

## СОДЕРЖАНИЕ

<b>ОПРЕДЕЛЕНИЯ И СОКРАЩЕНИЯ</b>	<b>2</b>
<b>ВВЕДЕНИЕ</b>	<b>3</b>
<b>1 Исследование аппаратных и программных средств</b>	<b>6</b>
1.1 Обзор области применения	6
1.2 Обзор архитектуры RISC-V	7
1.3 Обзор существующих эмуляторов RISC-V	9
1.3.1 Spike	9
1.3.2 QEMU	9
1.3.3 riscv-rust	10
1.4 Обзор существующих IDE	11
1.4.1 Visual Studio Code	11
1.4.2 Eclipse	11
1.4.3 IntelliJ IDEA	12
1.5 Выводы	12
<b>2 Проектирование программного обеспечения</b>	<b>15</b>
2.1 Требования к программному обеспечению	15
2.2 Используемые программные средства	16
2.3 Общий алгоритм работы	17
2.4 Функциональные компоненты	17
2.4.1 Эмулятор	18
2.4.2 Транслятор "ИК-ПП" и транслятор "ПП-ИК"	19
2.4.3 Ассемблер и дизассемблер	23
2.4.4 Отладчик	23
2.4.5 Статический анализатор	24
2.4.6 Расширение IDE и пользовательский интерфейс	24
<b>3 Использование программного обеспечения</b>	<b>26</b>
3.1 Загрузка и установка IDE Visual Studio Code	26
3.2 Загрузка расширения	27
3.3 Запуск среды разработки	27
3.4 Использование расширения	28
<b>ЗАКЛЮЧЕНИЕ</b>	<b>31</b>
<b>СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ</b>	<b>32</b>

## ОПРЕДЕЛЕНИЯ И СОКРАЩЕНИЯ

*Безопасность через неясность* - принцип, используемый для обеспечения безопасности, основная идея которого заключается в том, чтобы скрыть внутреннее устройство системы

*Двоичная трансляция* - эмуляция одного набора инструкций на другом за счёт трансляции машинного кода из первого набора инструкций во второй

*Микроархитектура* - способ реализации архитектуры набора команд в процессоре

*ПЛИС* - программируемая логическая интегральная схема

*Разрешительная (пермиссивная) лицензия* - лицензия на программное обеспечение, которая практически не ограничивает свободу действий пользователей ПО и разработчиков, работающих с исходным кодом

*gdb* - GNU Debugger

*IDE* - integrated development environment (интегрированная среда разработки)

*IEEE 754-2008* - широко используемый стандарт, описывающий формат представления чисел с плавающей точкой

*ISA* - instruction set architecture (архитектура набора команд)

*RISC* - reduced instruction set computer (компьютер с сокращенным набором команд)

*RISC-V* - открытый стандарт архитектуры набора команд процессора

*RISC-V International* - некоммерческая организация, координирующая и поддерживающая разработку архитектуры RISC-V

*WASM* - WebAssembly

*WebAssembly* - язык программирования низкого уровня для стековой виртуальной машины, спроектированный как цель компиляции для высокоуровневых языков, таких как Си, C++, Rust

*JSON (JavaScript Object Notation)* - текстовый формат обмена данными, основанный на JavaScript

## ВВЕДЕНИЕ

Любая новая технология сталкивается с дилеммой курицы и яйца, и RISC-V - не исключение. Относительно молодой открытый проект, опирающийся преимущественно на добровольцев и энтузиастов, закономерно сталкивается с проблемой принятия сообществами пользователей и разработчиков: для массового распространения технологии требуется большая база участников, но для большой базы участников требуется массовое распространение технологии. Текущее положение усугубляется предметной областью проекта: сфера разработки аппаратного обеспечения всё ещё является довольно закрытой, с небольшим количеством любителей и крупными компаниями, контролирующими рынок с помощью проприетарных технологий и препятствующими появлению новых конкурентов.

Актуальность темы работы заключается в необходимости продвижения концепции открытых систем в сфере аппаратного обеспечения и потребности в современной учебной среде разработки для низкоуровневого программирования под процессоры RISC-V. RISC-V - открытая и свободная система команд и процессорная архитектура на основе концепции RISC (reduced instruction set computer - компьютер с сокращенным набором команд) для микропроцессоров и микроконтроллеров.[\[1\]](#) Спецификация доступна для свободного и бесплатного использования, включая коммерческие реализации непосредственно в кремнии или конфигурировании ПЛИС (программируемой логической интегральной схемы). Имеет встроенные возможности для расширения списка команд и подходит для широкого круга применений, включая изучение организации и функционирования вычислительных машин и обучение языкам ассемблера. Данная архитектура является одним из самых перспективных кандидатов для фундамента вычислительной техники будущего, ведь в мире, где информационные технологии всё глубже проникают в повседневную жизнь, безопасность через неясность (принцип, используемый для обеспечения безопасности, основная идея которого заключается в том, чтобы скрыть внутреннее устройство

системы) и чрезмерная привязка к поставщикам просто недопустимы. Надёжные информационные системы должны строиться на открытых технологиях и существовать в среде конструктивной конкуренции, где происходит свободный обмен знаниями и опытом, а не полагаться на защищённые патентами и коммерческими тайнами решения.

Несмотря на все преграды, стоящие перед молодой архитектурой, популярность RISC-V растёт. Координирует и поддерживает разработку некоммерческая организация RISC-V International, а среди многочисленных участников этой организации есть и всемирно известные компании, например, Google, Intel, IBM, Nvidia.<sup>[2]</sup> Также в списке участников находятся отечественные компании, например, Syntacore, разработчик микропроцессорных технологий и инструментов на базе RISC-V. Силами всего сообщества разрабатываются как непосредственно аппаратные реализации архитектуры, так и большое количество программного обеспечения, включая эмуляторы и IDE (integrated development environment - интегрированная среда разработки). На последних двух и будет акцент, потому что многочисленные существующие решения или чрезмерно ориентированы на использование в производстве и не удовлетворяют критерию простоты для обучения, или слишком упрощены и далеки от производственного опыта. Таким образом, данная работа - это попытка создания простого, но приближенного к производству программного обеспечения для обучения низкоуровневому программированию под архитектуру RISC-V. Чем эффективнее и доступнее процесс обучения, тем ниже порог входа, и тем больше начинающих, любителей и специалистов придёт в сферу открытого аппаратного обеспечения.

Целью данной выпускной квалификационной работы является проектирование и разработка эмулятора RISC-V и учебной среды разработки для языка ассемблера. Для достижения заданной цели необходимо последовательно выполнить следующие ключевые задачи:

1. изучить существующие эмуляторы RISC-V и IDE,

2. выполнить проектирование собственных решений,
3. выполнить кодирование,
4. произвести тестирование,
5. составить руководство пользователя,
6. обеспечить возможность получения ПО пользователем.

Краткие итоги касательно выполнения задач и достижения цели будут сформулированы в заключении.

## **1 Исследование аппаратных и программных средств**

В данном разделе будут рассмотрены относящиеся к теме работы аппаратные и программные средства, а также сформулированы конкретные задачи для достижения поставленной цели.

### **1.1 Обзор области применения**

При самостоятельном изучении новых технологий или обучении других зачастую возникает вопрос качества обучения: насколько эффективны используемые инструменты и что можно в них изменить, чтобы быстрее достичь определённого уровня знаний и навыков? Изначально программирование было крайне "бюрократизировано", а вычислительные ресурсы ограничены, дороги и не доступны повсеместно, из-за чего программисту того времени приходилось проходить через:

1. изучение набора команд и особенностей работы конкретной ЭВМ,
2. написание программы без полноценной среды разработки,
3. ожидание в очереди для возможности доступа к ЭВМ,
4. запуск и отладку программы,
5. коррекцию программы и возвращение к пункту 3.

Следовательно, в данных условиях обучение новых программистов требовало ощутимых временных и материальных затрат. Однако, благодаря научно-техническому прогрессу вычислительные машины стали повсеместно доступны и в тысячи раз производительнее, что изменило подход к обучению: теперь акцент делается на самообучение и быструю обратную связь. Инструменты и технологии, используемые при обучении, также должны учитывать тенденции текущего времени.

Можно предположить, что близость учебной среды к производственной повышает эффективность обучения. Это действительно так, потому что полученный при обучении опыт легко применяется уже в реальной деятельности. Однако, производственная среда может предполагать наличие определённых компетенций, что может задавать повышенный порог входа,

поэтому, для его снижения потребуется где-то пойти на определённые упрощения, а где-то добавить больше сведений для устранения неясностей. Также учебная среда должна быть легкодоступной и предполагать быструю обратную связь для выявления ошибок.

## **1.2 Обзор архитектуры RISC-V**

RISC-V - это новая ISA (instruction set architecture - архитектура набора команд), изначально предназначенная для обучения и исследований в области архитектуры вычислительных машин, но которая также стремится стать стандартной бесплатной и открытой архитектурой для промышленного применения.[\[3\]](#) Ниже перечислены некоторые характеристики RISC-V:

- полностью открытая ISA, бесплатно доступная для научных кругов и промышленности;
- подходит для прямой аппаратной реализации, а не только для симуляции или двоичной трансляции (эмуляция одного набора инструкций на другом за счёт трансляции машинного кода из первого набора инструкций во второй);
- отсутствие излишней привязки к определённому стилю микроархитектуры (способа, которым данная ISA реализована в процессоре) или технологии реализации;
- самостоятельная небольшая базовая целочисленная система команд, которую можно использовать отдельно в качестве основы для разработки аппаратных ускорителей или для образовательных целей, а также дополнительные стандартные расширения для поддержки разработки программного обеспечения общего назначения;
- поддержка стандарта IEEE 754-2008 (широко используемый стандарт, описывающий формат представления чисел с плавающей точкой);
- поддержка как возможностей для расширения, так и возможностей для специализации;

- поддержка как 32-битных, так и 64-битных вариантов адресного пространства;
- поддержка многоядерных и многопроцессорных реализаций;
- опциональные инструкции переменной длины.

Архитектура набора команд RISC-V избегает, насколько это возможно, деталей реализации, и её следует рассматривать как программно-видимый интерфейс для широкого спектра реализаций, а не как дизайн конкретного оборудования.

RISC-V ISA состоит из базовой целочисленной ISA, которая должна присутствовать в любой реализации, и необязательных расширений. Базовая целочисленная ISA очень похожа на ранние процессоры RISC, за исключением отсутствия слотов задержки ветвления и поддержки опциональных кодировок инструкций переменной длины. База ограничена минимальным набором инструкций, достаточным для того, чтобы обеспечить разумную платформу для компиляторов, ассемблеров, компоновщиков и операционных систем (с дополнительными привилегированными операциями), и, таким образом, обеспечивает удобный "скелет" для построения процессора. Каждый базовый набор целочисленных команд характеризуется шириной и количеством целочисленных регистров, а также размером адресного пространства. Два основных набора команд, RV32I и RV64I, обеспечивают 32-битное или 64-битное адресное пространство соответственно. Также существует набор команд RV32E, подмножество RV32I, который был добавлен для поддержки небольших микроконтроллеров и имеет 16 целочисленных регистров, а не 32. В будущем готовым к использованию станет RV128I, поддерживающий 128-битное адресное пространство. В комплект стандартных расширений входят наборы команд для обеспечения поддержки целочисленного умножения/деления, атомарных операций и арифметики с плавающей запятой одинарной и двойной точности.



### 1.3 Обзор существующих эмуляторов RISC-V

Поскольку RISC-V - это открытый проект с модульной структурой, то для него существует большое количество эмуляторов различного качества и направленности. Ознакомимся с несколькими наиболее актуальными решениями и обозначим их слабые и сильные стороны, что необходимо при разработке собственного эмулятора.

#### 1.3.1 Spike

Spike - эталонный функциональный программный симулятор RISC-V ISA.[4] Spike реализует большое количество стандартных расширений, позволяет добавлять собственные инструкции, поддерживает интерактивную отладку и отладку с помощью gdb (GNU Debugger). Исходный код открыт и находится под разрешительной лицензией (лицензией на программное обеспечение, которая практически не ограничивает свободу действий пользователей ПО и разработчиков, работающих с исходным кодом).[5] К заметным недостаткам можно отнести следующие особенности разработки и сопровождения:

- ограниченная переносимость,
- усложнённый процесс сборки из-за неполной инструкции,
- устаревший набор инструментов разработки.

Стоит также отметить, что Spike предназначен в первую очередь для разработчиков аппаратного обеспечения, а в данной работе поставлена задача разработки эмулятора, целевой аудиторией которого преимущественно являются программисты.

#### 1.3.2 QEMU

QEMU - это универсальный эмулятор и виртуализатор различных процессорных архитектур с открытым исходным кодом. QEMU можно использовать несколькими способами. Наиболее распространена эмуляция всей системы, когда создаётся виртуальную модель всей вычислительной

машины (центрального процессора, памяти и прочих устройств) для запуска гостевой ОС. В этом режиме процессор может быть или эмулирован, или работать с гипервизором, таким как KVM или Xen, чтобы позволить гостевой системе работать непосредственно на процессоре хоста. Вторым поддерживаемым способом использования QEMU - эмуляция пользовательского режима. В данном случае QEMU может запускать процессы, скомпилированные для одной процессорной архитектуры, на другой процессорной архитектуре. В этом режиме процессор всегда эмулируется.[6]

К недостаткам можно отнести:

- сложность использования,
- естественно следующую из поддержки множества архитектур избыточность.

Хотя QEMU и является одним из самых популярных средств эмуляции и виртуализации, его нельзя считать полностью подходящим для использования в учебных целях без предварительной настройки.

### 1.3.3 riscv-rust

riscv-rust - это эмулятор процессора RISC-V и периферийных устройств, написанный на Rust и скомпилированный в WebAssembly (язык программирования низкого уровня для стековой виртуальной машины, спроектированный как цель компиляции для высокоуровневых языков, таких как Си, C++, Rust). Может быть легко встроен в проект Rust или JavaScript. Среди других особенностей стоит отметить:

- работу Linux и xv6,
- запуск в браузере,
- встроенный отладчик.

К недостаткам можно отнести неспешный темп разработки проекта и его акцент на запуск операционных систем, что может быть излишним в рамках данной работы.[7]

## 1.4 Обзор существующих IDE

Поскольку рынок IDE заполнен отличными продуктами от профессиональных команд с многолетним опытом, разработка собственной среды разработки является неоправданной. Лучше сфокусироваться на основной цели данной работы и использовать уже готовое решение. Осуществлять выбор стоит из IDE общего назначения, в которых хорошо проработаны как основные функции (например, редактирование текста, управление проектами), так и возможности для расширения. В таком случае, будет относительно просто предоставить интеграцию требуемых инструментов в виде единого расширения для уже готовой среды разработки. Ознакомимся с несколькими актуальными решениями.

### 1.4.1 Visual Studio Code

Visual Studio Code - это легкий, но мощный редактор исходного кода, доступный для ОС Windows, macOS и Linux. Он поставляется со встроенной поддержкой JavaScript, TypeScript, Node.js, и имеет богатую экосистему расширений для других языков (таких как C++, C#, Java, Python, PHP, Go) и сред выполнения (таких как .NET и Unity).[8] Visual Studio Code построен на платформе Electron, которая используется для разработки веб-приложений Node.js, использующих браузерный движок Blink. Функционал IDE можно дополнить с помощью расширений, доступных в центральной репозитории. Примечательной особенностью является возможность создавать расширения, которые добавляют поддержку новых языков, отладчиков и статических анализаторов с использованием Language Server Protocol.[9]

### 1.4.2 Eclipse

Eclipse - это интегрированная среда разработки, содержащая базовое рабочее пространство и систему подключаемых модулей для настройки среды. Eclipse написана в основном на Java и основным поддерживаемым языком является Java, но её также можно использовать для разработки

приложений на других языках программирования с помощью подключаемых модулей.[\[10\]](#)

### **1.4.3 IntelliJ IDEA**

IntelliJ IDEA - это интегрированная среда разработки, написанная на Java. Она разработана компанией JetBrains (ранее известной как IntelliJ) и доступна в виде версии Community Edition под лицензией Apache и в виде проприетарной коммерческой версии. Они не имеют каких-либо ограничений для коммерческого использования.

IntelliJ IDEA предоставляет определенные функции, такие как завершение кода путем анализа контекста, навигацию по коду, которая позволяет напрямую переходить к классу или объявлению в коде, рефакторинг кода, отладку кода. Среда разработки обеспечивает интеграцию с инструментами сборки/упаковки, такими как grunt, bower, gradle и SBT. Она поддерживает системы контроля версий, такие как Git, Mercurial, Perforce и SVN. IntelliJ поддерживает плагины, с помощью которых можно добавить дополнительные функции в IDE. Плагины можно загрузить и установить либо с веб-сайта репозитория плагинов IntelliJ, либо с помощью встроенной в IDE функции поиска и установки плагинов.[\[11\]](#)

## **1.5 Выводы**

Завершая обзор аппаратных и программных средств, составим сравнительные таблицы изученных решений: в таблице 1 произведено сравнение эмуляторов RISC-V, а в таблице 2 сравниваются интегрированные среды разработки. При итоговом выборе готового решения в рамках данной работы ключевыми качествами являются (тем не менее стоит также учитывать прочие характеристики):

- поддержка веб-платформы (одна из самых переносимых, полноценных и безопасных сред исполнения);

- активность команды разработчиков (скорость исправления ошибок и развития продукта, качество документации и технической поддержки);
- стоимость (влияет на доступность решения);
- лицензия (разрешительные и свободные лицензии позволяют избежать привязки к поставщику и упрощают использование решения).

Если среди рассмотренных решений не найдено подходящего, то разработку своего собственного решения можно считать оправданной.

Среди эмуляторов наиболее подходящим решением кажется `riscv-rust`, из особенностей которого стоит выделить простоту интеграции и поддержку веб-платформы. К сожалению, он обладает неполной документацией, что может значительно затруднить внедрение данного решения в нестандартный проект. Таким образом, разработка собственного эмулятора RISC-V может считаться целесообразной в рамках данной работы.

Таблица 1 - Сравнение эмуляторов RISC-V

	<b>Spike</b>	<b>QEMU</b>	<b>riscv-rust</b>
<b>Направленность</b>	АО	ПО	ПО
<b>Интеграция</b>	Сложно	Сложно	Легко*
<b>Веб-платформа</b>	Нет	Нет	Да
<b>Поддержка</b>	Плохо	Хорошо	Плохо
<b>Стоимость</b>	Бесплатно	Бесплатно	Бесплатно
<b>Лицензия</b>	BSD	GPL v2	MIT

Среди интегрированных сред разработки выделяется Visual Studio Code. Это одна из самых популярных и универсальных IDE, что может упростить процесс работы для уже знакомых с ней пользователей. Недавно была выпущена практически полноценная веб-версия с возможностью работы в режиме офлайн и поддержкой расширений, написанных с использованием специального API.[\[12\]](#) Таким образом, разработка расширения для Visual Studio Code также может считаться рациональным выбором в рамках данной работы.

Таблица 2 - Сравнение интегрированных сред разработки

	<b>VSCode</b>	<b>Eclipse</b>	<b>IDEA</b>
<b>Расширяемость</b>	Хорошо	Хорошо	Хорошо
<b>Веб-платформа</b>	Частично	Частично	Нет
<b>Поддержка</b>	Хорошо	Хорошо	Хорошо
<b>Стоимость</b>	Бесплатно	Бесплатно	Бесплатно (CE)
<b>Лицензия</b>	MIT (OC)	EPL	Apache (CE)

Таким образом, наиболее перспективным решением является разработка собственного расширения для VSCode. Само расширение будет состоять из следующих тесно интегрированных модулей:

1. эмулятор,
2. ассемблер,
3. дизассемблер,
4. отладчик,
5. статический анализатор,
6. транслятор "ИК-ПП",
7. транслятор "ПП-ИК"

Для достижения цели работы необходимо выполнить проектирование, кодирование и тестирование указанных модулей, после чего произвести их интеграцию в единое расширение для VSCode и обеспечить возможность использования итогового ПО пользователем.

## **2 Проектирование программного обеспечения**

Проектирование - важный этап в разработке комплексного программного обеспечения. Хорошо спроектированное программное обеспечение требует меньше временных и материальных затрат на этапах кодирования, тестирования и внедрения. В последующих подразделах будут сформулированы требования к ПО, определён набор используемых инструментов, построен общий алгоритм работы и описаны функциональные компоненты.

### **2.1 Требования к программному обеспечению**

Опираясь на результаты анализа предметной области и существующих решений (см. раздел 1), можно выделить следующие требования:

1. простота получения и использования;
2. способ организации работы, предоставляемый учебным программным обеспечением, должен быть приближен к производственному;
3. возможность быстрой обратной связи для выявления ошибок.

Для удовлетворения требований были приняты следующие решения:

1. использовать разрешительную лицензию и не требовать выплат за использование, а также использовать веб-платформу в качестве среды исполнения;
2. итоговое ПО должно стать расширением для одной из самых популярных сред разработки - Visual Studio Code;
3. ПО должно включать компонент для статического анализа в реальном времени.

Соблюдение всех указанных требований позволит создать программное обеспечение, способное конкурировать с существующими решениями.

## 2.2 Используемые программные средства

Выбор подходящих инструментов и технологий может значительно сократить затраты на разработку программного обеспечения. В данной работе будут использованы:

- браузер из семейства Chromium (открытая, безопасная, доступная и современная платформа для исполнения веб-приложений, простота доставки которых позволит пользователю практически мгновенно получить среду разработки без необходимости установки каких-либо сторонних программ);
- веб-версия IDE Visual Studio Code (доступная в браузере практически полноценная среда разработки, одной из ключевых особенностей которой является продвинутая расширяемость функционала с помощью пользовательских дополнений);
- модули WebAssembly (наборы из стандартизированных программных интерфейсов и переносимых низкоуровневых бинарных кодов, исполняемых соответствующей виртуальной машиной как в среде браузера (например, V8 в Chromium)[13], так и вне (например, V8 в Node.js));
- язык программирования JavaScript (один из самых популярных и доступных языков программирования с автоматическим управлением памятью, а также основной язык для написания логики веб-приложений);
- язык программирования Rust (системный язык программирования, разработанный для повышения производительности и безопасности, поддерживающий WebAssembly как одну из основных целей для компиляции)[14].

Перечисленные выше программные средства будут учитываться при дальнейшем проектировании ПО.



## 2.3 Общий алгоритм работы

Общий алгоритм работы итогового ПО для большего понимания следует изобразить через подобие диаграммы вариантов использования (см. рисунок 1), где для упрощения показаны только основные действия пользователя с системой.

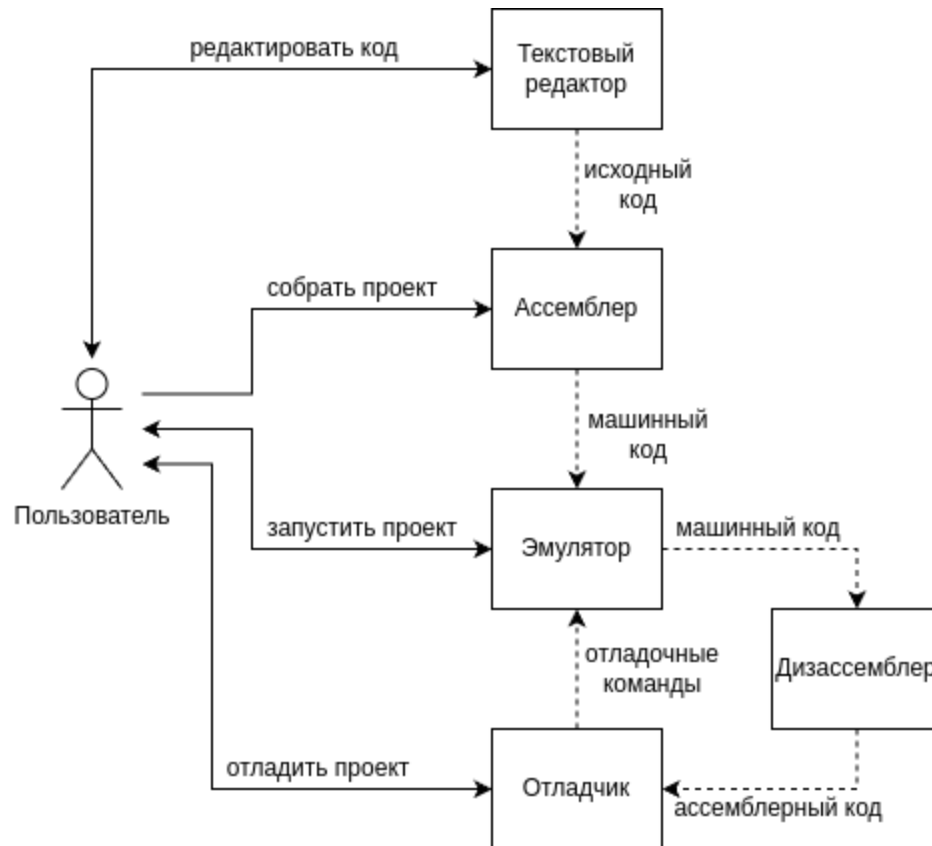


Рисунок 1 - Диаграмма основных вариантов использования ПО

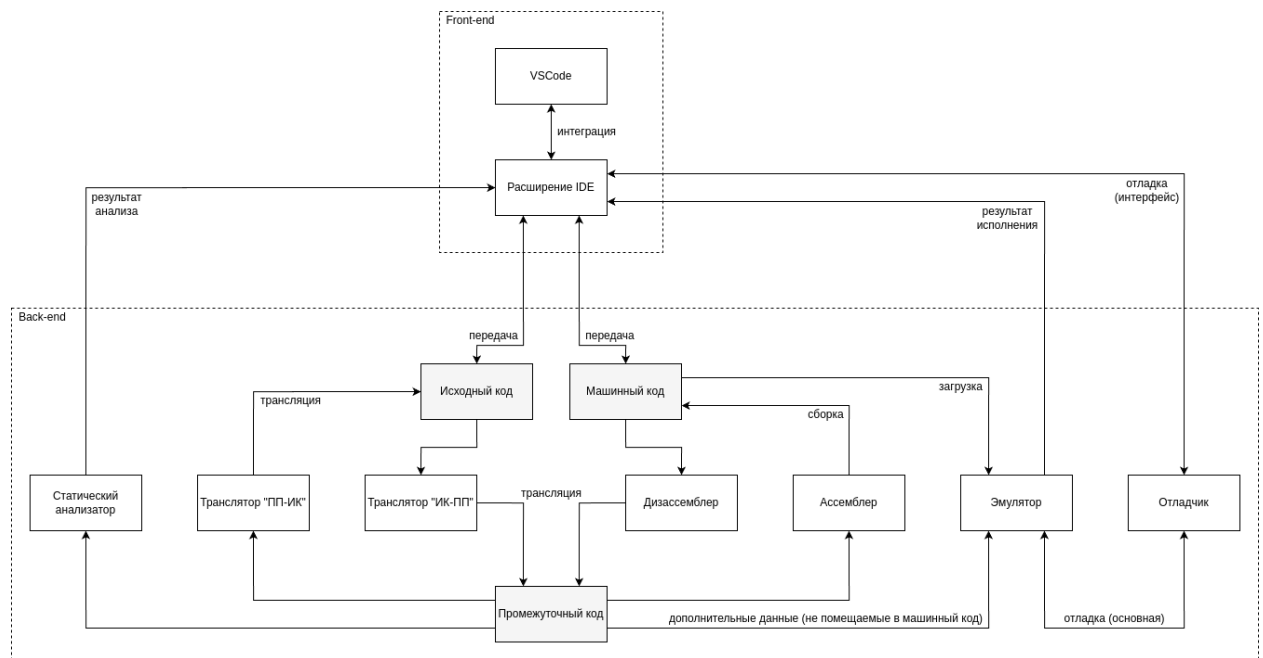
Приведённая выше диаграмма станет основой для дальнейших работ по проектированию компонентов разрабатываемого программного обеспечения.

## 2.4 Функциональные компоненты

Итоговое программное обеспечение должно состоять из следующих функциональных компонентов:

1. эмулятор,
2. транслятор “ИК-ПП” (Исходный Код - Промежуточное Представление),
3. транслятор “ПП-ИК” (Промежуточное Представление - Исходный Код),
4. ассемблер,

Общая схема взаимодействия компонентов изображена на рисунке 2.



## Рисунок 2 - Взаимодействие компонентов

В последующих подразделах будет приведено более подробное описание каждого функционального компонента.

### 2.4.1 Эмулятор

Эмулятор будет реализовывать базовый набор непривилегированных инструкций (версия спецификации: 20191213)[3] RV32I в одноядерном режиме. На рисунке 3 показан общий алгоритм работы эмулятора, опирающийся на классический цикл “fetch-decode-execute”.

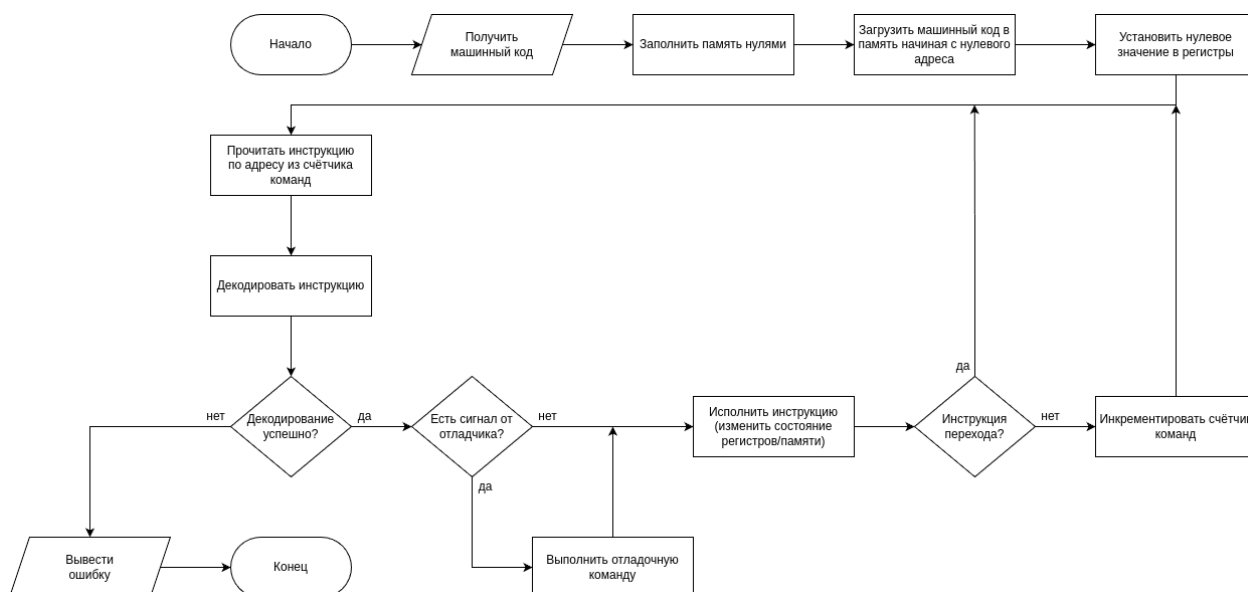


Рисунок 3 - Алгоритм работы эмулятора

Получившийся эмулятор будет простым, но достаточным программным решением в рамках данной работы.

#### 2.4.2 Транслятор "ИК-ПП" и транслятор "ПП-ИК"

Транслятор - это общий термин, который может относиться ко всему, что преобразует код с одного компьютерного языка на другой.[\[15\]](#) Промежуточное представление - это структура данных или код, используемый внутри компилятора или виртуальной машины для представления исходного кода.[\[16\]](#) Транслятор из исходного кода в промежуточное представление нужен для упрощения межкомпонентного взаимодействия.

Для получения промежуточного кода необходимо последовательно выполнить сканирование, разбор и перевод исходного кода.

Начнём с первого шага: сканирование (лексический анализ) - процесс выделения токенов (фиксированных групп символов) из линейного потока символов. На листинге 1 показан шаблон (грамматика), с помощью которого поток символов будет разбиваться на токены. Для записи грамматики используется дополненная форма Бэкуса-Наура (ABNF).

## Листинг 1 - Лексическая грамматика

```

name      = ALPHA 1*(ALPHA / DIGIT) (comma / space / eol)
comma     = " ,"
space     = SP
eol       = LF

number    = [minus] 1*DIGIT eol
minus     = "- "

comment   = semicolon *(space / VCHAR) eol
semicolon = "; "

```

Получившаяся лексическая грамматика является регулярной, поэтому для распознавания можно построить конечный автомат (см. рисунок 4).

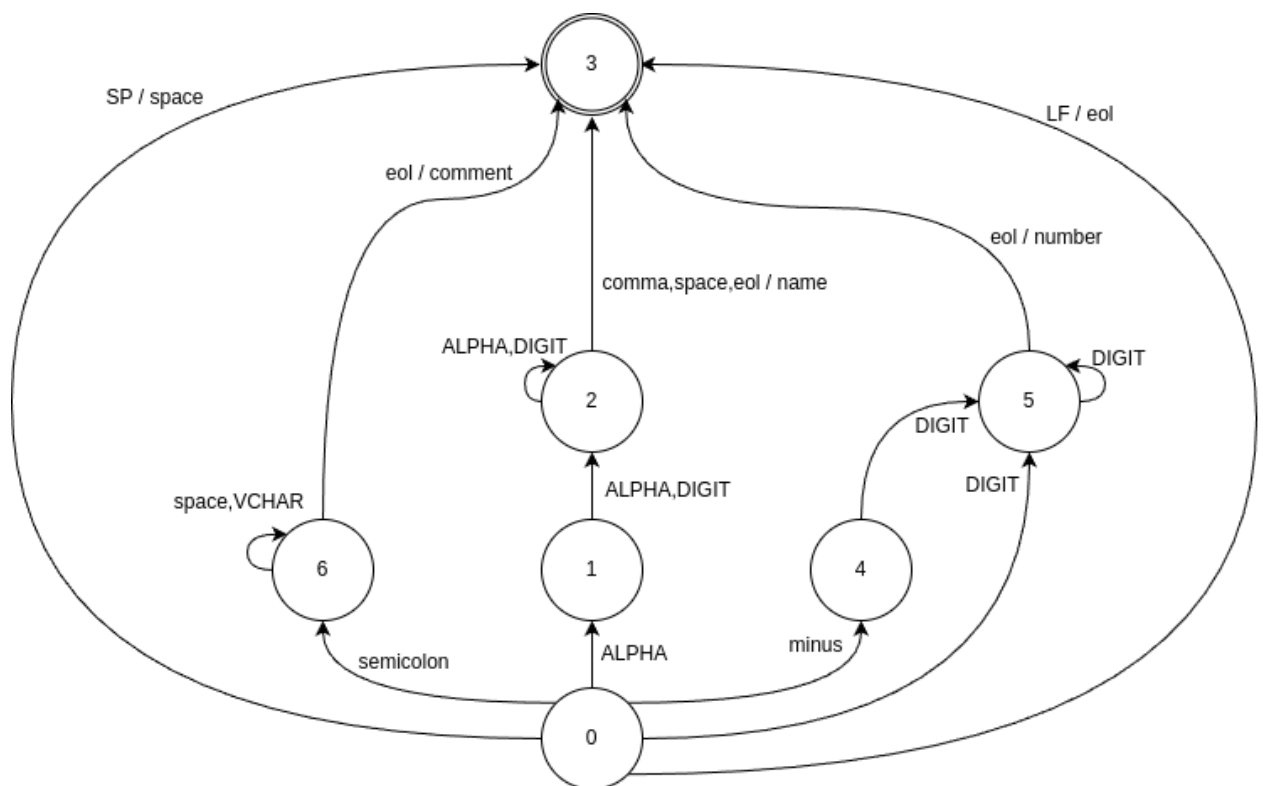


Рисунок 4 - Конечный автомат, распознающий лексическую грамматику  
Получившийся конечный автомат будет использован на этапе кодирования.

После лексического разбора будет подготовлена последовательность токенов в формате “НАЗВАНИЕ:ЗНАЧЕНИЕ”, где “НАЗВАНИЕ” - именование распознанного правила грамматики, а “ЗНАЧЕНИЕ” - поле, используемое для правил NAME и NUMBER.

Второй шаг - разбор. Разбор (синтаксический анализ) - это процесс построения абстрактного синтаксического дерева из последовательности

токенов в соответствии с определёнными правилами грамматики.[\[17\]](#) Однако, поскольку разрабатываемый в рамках данной выпускной квалификационной работы язык ассемблера является крайне упрощённым, его грамматика также является регулярной, а не контекстно-свободной (см. листинг 2). Соответственно, для разбора можно построить ещё один конечный автомат, а не прибегать к использованию алгоритмов синтаксического анализа (например, методу рекурсивного спуска). Конечным результатом разбора также станет линейная, а не древовидная структура данных.

#### Листинг 2 - Синтаксическая грамматика

```
; Для анализа принимается последовательность токенов.
; Допустимыми токенами являются:
; name, number, comment, space, eof
; Токены (name) и (number) содержат значения, используемые
; для анализа. Форма записи для получения значений:
; name:"value"
; number:_
; Для (name) это означает попытку получения из текущего
; токена терминала "value", которая может быть успешна только
; в том случае, если сохранённое значение совпадает с
; запрошенным.
; Для (number) это означает подстановку в правило грамматики
; любого числового терминала, сохранённого в токене.
; Дополнительное правило:
; REG - "x0" ... "x31"

r      = name:REG
i      = number:_

lui    = name:"lui"      r i
auipc  = name:"auipc"    r i
jal    = name:"jal"      r i
jalr   = name:"jalr"     r r i
beq    = name:"beq"      r r i
bne    = name:"bne"      r r i
blt    = name:"blt"      r r i
bge    = name:"bge"      r r i
bltu   = name:"bltu"     r r i
bgeu   = name:"bgeu"     r r i
lb     = name:"lb"       r r i
lh     = name:"lh"       r r i
lw     = name:"lw"       r r i
lbu    = name:"lbu"      r r i
lhu    = name:"lhu"      r r i
sb     = name:"sb"       r r i
sh     = name:"sh"       r r i
```

## Продолжение листинга 2

```
sw      = name:"sw"      r r i
addi    = name:"addi"    r r i
slti    = name:"slti"    r r i
sltiu   = name:"sltiu"   r r i
xori    = name:"xori"    r r i
ori     = name:"ori"     r r i
andi    = name:"andi"    r r i
slli    = name:"slli"    r r i
srli    = name:"srli"    r r i
srai    = name:"srai"    r r i
add     = name:"add"     r r r
sub     = name:"sub"     r r r
sll     = name:"sll"     r r r
slt     = name:"slt"     r r r
sltu    = name:"sltu"    r r r
xor     = name:"xor"     r r r
srl     = name:"srl"     r r r
sra     = name:"sra"     r r r
or      = name:"or"      r r r
and     = name:"and"     r r r
fence   = name:"fence"   r r i
ecall   = name:"ecall"
ebreak  = name:"ebreak"
```

Таким образом, после выполнения синтаксического анализа будет получен массив структур данных формата “ОПЕРАЦИЯ: [ОПЕРАНД 1, ОПЕРАНД 2, ОПЕРАНД 3]”. Для промежуточного представления будет использоваться формат JSON (JavaScript Object Notation - текстовый формат обмена данными, основанный на JavaScript) (см. листинг 3).

## Листинг 3 - Пример промежуточного представления

```
{
  "0": [
    "addi",
    "x1",
    "x1",
    "0"
  ],
  "1": [
    "jal",
    "x15",
    "8"
  ]
}
```

Относительно простой процесс лексического и синтаксического анализа возможен только потому, что в данной работе используется крайне примитивный и приближённый к машинному коду язык ассемблера. Директивы и другие “высокоуровневые” возможности макроассемблеров не поддерживаются.

Обратная трансляция из промежуточного представления в исходный код выполняется через последовательное прохождение по ключам и значениям JSON-объекта с последующим копированием и выдачей их в выходной текстовый поток с дополнительными символами-терминаторами (пробелы, запятые, переносы строки).

### **2.4.3 Ассемблер и дизассемблер**

Поскольку в проекте используется промежуточное представление кода, то задачи ассемблирования и дизассемблирования не вызывают больших трудностей. Для генерации машинного кода ассемблер должен пройти по ключам/значениям JSON-объекта (используемого для промежуточного представления) и упаковать операции с операндами в требуемые бинарные структуры-инструкции, описание которых дано в спецификации RISC-V.[\[3\]](#) Для дизассемблирования необходимо выполнить обратный процесс: выделить операции и операнды из бинарных структур и зафиксировать их в промежуточном представлении.

### **2.4.4 Отладчик**

В рамках данной работы отладчик будет представлять собой программный интерфейс для отладочных функций, встроенных в сам эмулятор. Такими функциями являются:

- просмотр точек останова,
- редактирование точек останова,
- управление исполнением в точках останова,
- просмотр содержимого памяти,

- редактирование содержимого памяти,
- просмотр содержимого регистров,
- редактирование содержимого регистров.

Это типичный функционал, часто встречающийся в других отладчиках.

#### **2.4.5 Статический анализатор**

Статический анализ кода - анализ программного обеспечения, производимый (в отличие от динамического анализа) без реального выполнения исследуемых программ.[\[18\]](#) В этом проекте статический анализатор будет давать подсказки о некорректных значениях числовых литералов в коде. Алгоритм прост: извлечь из промежуточного представления операции и операнды, определить тип инструкции для операции, выявить операнды-литералы и сопоставить их с допустимым диапазоном значений.

#### **2.4.6 Расширение IDE и пользовательский интерфейс**

Перечисленные выше функциональные модули являются самостоятельными компонентами, точкой интеграции которых выступает расширение для интегрированной среды разработки. В расширении будут определены внутренние коммуникации между модулями, а также построен пользовательский интерфейс в виде дополнительных графических элементов, которые установят правила ввода-вывода данных в системе “пользователь-программа”. Пример графического интерфейса можно увидеть на рисунке 5.



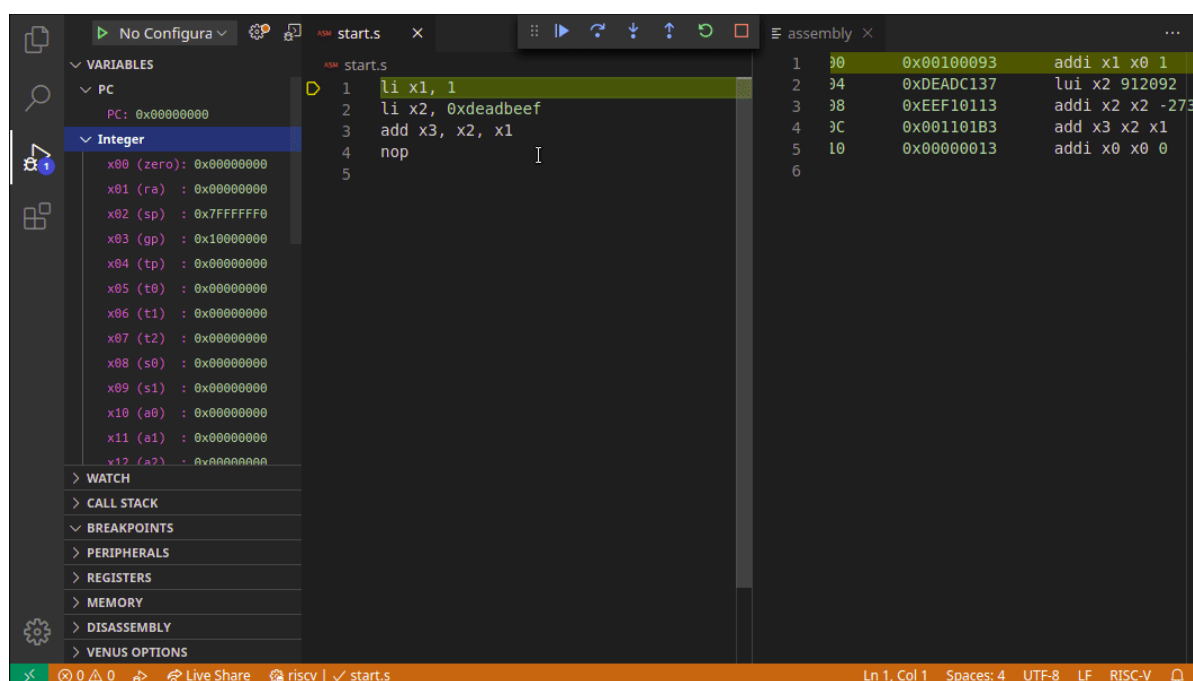


Рисунок 5 - Пример интерфейса, создаваемого расширением

В качестве примера взят интерфейс среды разработки VSCode с расширением “RISC-V Venus Simulator”.[19] Приёмы проектирования и организации графического интерфейса, используемые данным расширением, пригодятся при разработке собственного решения.

### 3 Использование программного обеспечения

Для того, чтобы воспользоваться разработанным программным обеспечением, необходимо выполнить следующие действия:

1. загрузить и установить IDE Visual Studio Code,
2. загрузить расширение,
3. запустить IDE Visual Studio Code с определёнными параметрами для использования расширения.

Последующие разделы можно рассматривать как руководство пользователя и пример использования разработанного программного обеспечения.

#### 3.1 Загрузка и установка IDE Visual Studio Code

Получить последнюю версию среды разработки можно на официальном сайте (см. рисунок 6).[\[20\]](#)

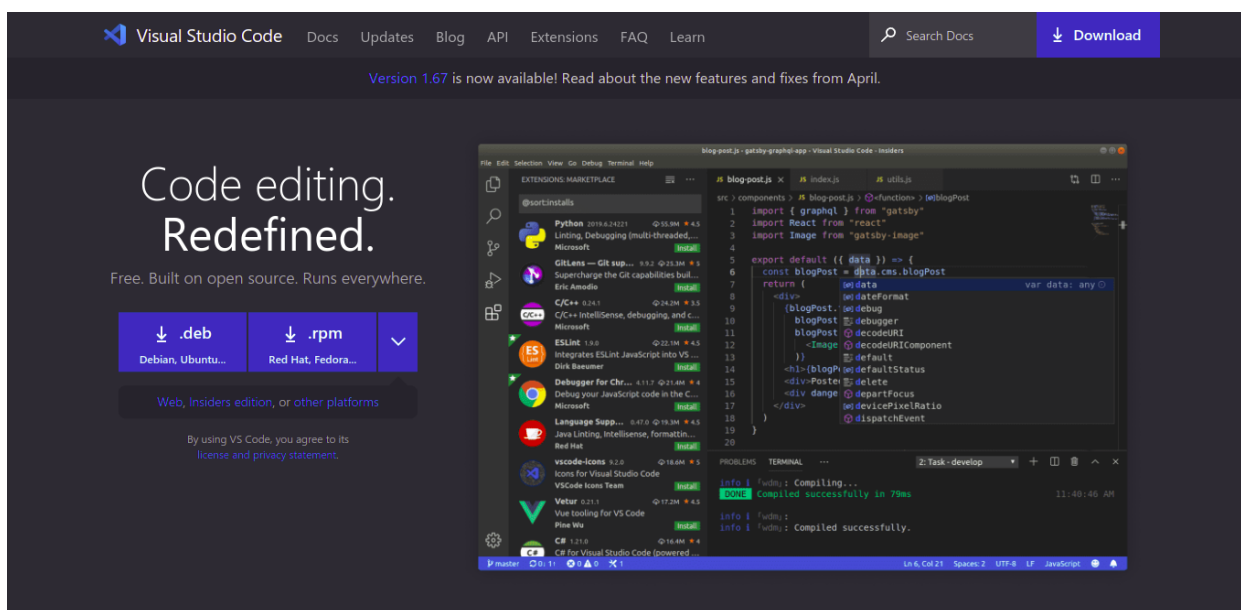
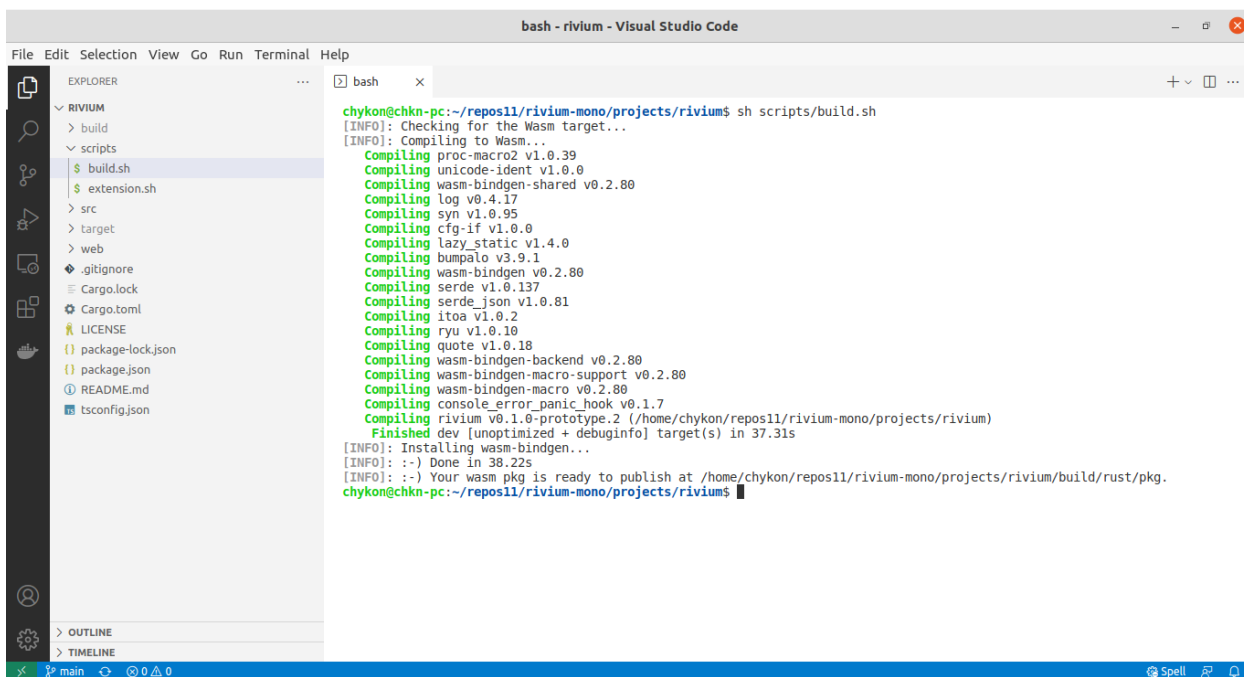


Рисунок 6 - Главная страница официального сайта VSCode

Пакеты установки доступны для всех основных платформ. Процесс установки варьируется в зависимости от операционной системы, поэтому пользователю может понадобиться ознакомиться с соответствующими руководствами.

## 3.2 Загрузка расширения

Расширение вместе с исходным кодом можно получить из Git-репозитория проекта, расположенного на сервисе GitHub.[21] Для получения копии репозитория можно использовать сам git, или же создать ZIP-архив с помощью сайта (Code > Download ZIP). Далее пользователю необходимо настроить командную оболочку, указав директорию “project/rivium” (вычисляется относительно корня репозитория) как текущую. Затем потребуется сгенерировать WASM-модуль и JS-обёртку. Для UNIX-систем это можно сделать с помощью сценария командной оболочки, запуск которого осуществляется командой “sh scripts/build/sh”. Пример запуска показан на рисунке 7.



```
bash - rivium - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
RIVIUIM
  build
  scripts
    $ build.sh
    $ extension.sh
  src
  target
  web
.gitignore
Cargo.lock
Cargo.toml
LICENSE
package-lock.json
package.json
README.md
tsconfig.json
OUTLINE
TIMELINE
main

chyon@chkn-pc:~/repos11/rivium-mono/projects/rivium$ sh scripts/build.sh
[INFO]: Checking for the Wasm target...
[INFO]: Compiling to Wasm...
Compiling proc-macro2 v1.0.39
Compiling unicode-ident v1.0.0
Compiling wasmbindgen-shared v0.2.80
Compiling log v0.4.17
Compiling syn v1.0.95
Compiling cfg-if v1.0.0
Compiling lazy_static v1.4.0
Compiling bumpalo v3.9.1
Compiling wasmbindgen v0.2.80
Compiling serde v1.0.137
Compiling serde_json v1.0.81
Compiling itoa v1.0.2
Compiling ryu v1.0.10
Compiling quote v1.0.18
Compiling wasmbindgen-backend v0.2.80
Compiling wasmbindgen-macro-support v0.2.80
Compiling wasmbindgen-macro v0.2.80
Compiling console_error_panic_hook v0.1.7
Compiling rivium v0.1.0-prototype.2 (/home/chyon/repos11/rivium-mono/projects/rivium)
Finished dev [unoptimized + debuginfo] target(s) in 37.31s
[INFO]: Installing wasmbindgen...
[INFO]: --) Done in 38.22s
[INFO]: --) Your wasm pkg is ready to publish at /home/chyon/repos11/rivium-mono/projects/rivium/build/rust/pkg.
chyon@chkn-pc:~/repos11/rivium-mono/projects/rivium$
```

Рисунок 7 - Результат выполнения сценария для генерации WASM-модуля  
Как показано на рисунке выше, можно использовать саму среду VSCode.

## 3.3 Запуск среды разработки

Запуск среды разработки должен осуществляться с помощью отдельного сценария командной оболочки. Для этого необходимо ввести в терминал команду “sh scripts/extension.sh”. После исполнения сценария будет

открыт новый экземпляр среды разработки, способный взаимодействовать с разработанным расширением.

### 3.4 Использование расширения

Для начала пользователю необходимо создать файл и ввести требуемый исходный код. Пример такого файла можно увидеть на рисунке 8.

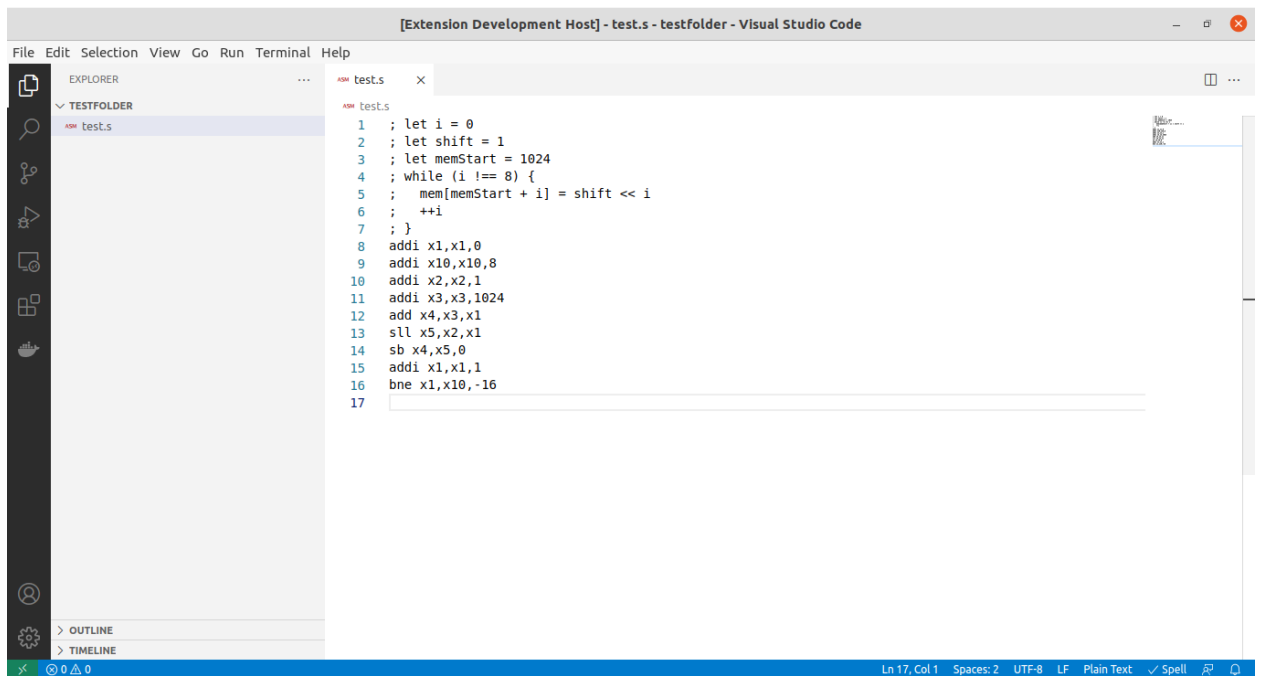


Рисунок 8 - Пример файла с исходный кодом

Затем необходимо создать новый экземпляр виртуальной машины, выбрав пункты “View > Command Palette...” и введя команду “Rivium: Init” (см. рисунок 9).

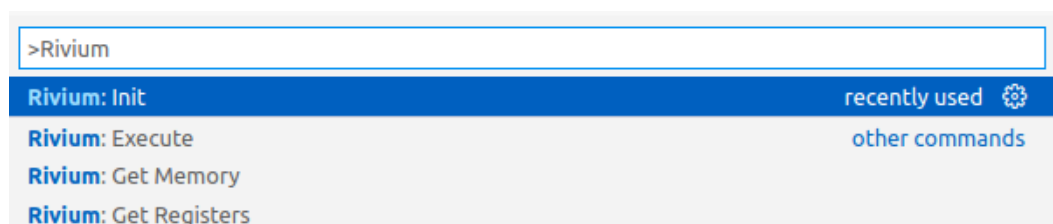


Рисунок 9 - Панель ввода команд

Результат исполнения можно увидеть, выбрав пункты “View > Output” и установив канал “rivium” (см. рисунок 10).



Рисунок 10 - Вывод канала “rivium” после исполнения команды “init”

Для исполнения кода необходимо ввести команду “Rivium: Execute”. Исходный код будет преобразован в машинный и загружен в память эмулятора, начиная с нулевого адреса. Результат выполнения показан на рисунке 11. Исполнение будет продолжаться до тех пор, пока не возникнет ошибка декодирования.

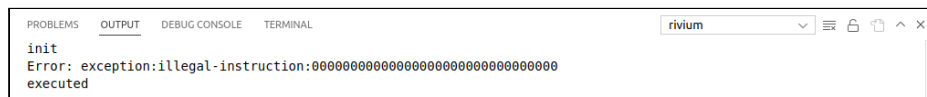


Рисунок 11 - Результат выполнения команды “execute”

Для просмотра содержимого регистров и памяти есть команды “Rivium: Get Registers” и “Rivium: Get Memory” соответственно. Результаты их выполнения можно увидеть на рисунках 12 и 13.

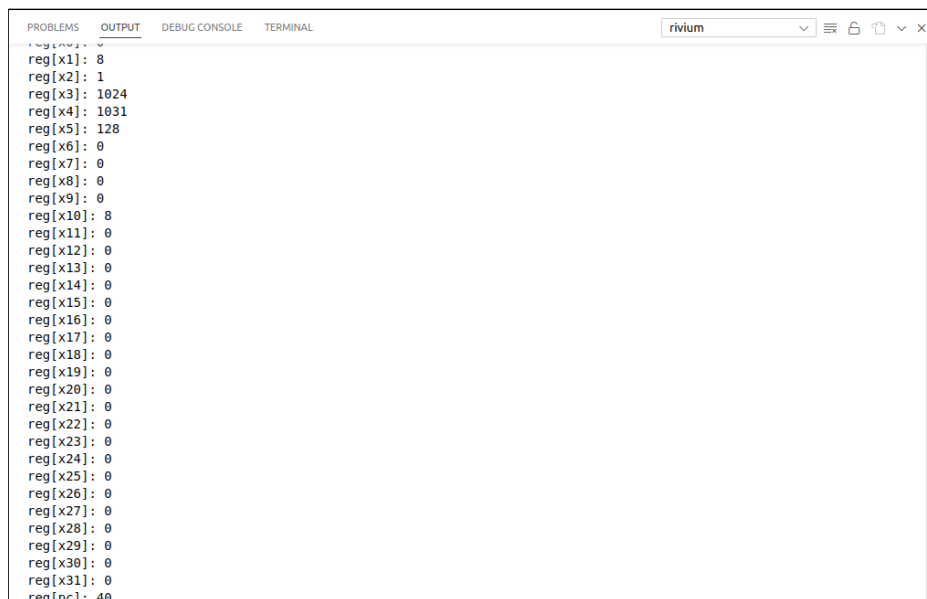


Рисунок 12 - Просмотр содержимого регистров



Рисунок 13 - Просмотр содержимого памяти

Изучив состояние регистров и памяти пользователь может сделать вывод о корректности и работоспособности алгоритма программы.

## **ЗАКЛЮЧЕНИЕ**

RISC-V, как и любая новая технология, столкнулся с проблемой популяризации среди разработчиков. Одним из способов её преодоления можно считать разработку доступных, качественных и приближенных к производственной среде учебных инструментов, помогающих быстро и без особых затрат подготавливать новых инженеров в сфере программного и аппаратного обеспечения. Международное сообщество профессионалов и любителей, сформировавшееся вокруг проекта RISC-V, понимает это и активно вкладывает ресурсы в разработку соответствующих программных и аппаратных решений, включая эмуляторы и интегрированные среды разработки.

Цель выпускной квалификационной работы, а именно проектирование и разработку эмулятора RISC-V и учебной среды разработки для языка ассемблера, можно считать достигнутой. В ходе работы были последовательно выполнены поставленные задачи, результатами которых являются:

1. описание существующих эмуляторов RISC-V и интегрированных сред разработки;
2. описание как общей архитектуры разрабатываемого приложения, так и каждого модуля в отдельности;
3. исходный код и набор тестов, подтверждающий работоспособность приложения;
4. доступное для загрузки с общедоступного Интернет-ресурса приложение;
5. инструкция по использованию.

Подробнее с ними можно ознакомиться в соответствующих разделах.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. RISC-V // Википедия. [2022]. Дата обновления: 11.02.2022. URL: <https://ru.wikipedia.org/?curid=5198947&oldid=119962967> (дата обращения: 11.02.2022).
2. Members // RISC-V International. [2022]. Дата обновления: 20.02.2022. URL: <https://riscv.org/members/> (дата обращения: 20.02.2022).
3. Waterman A., Asanovic K. The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA // Document Version. - 2019. - Т. 20191213.
4. The RISC-V ISA Simulator (Spike) // Chipyard main documentation. [2019]. Дата обновления: 28.09.2019. URL: <https://chipyard.readthedocs.io/en/latest/Software/Spike.html> (дата обращения: 12.03.2022).
5. Spike, a RISC-V ISA Simulator // GitHub. [2022]. Дата обновления: 12.03.2022. URL: <https://github.com/riscv-software-src/riscv-isa-sim> (дата обращения: 12.03.2022).
6. About QEMU // QEMU documentation. [2021]. Дата обновления: 07.09.2021. URL: <https://www.qemu.org/docs/master/about/index.html> (дата обращения: 12.03.2022).
7. RISC-V processor emulator written in Rust+WASM // GitHub. [2021]. Дата обновления: 22.08.2021. URL: <https://github.com/takahirox/riscv-rust> (дата обращения: 14.03.2022).
8. Documentation for Visual Studio Code // Visual Studio Code. [2022]. Дата обновления: 23.03.2022. URL: <https://code.visualstudio.com/docs> (дата обращения: 23.03.2022).
9. Visual Studio Code // Wikipedia. [2022]. Дата обновления: 22.02.2022. URL: [https://en.wikipedia.org/wiki/Visual\\_Studio\\_Code](https://en.wikipedia.org/wiki/Visual_Studio_Code) (дата обращения: 23.03.2022).
10. Eclipse (software) // Wikipedia. [2022]. Дата обновления: 30.01.2022. URL: [https://en.wikipedia.org/wiki/Eclipse\\_\(software\)](https://en.wikipedia.org/wiki/Eclipse_(software)) (дата обращения: 23.03.2022).



- 11.IntelliJ IDEA // JetBrains. [2022]. Дата обновления: 23.03.2022. URL: <https://www.jetbrains.com/idea/> (дата обращения: 23.03.2022).
- 12.VSCode.dev // VSCode. [2021]. Дата обновления: 20.10.2021. URL: <https://code.visualstudio.com/blogs/2021/10/20/vscode-dev> (дата обращения: 24.03.2022).
- 13.V8 JavaScript engine // V8 JavaScript engine. [2022]. Дата обновления: 20.04.2022. URL: <https://v8.dev/> (дата обращения: 20.05.2022).
- 14.Rust (programming language) // Wikipedia. [2022]. Д. о.: 17.05.2022. URL: [https://en.wikipedia.org/wiki/Rust\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Rust_(programming_language)) (дата обращения: 20.05.2022).
- 15.Translator (computing) // Wikipedia. [2022]. Д. о.: 29.03.2022. URL: [https://en.wikipedia.org/wiki/Translator\\_\(computing\)](https://en.wikipedia.org/wiki/Translator_(computing)) (дата обращения: 22.05.2022).
- 16.Intermediate representation // Wikipedia. [2022]. Д. о.: 16.05.2022. URL: [https://en.wikipedia.org/wiki/Intermediate\\_representation](https://en.wikipedia.org/wiki/Intermediate_representation) (дата обращения: 22.05.2022).
- 17.Nystrom R. Crafting Interpreters. - Genever Benning, 2021.
- 18.Static program analysis // Wikipedia. [2022]. Дата обновления: 06.05.2022. URL: [https://en.wikipedia.org/wiki/Static\\_program\\_analysis](https://en.wikipedia.org/wiki/Static_program_analysis) (дата обращения: 23.05.2022).
- 19.RISC-V Venus Simulator // Visual Studio Marketplace. [2022]. Дата обновления: 28.02.2022. URL: <https://marketplace.visualstudio.com/items?itemName=hm.riscv-venus> (дата обращения: 23.05.2022).
- 20.Visual Studio Code // Visual Studio Code. [2022]. Дата обновления: 05.06.2022. URL: <https://code.visualstudio.com/> (дата обращения: 05.06.2022).
- 21.chykon/rivium-mono // GitHub. [2022]. Дата обновления: 05.06.2022. URL: <https://github.com/chykon/rivium-mono> (дата обращения: 05.06.2022).