Брус-16: учебная игровая приставка. Совместное программно-аппаратное проектирование

Научный руководитель: П.Н. Советов, к.т.н.
Студент: Кирилл Павлов, 4 курс

РТУ МИРЭА

Зачем?

Междисциплинарный учебный проект:

1. **Языки, компиляторы, виртуальные машины**: разработка программного эмулятора, ассемблера и DSL-компилятора.
2. **Низкоуровневое программирование**: разработка простых игр для архитектуры уровня микроконтроллера.
3. **Цифровая схемотехника**: проектирование специализированной вычислительной системы с управляющим процессором, графической подсистемой, устройствами ввода.

Почему игровая приставка?

- Просто спроектировать процессор мало — нужна практическая задача.
- Студентам интересны игры, в том числе в стиле ретро.
- Такие проекты, как CHIP-8 и разнообразные fantasy consoles, используются в обучении, но не охватывают вопросы цифрового проектирования.

Почему НОВАЯ игровая приставка?

Чтобы не пользоваться готовым. Чтобы было интересно создавать свои игры. Наконец, исходя из особых задач обучения.

За 2 учебные пары средний студент должен суметь реализовать одно из:

- Программный эмулятор приставки.
- Ассемблер.
- Простую игру.

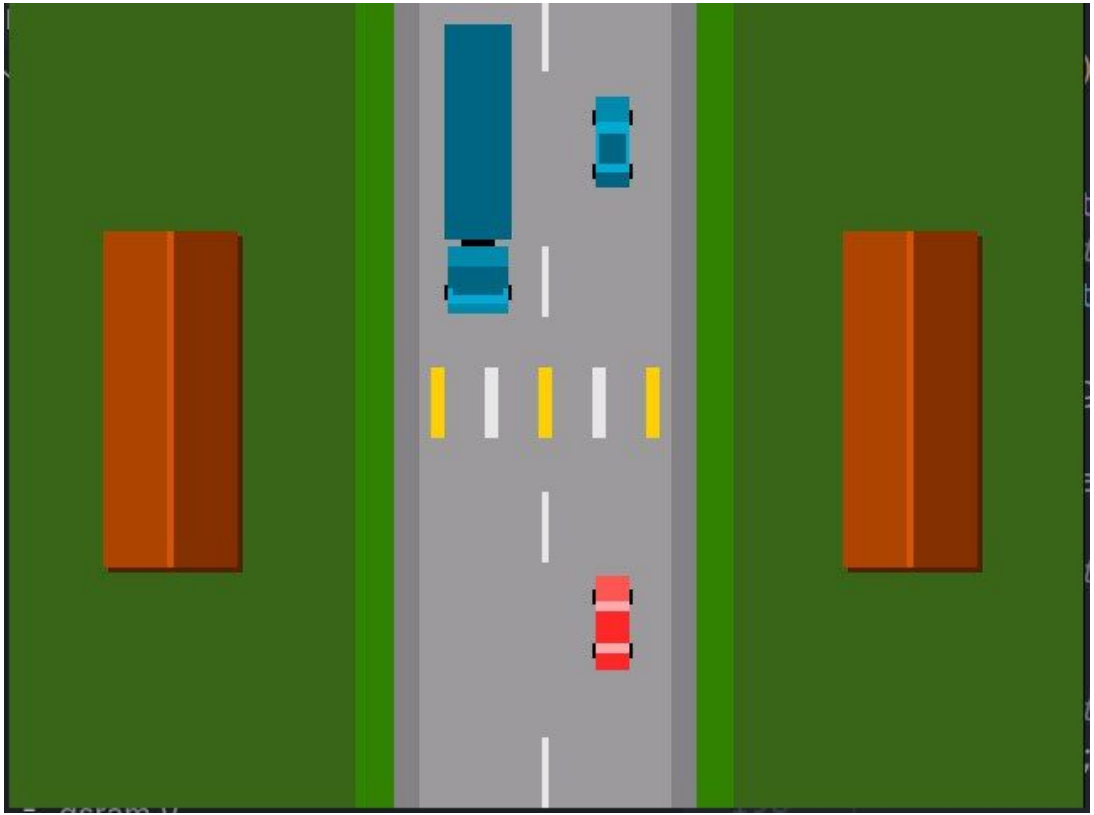
За 2-4 недели, одно из:

- DSL-компилятор.
- Реализацию Брус-16 на ПЛИС.

Основные элементы Брус-16

- **16-битный стековый CPU**. Стековая архитектура выбрана для упрощения создания компилятора.
- **GPU на прямоугольниках и без фреймбуфера**. Кадр строится из 64 аппаратных прямоугольников, для которых задаются X, Y, W, H, цвет и признак абсолютных координат. Это облегчает процесс создания игр даже для тех, кто не умеет рисовать. Нет фреймбуфера — меньше аппаратных ресурсов (мы хотим запустить приставку на платах уровня Tang Nano 9K).

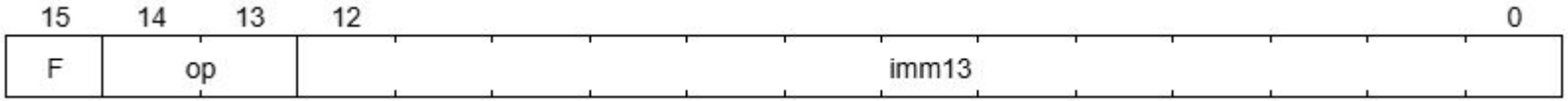
Как выглядят игры для Брус-16



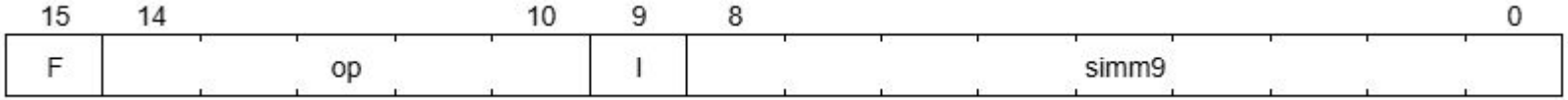
Процессор Брус-16

- 16-битная стековая архитектура с фреймами — локальными переменными в памяти данных.
- Код (8192 16-битных слова) и данные (8192) разделены.
- Два аппаратных стека: операций (32 ячейки) и возвратов (16 ячеек).
- Команда фиксированной длины 16 бит.
- Адресация только словами.
- Четыре регистра: PC, SP, RP, FP.
- Только абсолютные переходы.

Брус-16: система команд



JMP, JZ, CALL, PUSH_ADDR.



АЛУ: ADD, SUB, MUL, AND, OR, XOR, SHL, SHR, SHRA, EQ, NEQ, LT, LE, GT, GE, LTU.

Прочее: LOAD, STORE, LOCALS, SET_FP, ICALL, RET, PUSH_INT, PUSH_MR, POP, WAIT.

Реализация программного эмулятора: 138 SLOC.

Программная модель GPU (640x480x16bpp)

```
cursor_x, cursor_y = 0
rect_addr = RECT_MEM
for _ in range(RECT_NUM):
    is_abs = cpu.data[rect_addr + RECT_ABS]
    x = sext(cpu.data[rect_addr + RECT_X], 16)
    y = sext(cpu.data[rect_addr + RECT_Y], 16)
    w = cpu.data[rect_addr + RECT_W]
    h = cpu.data[rect_addr + RECT_H]
    color = cpu.data[rect_addr + RECT_COLOR]
    if is_abs:
        cursor_x = x
        cursor_y = y
    else:
        x += cursor_x
        y += cursor_y
    fill_rect(x, y, w, h, color)
    rect_addr += RECT_SIZE
```

Синхронизация CPU и GPU

Процессорная команда WAIT заставляет CPU “заснуть” после подготовки очередного кадра. Система далее “будит” процессор.

Это сопрограммная модель взаимодействия:

```
def setup():
    set_fp({KEY_MEM})
    background()
    while 1:
        update_frame()
        wait()
```

DSL-компилятор и ассемблер

```
def fact(n):
    if n < 2:
        return 1
    return n * fact(n - 1)
```

```
LABEL fact
LOCALS 1
SET_LOCAL 0
GET_LOCAL 0
PUSH_MR
LT 2
JZ L0
PUSH_INT 1
RET 1
LABEL L0
LABEL L1
GET_LOCAL 0
PUSH_MR
GET_LOCAL 0
PUSH_MR
SUB 1
CALL fact
MUL
RET 1
```

DSL-компилятор для подмножества Python. Реализация: 196 SLOC.

Реализация ассемблера: 96 SLOC.

Приветствуется участие в совместной разработке учебных материалов по Брус-16, а также использование этой архитектуры в университетских дисциплинах!

А теперь — слово Кириллу!

Как?

Итеративно, методом проб и ошибок.

- Код процессора менялся **4** раза.
- Код видеоядра менялся **3** раза.
- Множество правок в архитектуре по ходу разработки.
- Изучение предметной области по ходу разработки.

Архитектурные изменения:

- Фон-Неймовская архитектура -> Гарвардская.
- Относительные переходы -> абсолютные переходы.
- 1 общий формат команд -> 2 формата команд.
- Расширение АЛУ.
- Разделение LOAD на 2 инструкции.
- Отказ от аппаратного HIT из-за сложности реализации.
- Введение Wait.

Акт 1 Подготовка.

Дано: эмулятор на python, знания из курса информатики.

Подготовка:

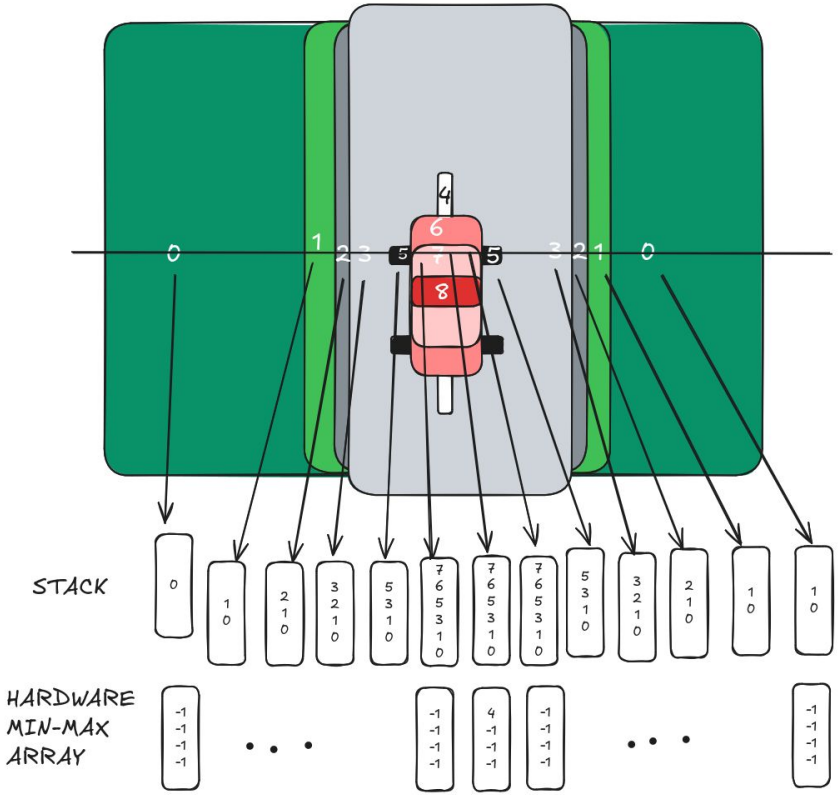
- Блог по реализации chip-8 на verilog.
- Спецификация verilog.
- ПЛИС Языки описания аппаратуры VHDL и Verilog, САПР, приёмы проектирования И.Е. Тарасов.
- Designing Video Games Hardware in Verilog (8bitworkshop).
- J1 и TINYCPU - модели стековых процессоров.
- Были рассмотрены и другие архитектуры, но они повлияли в меньшей мере.

Акт 2 Осмысление ограничений.

- Недостаток памяти для полноценного видеобуфера.
- Жёсткие ограничения на отрисовку кадра в реальном времени.
- Недостаток знаний об устройстве FPGA.
- Недостаток тактов для алгоритма художника для 1 линии.

Идея:

Последовательный алгоритм с ограничениями для отрисовки по 1 линии с 2-й буферизацией.



Акт 3 начало разработки

Первая итерация процессора:

- Единый формат команды.
- 3 такта на инструкцию.
- Запутанная последовательная логика процессора и вспомогательные регистры.
- Синхронные стеки *1R1W* с выделенным регистром для вершины стека.
- Преимущество: В симуляторе он работал!

Акт 4 середина разработки

Вторая итерация:

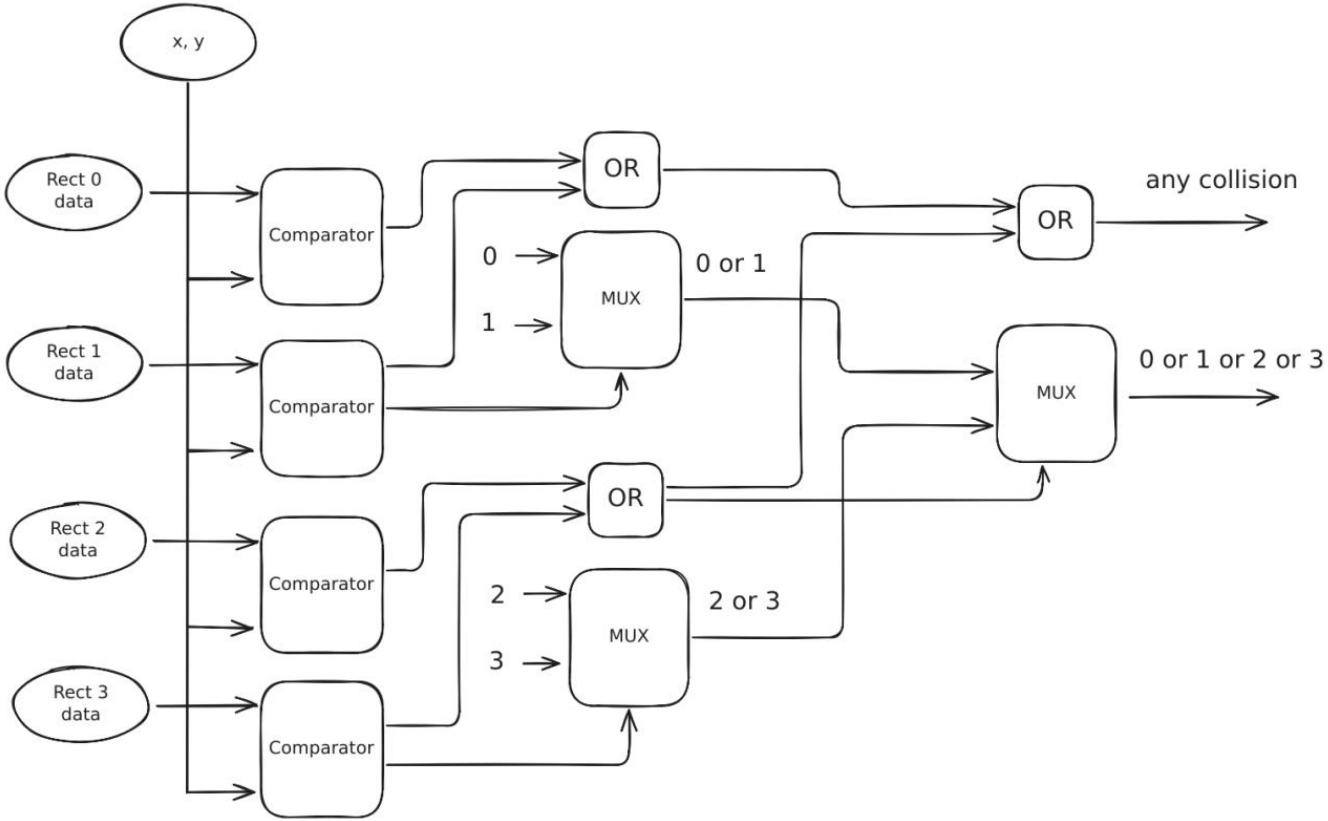
- 2 разных формата команды.
- Стеки *1R1W* с асинхронным чтением и выделенными регистры для вершины стека.
- 2 такта на инструкцию.
- Непростые архитектурные решения: чтение из синхронной памяти за 1 инструкцию, взятие 2-х аргументов со стека с учётом *1R1W.*
- Реализация GPU на основе параллельных компараторов и приоритетного шифратора, отрисовка в тот же так.
- Осмысление асинхронной логики.

Акт 5 выход на финишную прямую

Третья итерация:

- Разделение LOAD на LOAD и PUSH_MR.
- Стеки 2R1W с асинхронным чтением.
- 1 такт на инструкцию.
- Сокращение кода ядра на 150 строк кода.
- Независимая асинхронная логика, построенная вокруг регистров.
- Введение Wait.
- Процессор стал понятным и лаконичным.
- Переход видеоядра на бинарное дерево мультиплексоров.

Бинарное дерево мультиплексоров

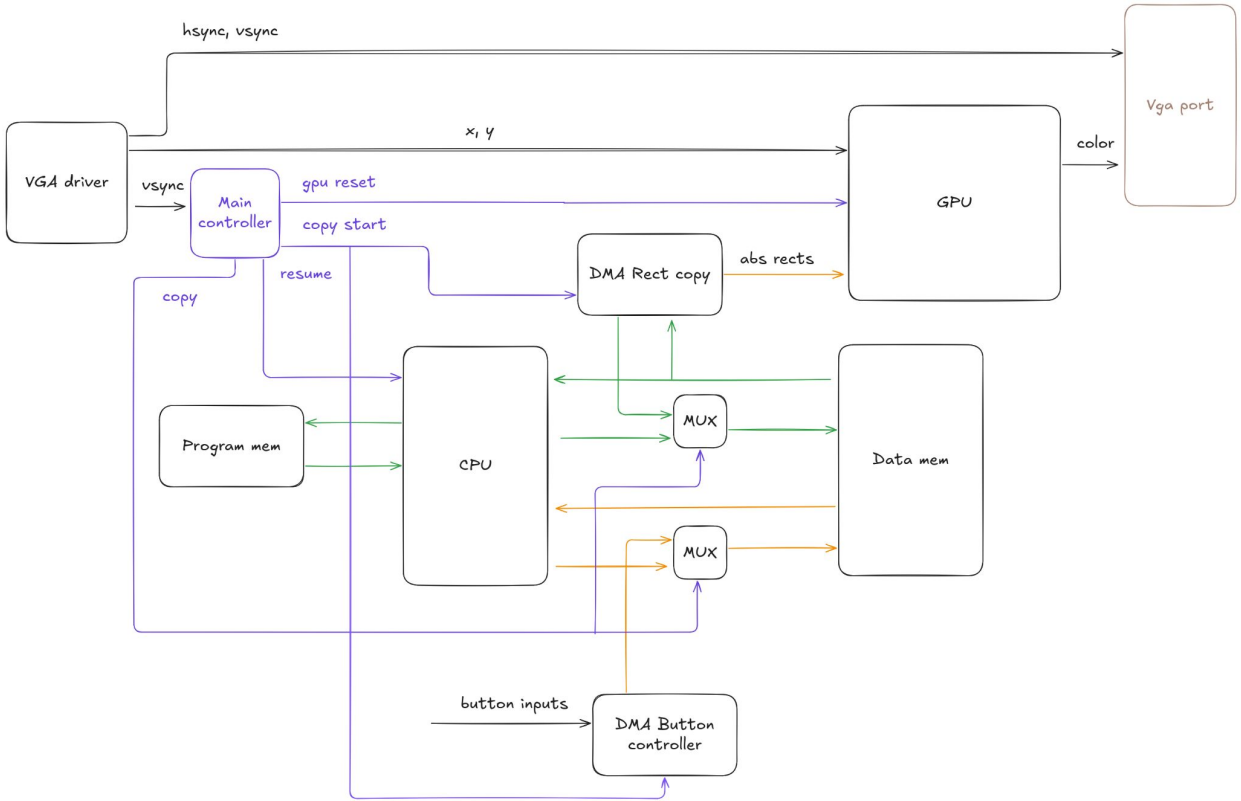


Процедура смены кадра

1. Процессор отработал и ожидает в состоянии wait.
2. Блок rect DMA копирует прямоугольники из системной памяти в память GPU с пересчётом координат.
3. Блок buttons DMA записывает в память состояние кнопок
4. Процессор возобновляет работу.

Процессор подготавливает новый кадр, а GPU отрисовывает текущий.

Архитектура



Дополнительные сложности

Неудобное тестирование на verilog -> Библиотеки cocotb.

Сложность тестирования системы целиком (на отрисовку кадра требуется 800x525 тактов) -> Модульное тестирование + сравнение дампов памяти verilog с памятью эмулятора на C.

Отсутствие знакового сравнения в 8bitworkshop IDE, 3-х битный цвет -> Симуляция вывода VGA с помощью Verilator и freeglut.

Отсутствие платы -> Verilator и freeglut.

Итог

- Код ядра ~ 460 SLOC.
- Асинхронная логика процессора, организованная вокруг регистров.
- 1 такт для всех инструкций.
- Раздельная память для инструкций и данных.
- Асинхронный GPU на основе параллельных компараторов и дерева мультиплексоров без видеобуфера и задержки.
- DMA контроллеры для копирования данных.
- Управление по VSYNC.

Демонстрация

verilator + vga
https://drive.google.com/file/d/1ygSRBZp9BoFQYHSV4DjXOJ6_blnPSIBf/view?usp=sharing

c + sdl
https://drive.google.com/file/d/1K1-AK2t0QhlZ_K6u_17JmzezoD-wY0sJ/view?usp=sharing

EXPLORER

BRUS-16

- __pycache__
- .gitignore
- a.out U
- architecture.svg U
- brus16_asm.py
- brus16_cfg.py M
- brus16_cpu.c
- brus16_cpu.h
- brus16_cpu.h.gch U
- brus16_cpu.h.pch U
- brus16_dsl.py
- brus16_isa.py
- brus16_sdl.c M
- bug.bin
- bug.py U
- car.svg
- data.bin
- game_code.v U
- game_data.v U
- game.bin
- game.py M
- LICENSE
- memory_at_0.txt U
- memory_at_1.txt U
- memory_at_2.txt U
- memory_at_3.txt U
- memory_at_4.txt U

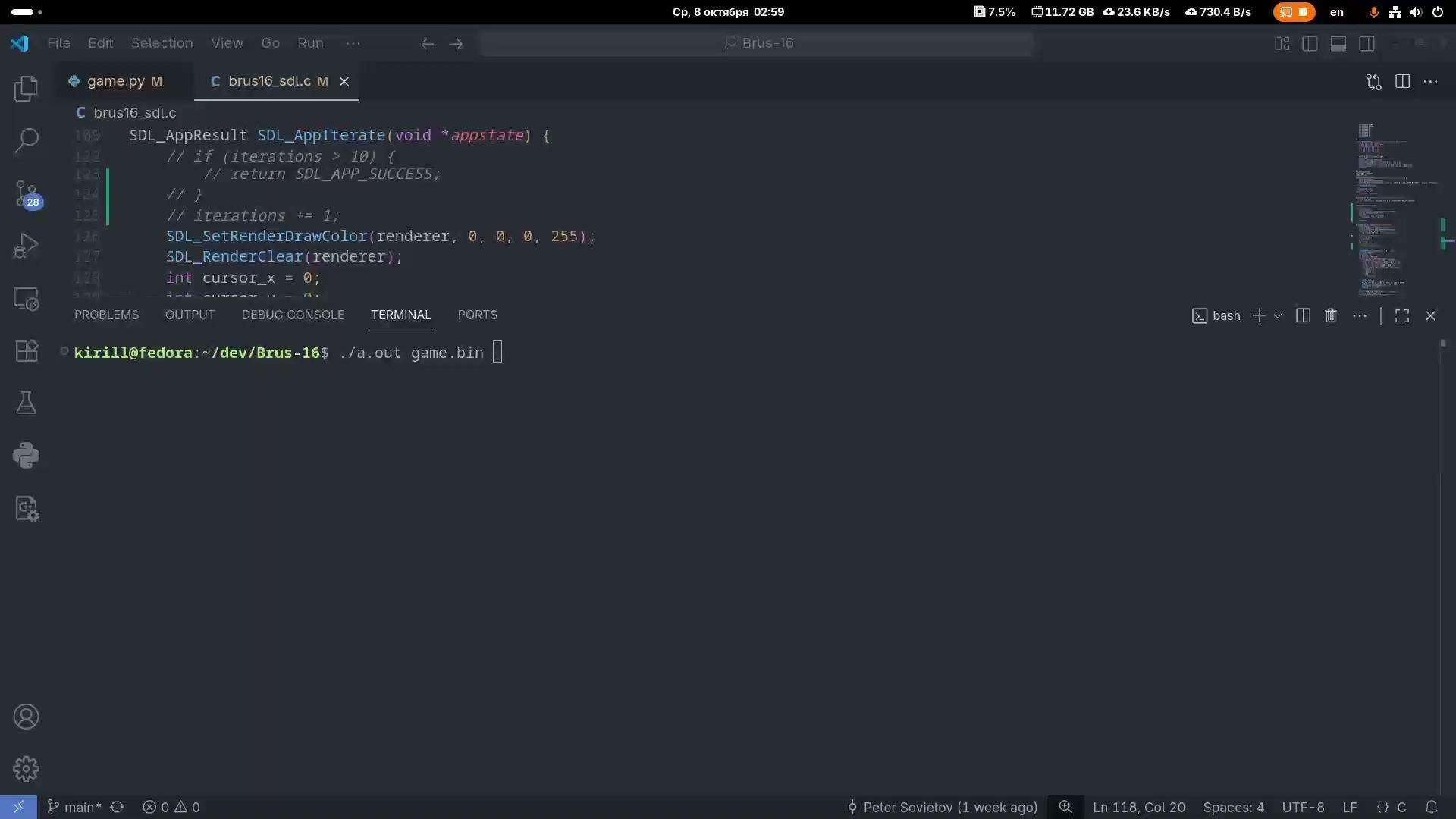
OUTLINE

TIMELINE

game.py M | game_data.v U

game.py > to_rgb565

```python
import ast
from brus16_dsl import comp
from brus16_asm import assemble, save
from brus16_cfg import *


def to_rgb565(r, g, b):
    r5 = (r >> 3) & 0x1f
    g6 = (g >> 2) & 0x3f
    b5 = (b >> 3) & 0x1f
    return (r5 << 11) | (g6 << 5) | b5


def get_rect_addr(rect, offs=0):
    return RECT_MEM + rect * RECT_SIZE + offs


GRASS_DATA = [1, 0, 0, SCREEN_W, SCREEN_H, to_rgb565(0x3a, 0x66, 0x1c)]

ROAD0_W = 225
ROAD1_W = 180
ROAD2_W = 150

ROAD0_DATA = [1, (SCREEN_W - ROAD0_W) // 2, 0, ROAD0_W, SCREEN_H, to_rgb565(0x33, 0x80, 0x00)]
ROAD1_DATA = [1, (SCREEN_W - ROAD1_W) // 2, 0, ROAD1_W, SCREEN_H, to_rgb565(0x80, 0x80, 0x80)]
ROAD2_DATA = [1, (SCREEN_W - ROAD2_W) // 2, 0, ROAD2_W, SCREEN_H, to_rgb565(0x99, 0x99, 0x99)]

LINE_RECT = 4
LINE_PARTS = 4
LINE_W = 4
LINE_H = 42
LINE_GAP = (SCREEN_H - LINE_PARTS * LINE_H) // (LINE_PARTS - 1)
LINE_DATA = []
for i in range(LINE_PARTS):
```


Ср, 8 октября 02:59
7.5%
11.72 GB
23.6 KB/s
730.4 B/s
en
File Edit Selection View Go Run
Brus-16
game.py M
brus16_sdl.c M
brus16_sdl.c
SDL_AppResult SDL_AppIterate(void *appstate) {
// if (iterations > 10) {
// return SDL_APP_SUCCESS;
// }
// iterations += 1;
SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255);
SDL_RenderClear(renderer);
int cursor_x = 0;
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
bash
kirill@fedora:~/dev/Brus-16$ ./a.out game.bin
main*
0 0
Peter Sovietov (1 week ago)
Ln 118, Col 20
Spaces: 4
UTF-8
LF
C