**Министерство образования и науки Российской Федерации**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**“САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ”**

**Факультет Программной инженерии и компьютерной техники**

**Образовательная программа Вычислительные системы и сети**

**Направление подготовки (специальность) 09.03.01 – Информатика и вычислительная техника**

# О Т Ч Е Т

об учебной практике по получению первичных профессиональных умений и навыков

Тема задания: *Разработка эмулятора для интерпретируемого языка программирования Chip-8 на основе Rust*

Обучающийся: *Коков Алексей Тимурович, гр. P3202*

Руководитель практики от университета: **Логинов Иван Павлович, Университет ИТМО, факультет программной инженерии и компьютерной техники, ассистент**

Практика пройдена с оценкой

Подписи членов комиссии:

_____ _____
(подпись)

_____ _____
(подпись)

_____ _____
(подпись)

Дата

Санкт-Петербург
2019

# Цель работы

Эмуляция — один из способов электронного архивирования текущих или устаревающих вычислительных систем. В такой трактовке целью эмуляции является точное воспроизведение оригинального цифрового окружения, что может быть труднодостижимым и затратным по времени, однако результат ценен из-за возможности достижения близкого сходства с изначальным цифровым объектом.

При помощи эмуляции пользователю открывается доступ к почти любому типу прикладного программного обеспечения или операционных систем на имеющейся платформе; при этом весь процесс работы происходит в таком же порядке, что и в оригинальном окружении. Это может быть весьма полезным, так в некоторых случаях приобретение оригинального обеспечения может быть невозможным: высокая цена, редкость или же полное отсутствие в продаже.

Целью данной работы явилось получение знаний о методиках описания архитектур и построения эмуляторов, а также их использование для воссоздания виртуальной машины на примере интерпретируемого языка программирования Chip-8. Полученный теоретический и практический опыт я надеюсь применить в дальнейшей профессиональной деятельности.

Как правило, эмулятор состоит из нескольких модулей, каждый из которых соответствует отдельной эмулируемой подсистеме оригинального устройства. В наиболее общем случае эмулятор состоит из следующих блоков, которые должны быть описаны программно:

- модуль эмуляции центрального процессора;
- модуль эмуляции подсистемы памяти;
- модули эмуляции различных устройств ввода-вывода.

В связи с этим были установлены следующие задачи:

- определение спецификаций физической архитектуры/виртуальной машины;
- описание приведенной архитектуры в предоставленной руководителем нотации описания архитектур;
- выбор программного обеспечения для создания эмулятора;
- программная реализация и проверка работоспособности.

# 1. Исследование теоретической составляющей и написание конспекта

Перед созданием виртуальной машины для эмуляции деятельности Chip-8 в первую очередь необходимо было исследовать спецификации данного языка в целях установления требуемых для разработки ресурсов и возможности реализации потенциальных уникальных особенностей.

В ходе прочтения теоретического материала была найдена информация о требованиях языка в отношении таких элементов, как память, устройства ввода-вывода, используемые регистры, а также некоторых свойств (таймеры).

## *1.1. Краткое описание*

Chip-8 – интерпретируемый язык программирования, разработанный Джозефом Вайсбекером (1932-1990) в середине 1970-х годов. Был создан для того, чтобы обеспечить более простую разработку видеоигр для микрокомпьютеров, таких как COSMAC VIP, DREAM 6800 и ETI 660.

Компьютеры, где использовался данный язык, как правило, использовали телевизор в качестве дисплея, имели 1-4 КБ ОЗУ (оперативное запоминающее устройство или *оперативная память*) и использовали 16-клавишную шестнадцатеричную клавиатуру для ввода. Интерпретатор занимал всего 512 байт памяти, а программы, вводившиеся в компьютер в шестнадцатеричном формате, были еще меньше.

## *1.2. Спецификации*

*Память*

Chip-8 способен использовать память ОЗУ объемом до 4 КБ – от 0x000 (0) до 0xFFF (4095). Первые 512 байт (0x000 – 0x1FF) – место, где находится интерпретатор, и ни в коем случае не используемое программами. Большинство программ начинаются с местоположения 0x200 (512), но некоторые начинаются с 0x600 (1536). Такие программы предназначены для ETI 660. Верхние 256 байт (0xF00 – 0xFFF) используются для обновления дисплея, а 96 байт до этого (0xEA0 – 0xEFF) зарезервированы для стека вызовов, внутреннего использования и других переменных.

*Регистры и стек*

Таблица 1. Используемые регистры

| Функциональная группа | Имя | Описание | Примечания |
|---|---|---|---|
| Общего назначения | V0-VE | Регистры общего назначения, используемые для хранения каких-либо данных | 8-разрядный |
| | VF | Регистр общего назначения, используемый для хранения состояний различных флагов в некоторых инструкциях (подробнее см. в разделе «Набор инструкций») | 8-разрядный, не должен использоваться программами |
| | I | Регистр, применяемый несколькими кодами операций для хранения адресов памяти | 16-разрядный, но используется только младшие 12 бит |
| Специализированные | DT | Регистр для таймера задержки | 8-разрядные; активны, пока в них содержится ненулевое значение. Подробнее см. в разделе «Таймеры» |
| | ST | Регистр для таймера звука | |
| Псевдорегистры | PC | Регистр, хранящий в себе текущий исполняемый адрес | 8-разрядные, недоступны для программ |
| | SP | Регистр, используемый для указания на верхний адрес стека | |

Стек представляет собой массив из шестнадцати 16-битных значений. В нем хранятся адреса, к которым интерпретатор должен вернуться после выполнения той или иной подпрограммы. Ранее в стеке аллоцировалось 48 байт, и позволялось до 24 уровней вложения, в более же современных имплементациях по умолчанию предоставляется 16 уровней вложения.

*Графика*

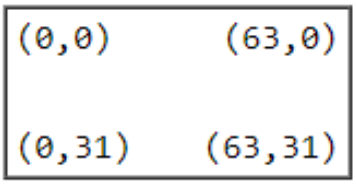


*Рисунок 1 - формат изображения*

Оригинальное разрешения дисплея Chip-8 составляет 64x32 пикселя, цвет монохромный. Графика выводится на экран с помощью спрайтов – группы байтов, являющихся двоичным представлением желаемой картинки. Спрайты могут иметь размер до 15 байт (ширина – до 8 пикселей, высота – до 15 пикселей).

| "0" | Binary | Hex |
|---|---|---|
| **** | 11110000 | 0xF0 |
| * * | 10010000 | 0x90 |
| * * | 10010000 | 0x90 |
| * * | 10010000 | 0x90 |
| **** | 11110000 | 0xF0 |

*Рисунок 2 - пример представления спрайтом символа "0"*

Цвет пикселей меняется следующим образом: установленные пиксели спрайта изменяют цвет соответствующего пикселя экрана, тогда как неустановленные пиксели ничего не делают. Флаг переноса в регистре VF устанавливается в 1, если любые пиксели экрана изменяются с установленного на неустановленное состояние при рисовании спрайта, иначе – в 0. Это используется для обнаружения коллизий.

Программы могут также ссылаться на группу спрайтов, представляющих шестнадцатеричные цифры от 0 до F. Эти спрайты имеют размер в 5 байт (8x5 пикселей). Они должны храниться в области памяти, отведенной для интерпретатора (0x000 – 0x1FF).

*Ввод данных*

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | C |
| 4 | 5 | 6 | D |
| 7 | 8 | 9 | E |
| A | 0 | B | F |

*Рисунок 3 - вид клавиатуры*

Ввод осуществляется с помощью шестнадцатеричной клавиатуры с 16 клавишами в диапазоне от 0 до F. Клавиши 2, 4, 6, 8 используются для задания направления.

Для считывания ввода используется 3 кода операций: один пропускает инструкцию, если нажата определенная клавиша, а другой делает то же самое, если конкретная клавиша не нажата; третий ожидает нажатия клавиши, а затем сохраняет его в одном из регистров данных.

*Таймеры*

Chip-8 предоставляет два таймера: задержки и звука. Оба таймера активны, пока соответствующий им регистр не равен нулю. Обновление идет с частотой 60 Гц.

Таймер задержки предполагает использование для определения времени событий в играх. Его значение может быть установлено и прочитано.

Таймер звука используется для воспроизведения звука зуммером (*buzzer*, подобно звонку), пока значение регистра не равно нулю. Воспроизводимый звук имеет только один тон, частота которого определяется автором интерпретатора.

### *1.3. Набор инструкций*

Chip-8 имеет 35 инструкций (кодов операций) размером 2 байта каждая. Хранится в формате big-endian.

Таблица 2. Список кодов операций

| Опкод | Сокращение | Описание | Примечания |
|---|---|---|---|
| 0NNN | **SYS** *addr* | Перейти к процедуре машинного кода по адресу NNN. | Используется только на старых компьютерах, в большинстве современных интерпретаторов игнорируется. |
| 00E0 | **CLS** | Очистить экран. | |
| 00EE | **RET** | Возврат из подпрограммы. | Интерпретатор устанавливает счетчик программ на адрес в верхней части стека, затем вычитает 1 из указателя стека. |
| 1NNN | **JP** *addr* | Перейти по адресу NNN. | |
| 2NNN | **CALL** *addr* | Вызвать подпрограмму, начинающуюся по адресу NNN. | Интерпретатор увеличивает указатель стека, а затем помещает текущий PC на вершину стека. PC затем устанавливается на NNN. |
| 3XNN | **SE V***x***,** *byte* | Пропустить следующую инструкцию, если Vx == NN. | Если равенство установлено, PC увеличивается на 2. |
| 4XNN | **SNE V***x***,** *byte* | Пропустить следующую инструкцию, если Vx != NN. | |
| 5XY0 | **SE V***x***, V***y* | Пропустить следующую инструкцию, если Vx == Vy. | |
| 6XNN | **LD V***x***,** *byte* | Vx = NN. | |
| 7XNN | **ADD V***x***,** *byte* | Vx += NN. | |
| 8XY0 | **LD V***x***, V***y* | Vx = Vy. | |
| 8XY1 | **OR V***x***, V***y* | Vx = Vx OR Vy. | OR, AND, XOR здесь – побитовые. |
| 8XY2 | **AND V***x***, V***y* | Vx = Vx AND Vy. | |
| 8XY3 | **XOR V***x***, V***y* | Vx = Vx XOR Vy. | |
| 8XY4 | **ADD V***x***, V***y* | Vx += Vy. | VF служит хранителем флага переноса (если результат больше 8 бит, VF = 1, иначе – 0) |
| 8XY5 | **SUB V***x***, V***y* | Vx -= Vy. | VF служит хранителем флага “not borrow” (если Vx>Vy, VF = 1, иначе – 0) |
| 8XY6 | **SHR V***x* **{, V***y***}** | Vx = Vx SHR 1. | Если младший бит Vx – 1, то VF = 1, иначе – 0. Затем Vx делится на 2. |
| 8XY7 | **SUBN V***x***, V***y* | Vx = Vy - Vx. | VF служит хранителем флага “not borrow” (если Vy>Vx, VF = 1, иначе – 0) |

| | | | |
|---|---|---|---|
| 8XYE | **SHL V*x* {, V*y*}** | Vx = Vx SHL 1. | Если старший бит Vx – 1, то VF = 1, иначе – 0. Затем Vx умножается на 2. |
| 9XY0 | **SNE V*x*, V*y*** | Пропустить следующую инструкцию, если Vx != Vy. | Если равенство установлено, PC увеличивается на 2. |
| ANNN | **LD I,** *addr* | I = NNN. | |
| BNNN | **JP V0,** *addr* | Перейти по адресу V0 + NNN. | |
| CXNN | **RND V*x*,** *byte* | Vx = random byte AND NN. | Интерпретатор генерирует случайный байт (значение от 0 до 255). |
| DXYN | **DRW V*x*, V*y*,** *nibble* | Отобразить спрайт из адреса, хранящегося в I размером N байт начиная с (Vx, Vy).<br>Интерпретатор считывает из памяти n байтов, начиная с адреса, хранящегося в I. Эти байты затем отображаются в виде спрайтов на экране в координатах (Vx, Vy). Спрайты выводятся на экран. | Если это приводит к удалению каких-либо пикселей, VF устанавливается на 1, в противном случае - на 0. Если спрайт расположен так, что его часть находится за пределами координат дисплея, он переносится на противоположную сторону экрана. |
| EX9E | **SKP V*x*** | Пропустить следующую инструкцию, если нажата клавиша со значением Vx. | Если равенство установлено, PC увеличивается на 2. |
| EXA1 | **SKNP V*x*** | Пропустить следующую инструкцию, если не нажата клавиша со значением Vx. | |
| FX07 | **LD V*x*, DT** | Vx = DT. | |
| FX0A | **LD V*x*, K** | Ожидать нажатия клавиши, затем сохранить значение нажатой клавиши в Vx. | Выполнение приостанавливается до нажатия клавиши. |
| FX15 | **LD DT, V*x*** | DT = Vx. | |
| FX18 | **LD ST, V*x*** | ST = Vx. | |
| FX1E | **ADD I, V*x*** | I += Vx. | |
| FX29 | **LD F, V*x*** | Установить I в местоположение спрайта для символа, содержагощеся в Vx. | Местоположение должно быть в области памяти, отведенной для интерпретатора (0x000 – 0x1FF). |
| FX33 | **LD B, V*x*** | Сохранить двоично-закодированное десятичное значение, хранящееся в Vx, по ячейкам памяти I, I+1 и I+2. | Интерпретатор вычисляет десятичное значение Vx и помещает сотни по адресу I, десятки – (I+1), единицы – (I+2). |
| FX55 | **LD [I], V*x*** | Сохранить значения регистров V0-Vx в память, начиная с адреса, указанного в I. | Смещение увеличивается на 1 для каждого записанного значения, но само I остается неизменным. |
| FX65 | **LD V*x*, [I]** | Заполнить регистры V0-Vx значениями из памяти, начиная с адреса, указанного в I. | |

### *1.4. Описание в приведенной нотации*

Нотация описывает архитектуру как набор следующих элементов:

• регистров;
• представлений регистров (отображение некоторых частей регистров, например, половины 16-байтового регистра);
• модели памяти;
• инструкций и их мнемоник.

Тесты описания проводились на примере всех доступных в Chip-8 команд. Результаты тестирования показали, что описание корректно.

Часть описания архитектуры приведена ниже в листинге 1, часть тестов – в листинге 2.

*Листинг 1 – часть описания архитектуры*

```
instruction se_byte = {0011, reg_as_param as vx, n8 as byte};
instruction sne_byte = {0100, reg_as_param as vx, n8 as byte};
instruction se_vx_vy = {0101, reg_as_param as vx, reg_as_param as vy,
0000};
instruction ld_vx_byte = {0110, reg_as_param as vx, n8 as byte};
instruction add_vx_byte = {0111, reg_as_param as vx, n8 as byte};
instruction ld_vx_vy = {1000, reg_as_param as vx, reg_as_param as vy,
0000};
instruction or_vx_vy = {1000, reg_as_param as vx, reg_as_param as vy,
0001};
```

*Листинг 2 – часть теста описания архитектуры*

```
LD VC, K
LD DT, VD
LD ST, V1
LD F, VA
LD B, VF
LD [I], VE
LD VE, [I]
ADD V3, 255
ADD V2, V4
ADD I, VD
SUB V7, VB
SUBN VA, VB
OR V1, V2
AND V0, VE
XOR VB, VC
SHR V0, V0
SHL V5, V6
RND V9, 255
DRW V5, V6, 0x04
SKP V8
SKNP V8
```

## 2. Программная реализация

Для разработки эмулятора Chip-8 над C++ было отдано предпочтение языку программирования Rust по следующим причинам:

- встроенная система управления пакетами Cargo

  В C++ при наличии зависимостей от не встроенных по умолчанию библиотек пользователь должен устанавливать их вручную, в то время как Rust при указании библиотек в Cargo.toml загружает и компилирует их самостоятельно. Использование модулей (ключевое слово mod) также облегчает работу в сравнении с include.

- улучшенное управление памятью

  В сравнении с C/C++, управление памятью в Rust более безопасно; например, Rust использует «умные» указатели – их отличие от обычных в том, что «умный» указатель следит за тем, когда память, на которую он ссылается, больше не используется (out of scope), и освобождает ее. В это же время в C/C++ с использованием обычных указателей возникают две проблемы, за которыми пользователь должен следить самостоятельно – утечка памяти (занятие памяти неиспользуемыми данными) и «висящий указатель» (при освобождении памяти старый указатель будет ссылаться на невалидные данные, но компилятор не будет об этом предупреждать). Подобные ошибки предотвращены в Rust.

- улучшенная инициализация данных

  В Rust все переменные должны быть проинициализированы при их объявлении, при этом у каждого типа (даже примитивных, в отличие от C) есть значения по умолчанию. В связи с этим не найдется таких переменных или объектов, которые не инициализированы.

- исключение «гонки за данными» - во время компиляции Rust при помощи встроенного т.н. «контролера занятий» следит за тем, чтобы одна и та же область данных не перезаписывалась параллельно

- удобная кросс-компиляция - можно компилировать проекты без дополнительного дистрибутива и вне зависимости от ОС, опять же, с использованием Cargo

- схожесть в синтаксисе с C/C++ - для более легкого перехода к его изучению

- большой выбор библиотек (в т.ч. разработанных пользователями), некоторые из которых перешли из C/C++ в Rust.

После выбора языка программирования состояла задача реализовать на нем три модуля/блока эмулятора, упомянутые в задачах: модуль эмуляции ЦП, подсистемы памяти и устройств ввода-вывода.

### *2.1. Модуль подсистемы памяти*

Одной из самых простых задач было реализовать модуль подсистемы памяти.

*Листинг 3 – система памяти*

```
pub struct Vm {
    pub opcode: u16,
    pub ram: [u8; RAM_SIZE],

    /// CPU registers (V0-VF)
    pub v: [u8; DATA_REGISTERS_COUNT],

    /// Index register and program counter
    pub i: u16,
    pub pc: usize,

    /// Screen: 64 x 32 pixels
    pub screen: [bool; SCREEN_PIXELS],

    /// HEX based keypad (0x0-0xF)
    pub key_states: [bool; KEYS_COUNT],

    /// Timer registers
    pub delay_timer: u8,
    pub sound_timer: u8,

    /// Stack and stack pointer
    pub stack: [u16; STACK_SIZE],
    pub sp: usize,

    pub draw_flag: bool,

    timer_refresh_cd: u8
}
```

По порядку:

- opcode – в Chip-8 каждый код операции имеет размер 2 байт; u16 – unsigned 16-bit (2 byte) integer.
- ram – память, размер – 4096 байт (4 КБ).
- v – набор регистров V0-VF (16 штук).
- i – индексный регистр (размер – 16 бит, но используются только младшие 12).
- pc – регистр, указывающий на текущий исполняемый адрес. В самом начале равен адресу 0x200 (512).
- screen – массив, хранящий в себе состояние 2048 (64*32) пикселей (1 – белый, 0 – черный).
- key_states – массив, хранящий в себе состояние клавиш на клавиатуре (1 – нажата, 0 – отжата).
- delay_timer – таймер задержки.
- sound_timer – таймер звука.
- stack – массив стека размером в 16 элементов по 2 байта каждый.
- sp – регистр, хранящий в себе указатель на вершину стека.
- draw_flag – переменная, указывающая, происходит ли сейчас перерисовка пикселей.
- timer_refresh_cd – переменная, представляющая собой счетчик до следующего обновления таймеров.

### *2.2. Модуль эмуляции центрального процессора (интерпретатор)*

Следующей задачей предстояло реализовать модуль эмуляции центрального процессора. Для этого необходимо было разобраться, что должно происходить во время эмуляции за каждый цикл:

- получение кода операции;
- трансляция опкода и его выполнение;
- обновление таймеров.

*Листинг 4 - пример трансляции опкода:*

```
pub fn translate_opcode(&mut self) {

    match self.opcode & 0xF000 {
        0x0000 => match self.opcode & 0x0FFF {

            // 00E0
            0x00E0 => cls(self),
            // 00EE
            0x00EE => ret(self),

            _ => panic!("Unknown opcode [0x0000]: {:X}", self.opcode)
        },
```

*Листинг 5 - функции обновления таймеров и эмуляции цикла:*

```
pub fn update_timers(&mut self, ui: &mut Ui) {

    // The reason for checking if == 10:
    // screen refresh rate - 600 Hz, timers refresh rate by docs should be 60 Hz,
    // so we refresh them every 10 cycles
    if self.timer_refresh_cd == 10 {
        if self.delay_timer > 0 {
            self.delay_timer -= 1;
        }
        if self.sound_timer > 0 {
            if self.sound_timer == 1 {
                ui.play_sound();
            }
            self.sound_timer -= 1;
        } else if self.sound_timer == 0 {
            ui.stop_sound();
        }
        self.timer_refresh_cd = 0;
    } else {
        self.timer_refresh_cd += 1;
    }
}

pub fn emulate_cycle(&mut self, ui: &mut Ui) {
    // fetch opcode: merge two memory locations for an opcode (build opcode with next
two bytes)
    self.opcode = (self.ram[self.pc as usize] as u16) << 8 | self.ram[self.pc as usize
+ 1] as u16;

    println!("Executing opcode 0x{:X}", self.opcode);

    self.translate_opcode();

    self.update_timers(ui);
}
```

Так как каждая ячейка памяти имеет размерность 1 байт, а стандартный размер каждого кода операции – 2 байта, необходимо совместить две ячейки памяти вместе. Для этого смещаем первую ячейку на 8 бит влево, а затем совершаем побитовое ИЛИ для добавления второй части кода операции в конец получившейся переменной.

Допустим, в первой ячейке лежало значение 0xA2, а во второй – 0xF0. При их совмещении у нас получается значение 0xA2F0. Теперь нам нужно декодировать этот код операции:

*Листинг 6 – опкод и его реализация*

*vm.rs*

```
// ANNN
0xA000 => ld_i_addr(self),
```

*opcodes.rs*

```
// ANNN =
// Set I to the address NNN.
pub fn ld_i_addr(vm: &mut Vm) {
    vm.i = vm.opcode & 0x0FFF;
    vm.pc += 2;
}
```

Так как нам необходимо записать NNN (младшие 12 бит) в I, воспользуемся побитовым И для того, чтобы избавиться от старших 4 бит, получая значение 0x2F0.

Так как размер каждой инструкции – 2 байт, необходимо увеличивать счетчик программы именно на 2, чтобы не случилось несостыковок при последующих расшифровках кодов операций. В других кодах операций, если нам необходимо пропустить следующую инструкцию, мы будем увеличивать счетчик программ на 4.

Таймеры: по документации обновление таймеров должно происходить с частотой в 60 Гц. Коды операций будут выполняться с частотой 600 Гц, поэтому введем дополнительную переменную *timer_refresh_cd*, позволяющую выполнять отсчет до следующего обновления таймеров.

После реализации цикла эмуляции встает задача инициализации системы.

```
const FONT: [u8; FONT_BYTES] = [
    0xF0, 0x90, 0x90, 0x90, 0xF0, // 0
    0x20, 0x60, 0x20, 0x20, 0x70, // 1
    0xF0, 0x10, 0xF0, 0x80, 0xF0, // 2
    0xF0, 0x10, 0xF0, 0x10, 0xF0, // 3
    0x90, 0x90, 0xF0, 0x10, 0x10, // 4
    0xF0, 0x80, 0xF0, 0x10, 0xF0, // 5
    0xF0, 0x80, 0xF0, 0x90, 0xF0, // 6
    0xF0, 0x10, 0x20, 0x40, 0x40, // 7
    0xF0, 0x90, 0xF0, 0x90, 0xF0, // 8
    0xF0, 0x90, 0xF0, 0x10, 0xF0, // 9
    0xF0, 0x90, 0xF0, 0x90, 0x90, // A
    0xE0, 0x90, 0xE0, 0x90, 0xE0, // B
    0xF0, 0x80, 0x80, 0x80, 0xF0, // C
    0xE0, 0x90, 0x90, 0x90, 0xE0, // D
    0xF0, 0x80, 0xF0, 0x80, 0xF0, // E
    0xF0, 0x80, 0xF0, 0x80, 0x80, // F
];
```

*Листинг 7 - Набор спрайтов, представляющих стандартные символы*

```
pub fn init() -> Vm {
    Vm {
        pc: PROGRAM_START,
        i: 0,
        opcode: 0,
        v: [0; DATA_REGISTERS_COUNT],
        ram: [0; RAM_SIZE],

        screen: [false; SCREEN_PIXELS],
        key_states: [false; KEYS_COUNT],

        stack: [0; STACK_SIZE],
        sp: 0,

        delay_timer: 0,
        sound_timer: 0,

        draw_flag: false,

        timer_refresh_cd: 10
    }
}

pub fn load_font(&mut self) {
    for i in 0..80 {
        self.ram[i] = FONT[i];
    }
}
```

*Листинг 8 - Инициализация системы*

Перед запуском первого цикла эмуляции необходимо проинициализировать все переменные и массивы для корректной работы. Логично предположить, что все изначальные состояния по умолчанию равны 0 или false (счетчик обновления равен 10 для предотвращения возможных ошибок при обновлении таймеров в первом же цикле).

Также при инициализации системы в память должен быть загружен набор шрифтов/спрайтов, представляющих стандартные символы (хоть у Chip-8 и нет чего-то наподобие BIOS, этот набор все равно должен присутствовать). Каждый символ имеет размер 5 байт, всего их 16, значит загрузка символов начинается с 0x00 и заканчивается в 0x50 (80 в десятичной).

После подгрузки шрифтов в память нам также необходимо загрузить исполняемую программу.

*Листинг 9 – загрузка исполняемого файла*

```
let mut buffer = Vec::new();
match file.read_to_end(&mut buffer) {
    Err(why) => panic!("couldn't read {}: {}", display, Error::description(&why)),
    Ok(_) => println!("{} contains:\n{} bytes", display, buffer.len()),
};

let buffer_size = buffer.len();

// Load the game into RAM
for i in 0..buffer_size {
    self.ram[i + 512] = buffer[i];
}
```

Загрузка начинается с элемента 0x200 (512), т.к. это память, предоставленная под выполняемые программы. После выполнения всех вышеперечисленных операций мы готовы выполнять первый цикл эмуляции.

*Примеры некоторых кодов операций*

**0x8XY4**: добавить значение регистра Vy к Vx, и установить флаг переноса.

*Листинг 10 – команда 0x8XY4*

```
// 8XY4 =
// Adds VY to VX. VF is set to 1 when there's a carry,
// and to 0 when there isn't.
pub fn add_vx_vy(vm: &mut Vm) {
    let x = ((vm.opcode & 0x0F00) >> 8) as usize;
    let vx = vm.v[x];
    let y = ((vm.opcode & 0x00F0) >> 4) as usize;
    let vy = vm.v[y];

    if vy > (0xFF - vx) {
        vm.v[0xF] = 1;
    } else {
        vm.v[0xF] = 0;
    }
    vm.v[x] = vx.wrapping_add(vy);
    vm.pc += 2;
}
```

Регистр VF установлен в 1, когда есть перенос, и в 0, если его нет. Поскольку регистр может хранить только значения от 0 до 255 (8-битное значение), это означает, что, если сумма VX и VY больше 255, она не может быть сохранена в регистре (или фактически снова

начинает отсчет с 0). Если сумма VX и VY больше 255, мы используем флаг переноса, чтобы система знала, что общая сумма обоих значений действительно больше 255.

**0xFX33**: записать BCD-представление значения регистра Vx по адресам, начинающимся с I

*Листинг 11 – команда 0xFX33*

```
// FX33 =
// Store binary-coded decimal representation of a value
// contained in VX to addr i, i+1, and i+2.
pub fn ld_b_vx(vm: &mut Vm) {
    let x = ((vm.opcode & 0x0F00) >> 8) as usize;
    let vx = vm.v[x];

    vm.ram[vm.i as usize] = vx / 100;
    vm.ram[(vm.i + 1) as usize] = (vx / 10) % 10;
    vm.ram[(vm.i + 2) as usize] = (vx % 100) % 10;
    vm.pc += 2;
}
```

Как можно увидеть, в ячейку по адресу I записывается значение, соответствующее сотням Vx, I+1 – десяткам, а I+2 – единицам.

**0xDXYN**: нарисовать спрайт по координатам (X, Y) высотой N. Спрайт берется по адресам, начинающимся с того, что указан в I.

*Листинг 12 – команда 0xDXYN*

```
// DXYN =
// Display n-byte sprite starting at memory location I at (Vx, Vy),
// Set VF = collision.
pub fn drw_vx_vy_n(vm: &mut Vm) {
    let x = ((vm.opcode & 0x0F00) >> 8) as usize;
    let vx = vm.v[x];
    let y = ((vm.opcode & 0x00F0) >> 4) as usize;
    let vy = vm.v[y];
    let height = vm.opcode & 0x000F;

    vm.v[0xF] = 0;
    for y_line in 0..height {

        // get byte
        let pixel = vm.ram[(vm.i + y_line) as usize];

        // for each pixel on this line
        for x_line in 0..8 {
            // check if the current pixel will be drawn by AND-ING it to 1 - IOW
            // check if the pixel is set to 1 (This will scan through the byte,
            // one bit at the time)

            if (pixel & (0x80 >> x_line)) != 0 {
                let current_position = ((vx as u16 + x_line as u16) + ((vy as u16 +
y_line) * 64)) % (32 * 64);

                // since the pixel will be drawn, check the destination location in
                // gfx for collision (verify if that location is flipped on (== 1))

                if vm.screen[current_position as usize]{
                    vm.v[0xF] = 1; // register the collision
                }
                vm.screen[current_position as usize] ^= true;
            }
        }
    }
```

```
    vm.draw_flag = true;
    vm.pc += 2;
}
```

Chip-8 рисует спрайты на экране. Код операции сообщает нам количество строк N, а также какие регистры V нам нужно проверить, чтобы получить координаты X и Y. Ширина каждого спрайта фиксирована (8 бит / 1 байт). Состояние каждого пикселя устанавливается с помощью побитовой операции XOR. Это означает, что он будет сравнивать текущее состояние пикселя с текущим значением в памяти. Если текущее значение отличается от значения в памяти, значение бита будет равно 1. Если оба значения совпадают, значение бита будет равно 0.

Предположим, значение опкода было равным 0xD003. Это означает, что необходимо нарисовать спрайт в позиции (0,0) высотой в 3 строки. Допустим, начиная с ячейки памяти I были установлены следующие значения:

```
memory[I]     =     0x3C; //00111100
memory[I + 1] =     0xC3; //11000011
memory[I + 2] =     0xFF; //11111111
```

Переводя это в двоичную систему счисления, получаем 00111100, 11000011 и 11111111 соответственно. Ориентируясь по ним, спрайт будет выглядеть так:

```
  ****
**    **
********
```

Исходя из полученных значений на экране вырисовываются пиксели в режиме XOR.

Для работы с вводом-выводом Chip-8 предоставляет 3 кода операций: EX9E, EXA1 и FX0A. Первый опкод пропускает следующую инструкцию, если определенная клавиша находится в «зажатом» состоянии, второй опкод делает то же самое, но клавиша должна быть «отжата», а третий ожидает нажатия любой клавиши и записывает в Vx ту, которая была нажата.

Рассмотрим принцип работы **0xEX9E**:

*Листинг 13 – команда 0xEX9E*

```
// EX9E =
// Skips the next instruction if the key stored in VX
// is pressed.
pub fn skp_vx(vm: &mut Vm) {
    let x = ((vm.opcode & 0x0F00) >> 8) as usize;
    let key = vm.v[x];

    if vm.key_states[key as usize] == true {
        vm.pc += 4;
    } else {
        vm.pc += 2;
    }
}
```

Здесь проверяется, была ли нажата клавиша, значение которой лежит в регистре Vx. Если да, следующая инструкция пропускается.

### *2.3. Модуль устройств ввода-вывода.*

Данный модуль был описан при помощи использования библиотеки SDL2.

SDL (Simple DirectMedia Layer) - это кроссплатформенная библиотека разработки, предназначенная для обеспечения низкоуровневого доступа к устройствам ввода-вывода (клавиатура, аудио, графические устройства) через OpenGL и Direct3D – API для рендеринга 2D- и 3D- графики.

Для начала нужно определить, с какими элементами придется работать. Очевидно, что должен быть какой-то объект, позволяющий обеспечить работу с графическими элементами. Так как у нас имеется таймер звука, необходимо, чтобы мы также могли передавать сигнал по аудиоканалу при возникновении каких-либо событий (например, ударение мяча о платформу).

*Листинг 14 – структура устройств ввода-вывода*

```
pub struct Ui {
    pub canvas: Canvas<Window>,
    device: AudioDevice<SquareWave>
}
```

Canvas<Window> позволяет манипулировать как оболочкой, так и внутренней частью окна; Вы можете манипулировать попиксельно, линиями, цветными прямоугольниками или вставлять текстуры (Textures) в данный Canvas.

AudioDevice<SquareWave> представляет собой устройство, позволяющее проигрывать сигнал типа «меандр» (по сути является писком).

*Листинг 15 – блок инициализации*

```
pub fn init(sdl_context: &Sdl, scale: u32) -> Ui {
    let video_subsystem = sdl_context.video().unwrap();
    let window = video_subsystem.window("chip-8", 64 * scale, 32 * scale)
        .position_centered()
        .opengl()
        .build()
        .unwrap();
    let mut canvas = window.into_canvas().build().unwrap();

    canvas.set_draw_color(Color::RGB(0, 0, 0));
    canvas.clear();
    canvas.present();


    let audio_subsystem = sdl_context.audio().unwrap();
    let desired_spec = AudioSpecDesired {
        freq: Some(44000),
        channels: Some(1),      // mono
        samples: None,          // default sample size
    };

    let device = audio_subsystem.open_playback(None, &desired_spec, |spec| {
        // Initialize the audio callback
        SquareWave {
            phase_inc: 440.0 / spec.freq as f32,
            phase: 0.0,
            volume: 0.25,
        }
    }).unwrap();

    Ui {
        canvas,
        device
    }
}
```

В функцию инициализации мы передаем контекст всего приложения для того, чтобы потом получить из него «подсистемы» графики и звука - VideoSubsystem и AudioSubsystem соответственно, а также масштаб изображения (в данном случае я выбрал масштаб 1024x768).

Из графической подсистемы мы инициализируем WindowBuilder при помощи метода window с заданием заголовка окна, а также его масштабами. Затем из получившегося Window, созданного при помощи build(), мы инициализируем объект Canvas, с помощью которого мы затем будем отображать различные изображения в окне.

Из звуковой подсистемы мы инициализируем AudioDevice со звуком SquareWave, предварительно задав ему указанные параметры с помощью объекта AudioSpecDesired – частота, количество каналов и буферное время (время, приходящееся на обработку поступающих сигналов). С помощью этого устройства мы сможем воспроизводить волну типа меандр или «квадратную волну».

*Листинг 16 – логика отрисовки пикселей и воспроизведения звуков*

```
/// Draws the CPU's display to the canvas
pub fn draw_canvas(&mut self, vm: &mut Vm, scale: u32) {
    for i in 0..64 * 32 {
        let current_pixel = vm.screen[i];
        let x = (i % 64) * scale as usize;
        let y = (i / 64) * scale as usize;

        self.canvas.set_draw_color(Color::RGB(0, 0, 0));
        if current_pixel {
            self.canvas.set_draw_color(Color::RGB(255, 255, 255));
        }
        let _ = self.canvas.fill_rect(Rect::new(x as i32, y as i32, scale, scale));
    }
    self.canvas.present();
}

/// Plays a square wave sound
pub fn play_sound(&mut self) {
    self.device.resume();
}

/// Stops the sound
pub fn stop_sound(&mut self) {
    self.device.pause();
}
```

Для воссоздания шестнадцатеричной клавиатуры, используемой в Chip-8, для удобства необходимо осуществить следующее отображение:

```
Virtual Keypad       Keyboard (QWERTY)
+-+-+-+-+                +-+-+-+-+
|1|2|3|C|                |1|2|3|4|
+-+-+-+-+                +-+-+-+-+
|4|5|6|D|                |Q|W|E|R|
+-+-+-+-+       =>       +-+-+-+-+
|7|8|9|E|                |A|S|D|F|
+-+-+-+-+                +-+-+-+-+
|A|0|B|F|                |Z|X|C|V|
+-+-+-+-+                +-+-+-+-+
```

*Рисунок 4 - предпочтительное отображение*

*Листинг 17 – отображение шестнадцатеричной клавиатуры*

```
pub fn set_key_pressed(&mut self, vm: &mut Vm, keycode: Keycode) {
    match keycode {
        Keycode::Num1 => { vm.key_states[0x1] = true; }
        Keycode::Num2 => { vm.key_states[0x2] = true; }
        Keycode::Num3 => { vm.key_states[0x3] = true; }
        Keycode::Q => { vm.key_states[0x4] = true; }
        Keycode::W => { vm.key_states[0x5] = true; }
        Keycode::E => { vm.key_states[0x6] = true; }
        Keycode::A => { vm.key_states[0x7] = true; }
        Keycode::S => { vm.key_states[0x8] = true; }
        Keycode::D => { vm.key_states[0x9] = true; }
        Keycode::Z => { vm.key_states[0xA] = true; }
        Keycode::X => { vm.key_states[0x0] = true; }
        Keycode::C => { vm.key_states[0xB] = true; }
        Keycode::Num4 => { vm.key_states[0xC] = true; }
        Keycode::R => { vm.key_states[0xD] = true; }
        Keycode::F => { vm.key_states[0xE] = true; }
        Keycode::V => { vm.key_states[0xF] = true; }
        _ => {}
    }
}
```

### *2.4. Главный блок программы*

Теперь, когда все модули описаны, можно составить главный блок программы, управляющий процессом эмулятора.

*Листинг 18 – главный блок*

```
const SCALE: u32 = 16;

fn main() {
    let sdl_context = sdl2::init().unwrap();
    let args: Vec<String> = env::args().collect();
    let mut vm = Vm::init();
    vm.load_font();

    vm.load_game(&args[1]);

    let mut ui = Ui::init(&sdl_context, SCALE);

    let mut event_pump = sdl_context.event_pump().unwrap();
    'running: loop {
        for event in event_pump.poll_iter() {
            match event {
                Event::Quit { .. } | Event::KeyDown { keycode: Some(Keycode::Escape),
.. } => {
                    break 'running;
                }
                Event::KeyDown { keycode: Some(keycode), .. } =>
ui.set_key_pressed(&mut vm, keycode),
                Event::KeyUp { keycode: Some(keycode), .. } =>
ui.set_key_released(&mut vm, keycode),
                _ => {}
            }
        }
        ::std::thread::sleep(Duration::new(0, 1_000_000_000u32 / 600));
        vm.emulate_cycle(&mut ui);
        if vm.draw_flag {
            ui.draw_canvas(&mut vm, SCALE);
        }
    }
}
```

Сначала загружаем модули внутренней памяти и процессора, также загружая в него исполняемую программу (в данном случае это игра), путь к которой мы передаем через аргумент командной строки.

Затем происходит инициализация event_pump (объект EventPump), который следит за различными событиями, происходящими в контексте приложения; в данном случае выход или нажатие/отжатие клавиш.

thread::sleep() служит для задания частоты выполнения опкодов, здесь частота – 600 Гц. После сна вызывается метод цикла эмуляции, а если выставлен флаг рисования, то также и метод вырисовки пикселей.

## 3. Примеры работы

Рабочую версию эмулятора/интерпретатора можно найти и загрузить, клонировав репозиторий по ссылке https://github.com/nailstorms/chip-8.

Запуск происходит через командную строку. Для этого необходимо перейти в директорию с эмулятором, а затем прописать одну из приведенных комбинаций команд:

*cargo run [путь к бинарному файлу]*

или

*cargo build --release*
*cd /target/release*
*./chip-8 [путь к бинарному файлу]*

Отличие второй комбинации от первой в том, что она собирает эмулятор для повторного использования, и в будущем запуск будет производиться именно из директории /release, при этом не будет уходить дополнительного времени на сборку/компиляцию кода эмулятора.

```
C:\Users\nailstorm\Desktop\itmo niversity\github\projects>cd chip-8
C:\Users\nailstorm\Desktop\itmo niversity\github\projects\chip-8>cargo run games\PONG
```

*Рисунок 5 - пример команды запуска эмулятора*

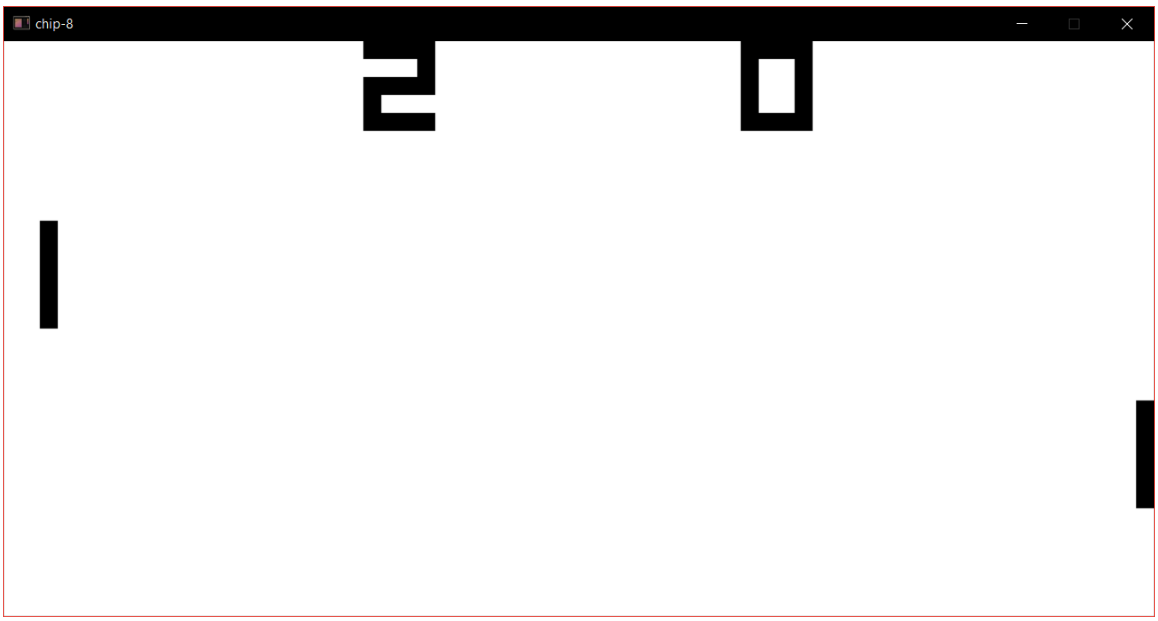


*Рисунок 6 - игра Pong*

## Итоговые результаты

В ходе прохождения практики была исследована архитектура интерпретируемого языка программирования Chip-8, а также были составлены конспект и описание архитектуры в приведенной преподавателем нотации. Был написан эмулятор/интерпретатор на основе языка программирования Rust, позволяющий воспроизводить разного вида программы (преимущественно игры) 20-30 летней давности на современных платформах.

Мною была изучена различная документация Chip-8, предоставленная как и создателем данного языка, так и различными другими независимыми разработчиками; были приобретены весьма полезные навыки анализа разнообразных видов документаций и статей, описания физических архитектур, а также разработки приложений с использованием возможностей языка Rust и его комплектующих.

## Список использованной литературы


1. **CHIP-8 – Wikipedia** [https://en.wikipedia.org/wiki/CHIP-8]
2. **Cowgod's Chip-8 Technical Reference** [http://devernay.free.fr/hacks/chip8/C8TECH10.HTM]

В вышеописанных источниках приведена теоретическая информация о Chip-8, его памяти, регистрах, кодах операций и командах. Конспект был составлен с использованием данной информации.

3. **Korenkov I. et al. DECLARATIVE TARGET ARCHITECTURE DEFINITION FOR DATA-DRIVEN DEVELOPMENT TOOLCHAIN //International Multidisciplinary Scientific GeoConference: SGEM: Surveying Geology & mining Ecology Management. – 2018. – Т. 18. – С. 271-278.**

Статья, содержащая информацию о синтаксисе описания архитектур.

4. **mattmik : Mastering CHIP-8** [http://mattmik.com/files/chip8/mastering/chip8.html]
5. **How to write an emulator (CHIP-8 interpreter)** [http://www.multigesture.net/articles/how-to-write-an-emulator-chip-8-interpreter/]

Информация, помогающая реализовать основные аспекты Chip-8 при создании эмулятора.

6. **The Rust Programming Language** [https://doc.rust-lang.org/book/index.html]
7. **Cargo** [https://doc.rust-lang.org/cargo/]
8. **SDL2** [https://rust-sdl2.github.io/rust-sdl2/sdl2/]

Документация языка программирования Rust, являющегося основным языком программирования в процессе реализации эмулятора, а также его системы управления пакетами Cargo и библиотеки SDL2, с помощью которой была описана работа устройств ввода-вывода.