исполнения команд.

# Описание работы

# Аппаратная часть

## Пользовательский ввод

Ввод осуществляется через инструмент "Клавиатура", который позволяет пользователю передавать команды и данные в систему.

cl — сигнал тактового генератора, который позволяет клавиатуре отправить символ, kb\_ctrl — сигнал разрешения отправки данных от декодера.

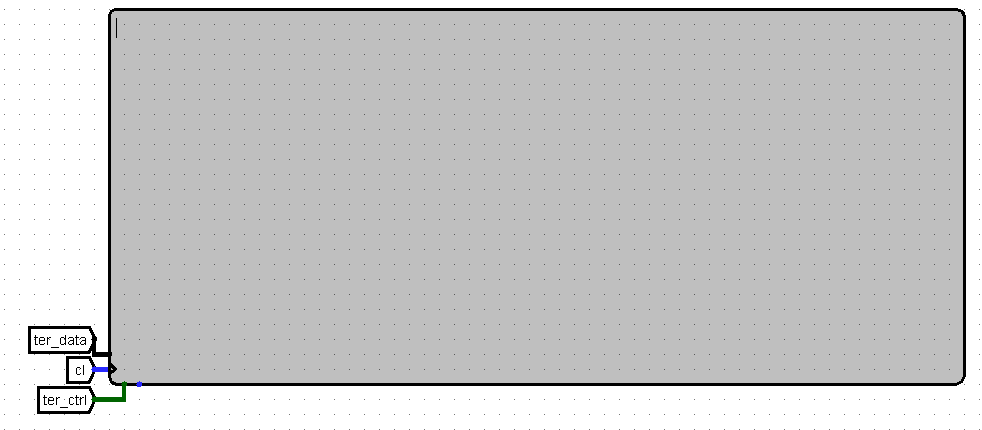
При нажатии клавиши клавиатура генерирует код символа и передаёт сигнал о готовности считывания символов.

Вывод из клавиатуры обеспечивают: kb\_data, передающая введённый символ за такт, kb\_ready, которая сообщает о наличии символов в клавиатуре.



Переданный символ обрабатывается процессором и отправляется на терминал.

Терминал принимает значения вводимых символов через ter\_data из обработчика клавиатуры, сигнал тактового генератора, бит разрешения записи ter\_ctrl.



## Обработчик клавиатуры и терминала

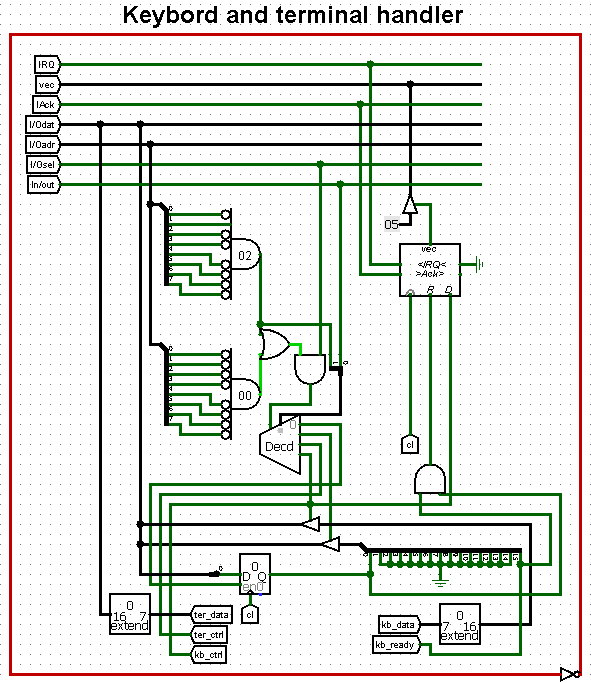
Обработчик клавиатуры принимает сигналы от клавиатуры, подготавливает данные для отправки на шину I/Odat (и дальнейшей отправки на кольцевой буфер), управляет генерацией запросов на прерывание.

Когда пользователь вводит данные, клавиатура выставляет код символа на kb\_data и сигнал готовности kb\_ready. Данные с kb\_data расширяются до 16 бит и отправляются через управляемый буфер, активируемый декодером, на шину I/Odat для чтения процессором.

Сигнал kb\_ready отправляется на схему Interrupt Arbiter, откуда генерируются сигналы IRQ, отправляется вектор прерывания по шине vec (вектор номер 5) и вызывается обработчик прерываний \_kb\_isr.

Для разрешения на получения прерываний в коде программы мы загружаем в переменную ISTATE (по адресу 0xff00) единицу, после чего регистр разрешения прерывания встаёт в значение 1, разрешая выполнить прерывание.

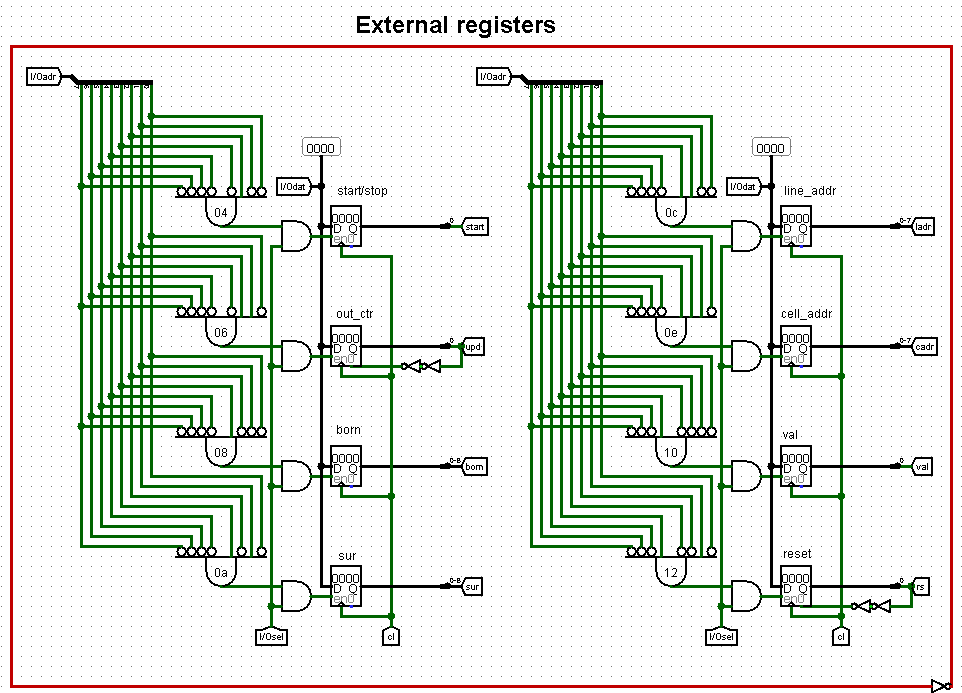
На шине I/Odat данные отправляются на ter\_data, урезанные до 7 бит, для отображения введенного символа на терминале. Декодер получает на разрешающий вход сигнал с I/Oadr и с I/Osel, который пропускается через вентили, а на выбирающий вход: на первый бит — значение in/out, а на второй — бит равенства сигнала I/Oadr с 0x02. Из декодера значения идут на управляющие буферы, регистр, ter\_ctrl и kb\_ctrl.



**Внешние регистры**

Также шина I/Odat отправляет данные на внешние регистры, если те находятся в нужных ячейках памяти, которые сравниваются с I/Oadr. Все внешние регистры находятся в зоне памяти отведенной для memory-mapped I/O, по показанным ниже адресам:

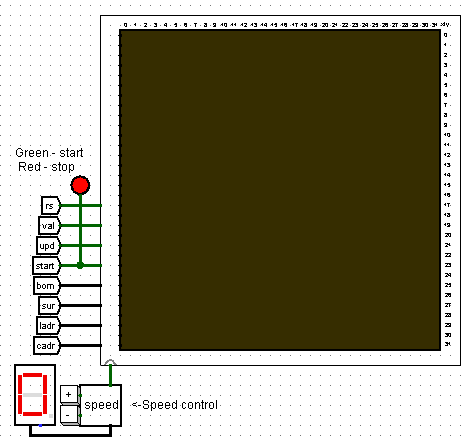
* 0x04 – Start/stop value (1 is start, 0 is stop);
* 0x06 – Asynchronous cell state update;
* 0x08 – Current born rule;
* 0x0a – Current survival rules;
* 0x0c – Line address for cell update;
* 0x0e - Cell address in line for cell update;
* 0x10 – Value for cell update;
* 0x12 – Reset bit.



## Видеобуфер

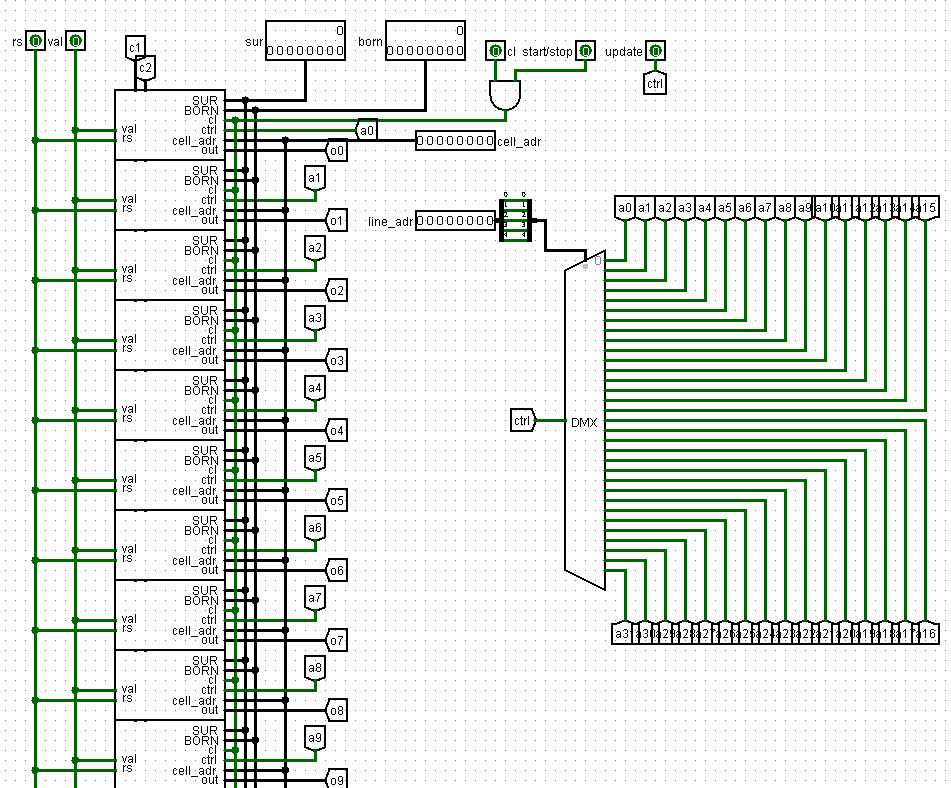
Видеобуфер принимает все значения от внешних регистров и значение тактового генератора, частота которого может быть дополнительно регулируется схемой speed\_ctrl.

Видеобуфер хранит текущее состояние игрового поля и просчитывает следующее, также он соединён с дисплеем и передаёт на каждый его вход необходимые значения.

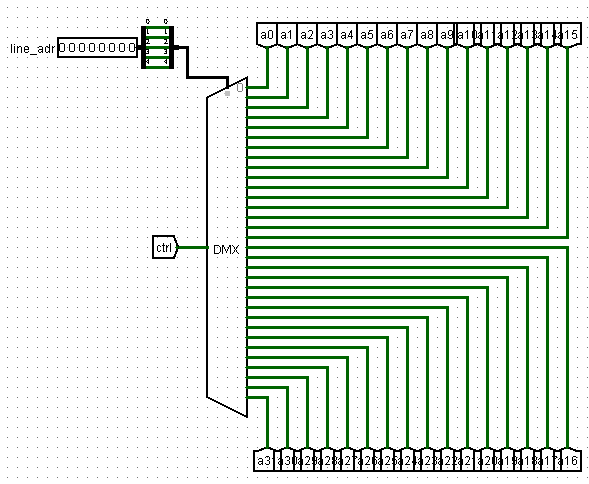


Входные значение видеобуфера передаются одинаковыми подсхемами "line", каждый из которых соответствует своей строкам поля, отвечает за хранение и обработку состояний клеток одной строки игрового поля. Все 32 подсхемы соединены между собой через up и down для учёта значений клеток в соседних строках.

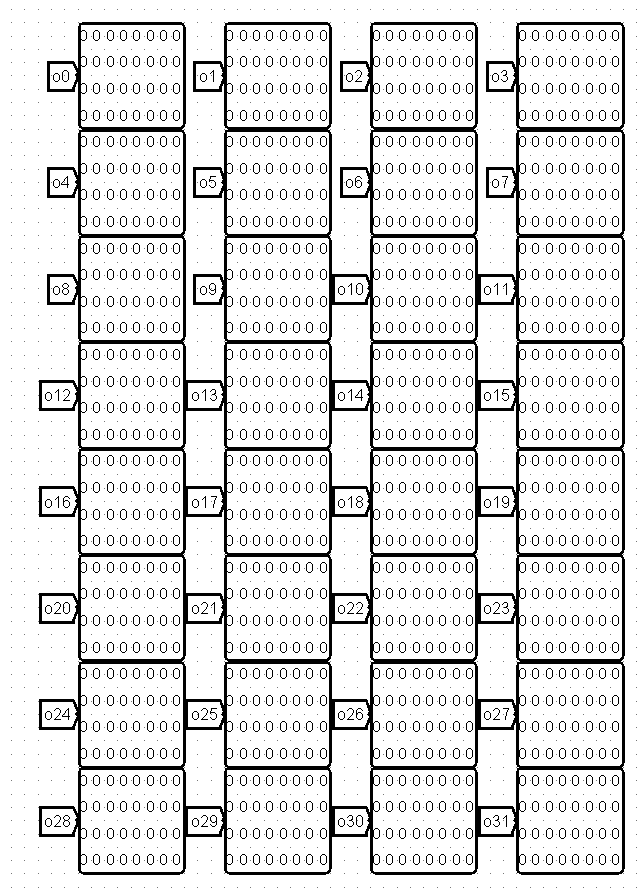
Каждая подсхема "line" выводит out – 32 битовый массив, соответствующий значениям всех 32 клеток во всех строках.



Расчёт значений ctrl, cell\_adr происходит благодаря демультиплексору. На выбирающий вход он принимает значения line\_adr, а на выход выдаёт значение update необходимой строке для редактирования.



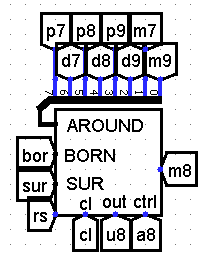
Каждая подсхема "line" имеет выход out, который передаёт значения всех клеток в строке и соединяется с соответствующей строкой на дисплее.



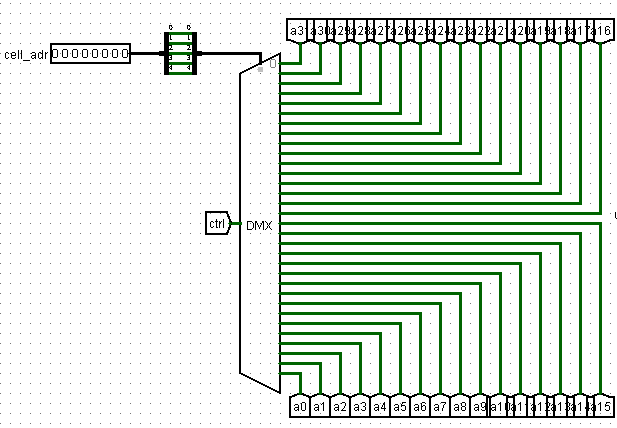
## Описание работы подсхемы "line"

Подсхема "line" отвечает за определение значений клетки в строке игрового поля.

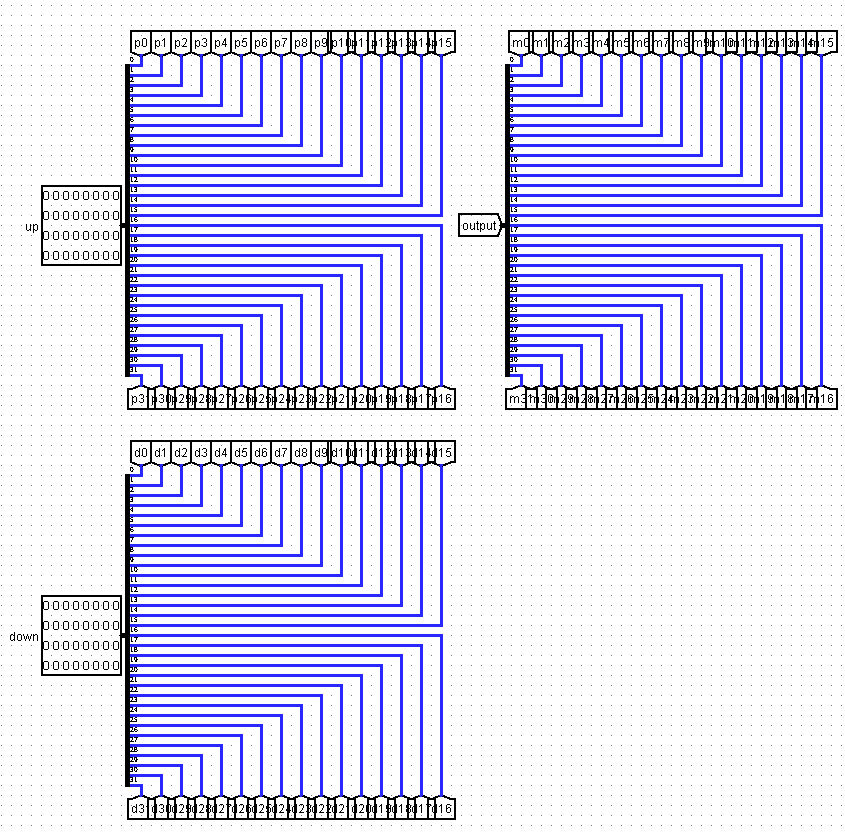
Значение каждой клетки обрабатывается через подсхему "cell". Здесь она принимает значения правил рождения и выживания (born и sur), сигнал очистки поля (rs) 8-битную маска из значений трёх верхних клеток, трёх нижних клеток, левой и правой клетки, значения намеренного изменения значения и значения для обновления состояния данной клетки. Каждая клетка обладает 1-битным выходом, соответствующему значению клетки.



Расчёт значений принудительного редактирования клетки выполняется через один демультиплексор. На принимающий вход он принимает адрес клетки, тем самым посылая сигнал update на нужную клетку.



Также в подсхеме “line” присутствует несколько разветвителей, которые нужны для корректной обработки соседей для каждой клетки.

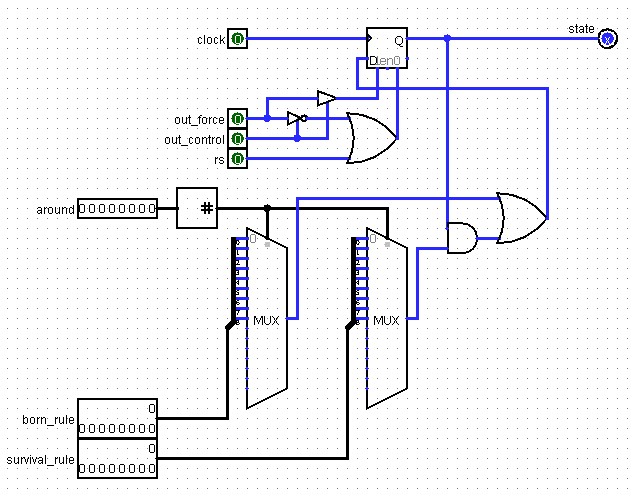


## Описание работы подсхемы "cell"

Подсхема cell принимает born\_rule и survival\_rule, out\_force, out\_control, rs (бит очистки поля) и маску соседей around.

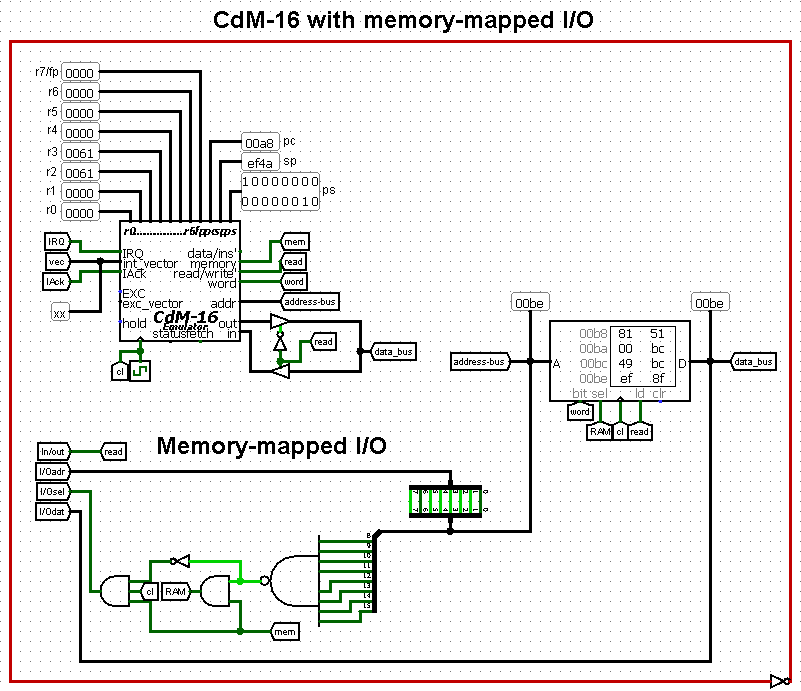
Вход around, пропускается через сумматор битов, получая количество живых соседних клеток. Оно идет в выбирающие входы мультиплексоров, а born\_rule и survival\_rule - в биты данных. Тем самым определяется жива ли клетка на этом такте симуляции. Эти выводы пропускаются через вентили и попадают на D-триггер, откуда потом выводятся в state (состояние клетки).

Таким образом, в случае если out\_control равен 1, то клетка принимает значение out\_force, иначе клетка принимает значение, исходя из состояния соседних клеток и обозначенных правил игры.



## Процессор

Процессор CdM-16 работает по фон-неймановской архитектуре, то есть с использованием единого адресного пространства для инструкций и данных. Также часть адресного пространства отдана для memory-mapped I/O (адреса 0xff00 – 0xffff). Он читает и записывает данные через I/Oadr — адрес устройства ввода/вывода, I/Osel — сигнал выбора устройства, I/Odat — шина данных. Процессор управляет передачей данных между клавиатурой, терминалом и видеобуфером.



# Описание работы программной части

## Точка входа *(\_start*)

\_start запускает основной код программы в момент запуска.

При включении системы происходит инициализация стека, а также поднимается флаг *ISTATE*, отвечающий за включение прерываний от клавиатуры, выводится строка “> “ (приглашение к вводу команды) с помощью подпрограммы *\_print*, запускается основной цикл программы *\_main*.

|  |
| --- |
| \_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 |

## Подпрограмма *\_print*

Отвечает за отправку на терминал символов с клавиатуры.

|  |
| --- |
| \_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 |

## Основной цикл (*\_main*)

*\_main* необходимы для координации обработки команд.

*\_main* проверяет значение флага *cmdFlag*, ненулевое значение которого вызывает функцию *main* парсинга команд из **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 |

## Прерывание (*\_kb\_isr*)

*\_kb\_isr* активируется при поступлении сигнала **IRQ** от **Interrupt Arbiter** и вектора прерывания номер 5. *\_kb\_isr* отвечает за чтение символа, отдельную обработку клавиш Backspace и Enter и помещение символов в программный кольцевой буфер, размер которого составляет 32 байта.

*\_kb\_isr* читает символ, и запускает соответствующий алгоритм обработки:

* Если это Backspace(0x08), то производится проверка буфера на наличие символов. Если в буфере есть символы — символ отправляется на терминал для стирания последнего введённого символа на экране. Указатель на конец буфера сдвигается назад (если буфер не пустой).
* Если это Enter(0x0a), то символ выводится на терминал, а в буфер записывается детерминирующий нуль, обозначающий конец строки. Флаг *cmdFlag* устанавливается в значение 1, что сигнализирует *\_main* о готовности обработки команды.
* Если это другой символ, то символ выводится на терминал. После этого выполняется проверка на наличие места в кольцевом буфере. Если оно закончилось, то выводится сообщение о переполнении строки. Если же место есть, то символ помещается в буфер и указатель конца буфера увеличивается на единицу.

|  |
| --- |
| \_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 |

## Парсинг команд (parser.c)

Файл parser.c содержит функцию *parse()* для обработки команд, вызываясь из *\_main*.

При исполнении функции введённая строка считывается из кольцевого буфера во временный буфер *cmd\_buffer*, пока не будет найдет символ детерминирующего нуля, пропускаются ведущие пробелы, а затем из *cmd\_buffer* извлекается первое слово, которое перемещается в 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**);**  ...  **}** |

Затем идёт подсчёт числа аргументов через функцию *parseArgs()*, разбиение строки на аргументы по пробелам и сохранение указателей на начала аргументов.

|  |
| --- |
| 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**;**  **}** |

После этого идёт сравнение с имеющимися названиями команд для их последующего вызова, в случае если команда не соответствует имеющемуся набору команд или, то возвращается сообщение об ошибке.

|  |
| --- |
| **...**  **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  **}** |
|  |

Каждая команда вызывает соответствующую ей функцию-обработчик, которая проверяет число аргументов на несоответствие необходимому числу аргументов вызываемой программы (в этом случае возвращается ошибка). Для извлечения числовых аргументов используется *my\_atoi\_safe()*, который также проверяет корректность введённых аргументов.

|  |
| --- |
| 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**

Взаимодействие с аппаратной частью осуществляется через отображённые в памяти регистры. Видеобуфер получает *CELL\_ADR, LINE\_ADR, VALUE, UPDATE, START\_STOP, BORN, SURV, RESET* для управления состояния поля и правил игры, а *CURR\_CHAR* отвечает за получение символов с клавиатуры и отправку символов на печать в терминал.

* Функция *StartCmdWrapper* задаёт значение 1 флагу *START\_STOP* для запуска игры и переменной *gmState* для временной остановки игры при использовании других функций и последующего возобновления игры.
* Функция *StopCmdWrapper* делает обратное функции *StartCmdWrapper*.
* Фунция *FillCmdWrapper* проверяет переданные аргументы на корректность и вызывает *FillCmd*, передавая ей две координаты: левого верхнего и правого нижнего углов прямоугольника, а также значение для заполнения. Функция *FillCmd* передаёт одинаковые значения состояния клеткам в переданном пользователем прямоугольнике, временно останавливая игру.
* Функция *CleanCmdWrapper* проверяет введенные аргументы передаёт флагу *RESET* значение 1 для полной очистки поля.
* Функция *RuleCmdWrapper* преобразует строку цифр в соответствующую битовую маску для BORN/SURV, обрабатывая повторяющиеся и неупорядоченные цифры.
* Функция *SetCmdWrapper* проверяет введенные аргументы и вызывает *SetCommand*, передавая ей координаты и значение клетки. Функция *SetCommand* устанавливает переданное значение клетке по заданной координате.
* Функция *SetGliderCmdWrapper* проверяет введенные аргументы и вызывает *SetGliderCmd*, передавая координаты левого верхнего угла глайдера. Функция *SetGliderCmd* вызывает *SetCommand* заполняя нужные клетки для создания глайдера, временно останавливая игру.
* Функция *SetBeeQueenCmdWrapper* – аналогична *SetGliderCmdWrapper*, только рисует паттерн “bee-queen”.
* Функция *SetLWSSCmdWrapper* – аналогична *SetGliderCmdWrapper*, только рисует паттерн “light-weight spaceship”.
* Функция *SetLWSSCmdWrapper* – аналогична *SetGliderCmdWrapper*, только рисует паттерн “light-weight spaceship”.
* Функция *SetHWSSCmdWrapper* – аналогична *SetGliderCmdWrapper*, только рисует паттерн “heavy-weight spaceship”.
* Функция HelpCmdWrapper печатает общую справку, если нет аргументов, или по конкретной команде, если передан один аргумент – имя команды.

# Руководство пользователя

1. Откройте файл main.circ в Logisim.
2. Загрузите в память образ program.img
3. Выберите максимальную тактовую частоту.
4. Включите такты.
5. Вы можете вводить команды через клавиатуру. Если вы хотите узнать набор команд введите ‘help’, если необходима справка по конкретной команде — ‘help <имя команды>’.

## Список команд и их описание

1. set <x> <y> <val> — установить значение клетки (x, y);
2. rule <born> <sur> — задать правила игры: рождение и выживание;
3. stop — остановить игру;
4. start — начать игру;
5. fill <x1> <y1> <x2> <y2> <val> — заполнить прямоугольник ((x1, y1), (x2, y2)) значением <val>;
6. clean — очистить всё поле;
7. help — вывести описание команд, help [command] — вывести описание конкретной команды;
8. set-glider <x> <y> — нарисовать работающий глайдер на координатах (x, y).
9. set-bee-queen <x> <y> — нарисовать bee-queen pattern на координатах (x, y).
10. set-lwss — нарисовать light-weight spaceship на координатах (x, y).
11. set-hwss — нарисовать heavy-weight spaceship на координатах (x, y).