**Курс**

**«Основы программирования на языке Assembler под архитектуру RISC-V»**

****

# Отказ от ответственности

Данный материал является переводом и адаптацией учебного курса «Foundations of RISC-V Assembly Programming» под авторством Маттиаса Кёнига (Matthias Koenig), расположенного по адресу: https://training.linuxfoundation.org/training/foundations-of-risc-v-assembly-programming-lfd117x/ и распространяемого под лицензией CC BY 4.0. Перевод и адаптация выполнены Дмитрием Булахом, НИУ МИЭТ по заказу Альянса RISC-V, допускается к использованию под лицензией CC BY 4.0.

Оглавление

[Отказ от ответственности 2](#_Toc155820016)

[Добро пожаловать в курс! 6](#_Toc155820017)

[Общие сведения 6](#_Toc155820018)

[Введение в курс 6](#_Toc155820019)

[Цели обучения на курсе 6](#_Toc155820020)

[Прежде, чем мы начнём 6](#_Toc155820021)

[Необходимые знания 7](#_Toc155820022)

[Структура курса 7](#_Toc155820023)

[Глава 1. Общая информация о языке Assembler 8](#_Toc155820024)

[Введение 8](#_Toc155820025)

[О чём эта глава 8](#_Toc155820026)

[Цели главы 8](#_Toc155820027)

[Общая информация о языке Assembler 8](#_Toc155820028)

[RISC-V 8](#_Toc155820029)

[Язык Assembler 9](#_Toc155820030)

[Краткие сведения о микропроцессоре 10](#_Toc155820031)

[Классический RISC-конвейер 10](#_Toc155820032)

[Глава 2. Инструменты разработчика 12](#_Toc155820033)

[Введение 12](#_Toc155820034)

[Обзор главы 12](#_Toc155820035)

[Цели обучения 12](#_Toc155820036)

[Набор инструментов разработчика 12](#_Toc155820037)

[Обзор наборов инструментов 12](#_Toc155820038)

[Симулятор Ripes 13](#_Toc155820039)

[Эмулятор Qemu и набор инструментов GNU-toolchain для процессора RISC-V 14](#_Toc155820040)

[Пример использования набора инструментов 15](#_Toc155820041)

[Запуск Visual Studio Code и PlatformIO вместе с Qemu и пакетом инструментов RISC-V GNU Toolchain 17](#_Toc155820042)

[Пример работы в Visual Studio Code с расширением PlatformIO 18](#_Toc155820043)

[Мини-компьютер StarFive VisionFive и набор инструментов RISC-V GNU Toolchain 19](#_Toc155820044)

[Глава 3. Набор непривилегированных инструкций RISC-V 21](#_Toc155820045)

[Введение 21](#_Toc155820046)

[Обзор главы 21](#_Toc155820047)

[Цели обучения 21](#_Toc155820048)

[Набор непривилегированных инструкций RISC-V 21](#_Toc155820049)

[Основной набор целочисленных инструкций 21](#_Toc155820050)

[Расширения и именования 21](#_Toc155820051)

[Регистры RISC-V 22](#_Toc155820052)

[Обзор инструкций 24](#_Toc155820053)

[Простой пример 24](#_Toc155820054)

[Кодирование инструкций 25](#_Toc155820055)

[Арифметические и логические операции (с непосредственными значениями) 26](#_Toc155820056)

[Арифметические и логические операции (с регистрами) 28](#_Toc155820057)

[Загрузка и сохранение 29](#_Toc155820058)

[Управление 31](#_Toc155820059)

[Системный интерфейс 33](#_Toc155820060)

[Упорядочение памяти 34](#_Toc155820061)

[‘M’-расширение команд 35](#_Toc155820062)

[Замечания касательно архитектуры RV64 36](#_Toc155820063)

[Глава 4. Программирование на языке Assembler 37](#_Toc155820064)

[Введение 37](#_Toc155820065)

[Обзор главы 37](#_Toc155820066)

[Цели обучения 37](#_Toc155820067)

[Метки в языке Assembler 37](#_Toc155820068)

[Адресация с использованием меток 38](#_Toc155820069)

[Директивы языка Assembler 39](#_Toc155820070)

[Пример на директивы языка Assemler 41](#_Toc155820071)

[Псевдо-инструкции 43](#_Toc155820072)

[Псевдо-инструкции для загрузки, сохранения и дополнения 44](#_Toc155820073)

[Псевдо-инструкции для условной установки битов 46](#_Toc155820074)

[Псевдо-инструкции для условного ветвления 46](#_Toc155820075)

[Псевдо-инструкции для безусловного перехода 47](#_Toc155820076)

[Бинарный интерфейс для приложений (ABI) и пользовательский режим 48](#_Toc155820077)

[Стек 50](#_Toc155820078)

[Реализация контроля за ходом выполнения программ 50](#_Toc155820079)

[Условия 50](#_Toc155820080)

[Циклы 51](#_Toc155820081)

[Функции 52](#_Toc155820082)

[Рекурсия 52](#_Toc155820083)

[Глава 5. Использование системных библиотек 54](#_Toc155820084)

[Введение 54](#_Toc155820085)

[Обзор главы 54](#_Toc155820086)

[Цели обучения 54](#_Toc155820087)

[Использование системы и библиотек 54](#_Toc155820088)

[Взаимодействие с системой 54](#_Toc155820089)

[Взаимодействие с библиотеками 57](#_Toc155820090)

[Подсчёт символов 59](#_Toc155820091)

[Проверка чисел на простоту 61](#_Toc155820092)

[Глава 6. Примеры 64](#_Toc155820093)

[Введение 64](#_Toc155820094)

[Обзор главы 64](#_Toc155820095)

[Цели обучения 64](#_Toc155820096)

[Примеры 64](#_Toc155820097)

[Связный список 64](#_Toc155820098)

[Подсчёт среднего значения 67](#_Toc155820099)

# [Добро](https://learning.edx.org/course/course-v1:LinuxFoundationX+LFD117x+3T2022/block-v1:LinuxFoundationX+LFD117x+3T2022+type@sequential+block@73471eb62dcf4aa59e55add87fdf9a0f) пожаловать в курс!

## [Общие](https://learning.edx.org/course/course-v1:LinuxFoundationX+LFD117x+3T2022/block-v1:LinuxFoundationX+LFD117x+3T2022+type@sequential+block@73471eb62dcf4aa59e55add87fdf9a0f) сведения

### Введение в курс

Этот курс предназначен для базового понимания применения языка программирования Assembler при работе с микропроцессорами архитектуры RISC-V. Понимание и способность писать программы на языке Assembler очень важны для программистов, работающих с «железом» на низком уровне. Основы языка Assembler помогают в понимании и реализации частей операционных систем и компиляторов. Задачи вроде отладки и обнаружения критичных с точки зрения производительности участков кода проще решать, обладая базовыми знаниями языка Assembler.

Содержимое курса покрывает такие темы, как инструкции процессора RISC-V (RV32I, RV64I), псевдо-инструкции и директивы языка Assembler, интерфейс для написания программ (application programming interface, API), взаимодействие с программами, написанными на языке программирования C и понимание принципов дизассемблирования бинарного кода. Также рассматриваются идея и основные моменты, связанные с расширениями команд.

Основной задачей курса является предоставление информации, необходимой для выполнения поставленных задач, связанных с программированием, и получение практики программирования на языке Assembler на основе предоставляемых примеров. Они затрагивают такие вопросы, как циклы, условные операторы, функции и другие.

Показываются примеры запуска разрабатываемых программ на базе ОС Linux, работающей на процессоре RISC-V. По окончании этого курса вы получите достаточно глубокое понимание архитектуры инструкций процессора RISC-V и получите знания, необходимые для программирования на языке Assembler для конкретных аппаратных средств.

### Цели обучения на курсе

В конце этого курса вы должны будете:

* разбираться в наборе инструментов программирования для RISC-V;
* получить понимание архитектуры инструкций процессора RISC-V;
* научиться программировать под RISC-V на языке Assembler на базовом уровне;
* уметь реализовывать программный код и запускать его под ОС Linux на RISC-V;
* понимать механизмы дизассемблирования и отладки программного кода.

### Прежде, чем мы начнём

Прежде, чем окунуться в дисциплину, мы настоятельно рекомендуем ознакомиться с её содержанием. Оно предоставляет наиболее важную информацию о курсе включая:

* обзор содержимого курса;
* входные требования для освоения дисциплины;
* цели изучения курса;
* тайминги освоения курса;

и многие другие

### Необходимые знания

Для успешного прохождения курса необходимо обладать базовыми знаниями в области программирования.

### Структура курса

Глава 1 посвящена обсуждению связи архитектуры RISC-V с языком Assembler, а также специфическим характеристикам языка.

В главе 2 рассматривается установка и использование ряда симуляторов и эмуляторов, а также обсуждается применение набора инструментов GNU toolchain для разработки приложений.

Глава 3 рассматривает базовые инструкции и расширения RISC-V, принцип кодирования инструкций языка Assembler в машинный код, основные категории базовых целочисленных инструкций.

В главе 4 рассматривается применение меток, директив языка и псевдоинструкций, а также то, как реализовать управление работой программы и зачем использовать бинарный интерфейс приложения (ABI).

В главе 5 рассматривается написание программ на языке Assembler с использованием системных вызовов и библиотек C.

Глава 6 рассматривает реализацию простейших структур данных и использование функций malloc и free - функций библиотеки C для выделения и освобождения оперативной памяти. Также обсуждается написание программ с использованием чисел с плавающей запятой.

# Глава 1. Общая информация о языке Assembler

## Введение

### О чём эта глава

Эта глава знакомит слушателя с основными знаниями в области архитектуры RICS-V и языка Assembler. Если говорить вкратце, она охватывает демонстрацию способов применения языка Assembler для программирования под архитектуру RISC-V и в целом понимание того, почему знание языка Assembler является преимуществом для любого разработчика программного обеспечения.

### Цели главы

После освоения материала главы вы должны будете:

* понимать связь между языком Assembler и микропроцессором RISC-V;
* понимать особенности языка Assembler;
* понимать преимущества от знания языка Assembler;
* понимать взаимосвязь между микропроцессорами, языком Assembler и архитектурой RISC.

## Общая информация о языке Assembler

### RISC-V

RISC-V – это спецификация архитектуры и набора инструкций для 32, 64 и 128 битных микропроцессоров (ISA, Instruction Set Architecture). ISA обеспечивает основу для языка машинных кодов микропроцессоров. Она описывает стандарт, которому должен соответствовать микропроцессорRISC-V.

Термин RISC расшифровывается как Reduced Instruction Set Computer, компьютер с ограниченным набором инструкций, а используемый суффикс «V» соответствует римской цифре 5, которая показывает версию RISC-архитектуры, разрабатываемой университетом Беркли, Калифорния.

Спецификации архитектуры RISC-V опубликованы под лицензией «Creative Commons», таким образом, RISC-V является открытой архитектурой и набором инструкций, что позволяет любому желающему проектировать процессоры на этой архитектуре без каких-либо отчислений на использование прав собственности.

Руководство с описанием набора инструкций для архитектуры RISC-V разбито на два тома. Первый том, описывающий инструкции пользовательского уровня, содержит основную информацию об архитектуре, базовый набор инструкций для 32-х, 64-х и 128-ми битных целочисленных архитектур, стандартных расширениях базового набора инструкций и основные соглашения. Говоря в общем, первый том содержит всю необходимую информацию для того, чтобы можно было писать программы на языке Assembler, поскольку большинство пользовательских программ на языке Assembler задействуют именно инструкции пользовательского уровня. Второй том руководства содержит описание архитектуры привилегированных инструкций, которые используются в основном при разработке операционных систем и низкоуровневых решений для встраиваемых систем.

В рамках этого курса будут рассмотрена информация лишь из первого тома.

### Язык Assembler

Языки программирования Assembler – группа языков программирования, которые позволяют представить в удобном для чтения и написания человеком представление машинного кода. Инструкции, написанные на этих языках, могут быть напрямую транслированы в машинные коды. Языки Assembler – языки программирования самого низкого уровня и различаются в зависимости от того, под какую архитектуру микропроцессора они используются. Таким образом говорят, что язык Assembler – это аппаратно-зависимый язык.

Программы на языке Assembler пишут с использованием простых текстовых редакторов или специальных редакторов кода – интегрированных сред разработки. Сборка программ с использованием утилит, называемых ассемблерами (сборщиками), генерирует исполняемый машинный код.

Поскольку есть прямое соответствие между набором инструкций в машинных кодах и инструкциями на языке Assembler, существует процесс дизассемблирования (Disassembly), который является логически обратным процессу сборки исполняемых файлов из исходного кода и позволяет получить, наоборот, исходный код на языке Assembler из скомпилированного исполняемого бинарного кода. При выполнении этой процедуры можно получить практически полный код на языке Assembler за исключением некоторых объектов исходного кода, которые теряются в процессе компиляции, например, имена меток, имена переменных, комментарии исходного кода и т.д.

Знание языка Assembler и способов его применения для решения практических задач помогает понять взаимодействие между программным и аппаратным обеспечением. В частности, эти знания принесут пользу в решении таких задач, как проектирование компьютерной архитектуры, аппаратное программирование и близкое к аппаратному, отладка программного обеспечения, обратный инжиниринг (reverse engineering) и разработка компиляторов.

Обычно части встраиваемых систем и операционные системы реализуются на языке Assembler, поскольку имеют самое близкое отношение к аппаратной составляющей: загрузчики, обработчики прерываний, драйвера устройств и различные процедуры и функции для управления ресурсами процессора.

Отладка программ, то есть поиск и устранение неисправностей в программном обеспечении, зачастую предполагает пошаговое исполнение программного кода с анализом работы машинных инструкций как раз на уровне инструкций языка Assembler. Решение задач обратного проектирования (reverse engineering), в ходе которых программист анализирует алгоритм работы скомпилированного исполняемого кода, может быть выполнено лишь при понимании принципа исполнения машинных инструкций, и, следовательно, знания языка Assembler.

Также знания языка Assembler требует разработка компиляторов и анализ промежуточных данных, формируемых компилятором. Программы, написанные на высокоуровневых языках программирования, таких как С++ или Rust, транслируются в промежуточное представление на языке Assembler в процессе компиляции.

### Краткие сведения о микропроцессоре

Для достижения целей курса мы рассмотрим микропроцессор лишь поверхностно. Микропроцессор имеет модуль управления (control unit), арифметико-логическое устройство, АЛУ (arithmetic logic unit, ALU), регистры (registers) и шины данных и команд (signal/data buses) для обмена данными между микропроцессором и внешней памятью.

Микропроцессор выполняет инструкции с применением модуля управления. Любая компьютерная программа – это набор указаний микропроцессору о том, что нужно делать и с чем. Основная задача модуля управления заключается в трансляции инструкций микропроцессору и контроле за ходом исполнения программ.

Исходя из названия, арифметико-логическое устройство (АЛУ) позволяет выполнять целочисленные логические и арифметические операции, при этом логические операции – это побитовые операции над целыми числами.

Регистры процессора – небольшие именованные ячейки памяти, расположенные прямо внутри микропроцессора. Микропроцессор умеет быстро загружать данные в регистры из внешней памяти или получая их от других устройств, обрабатывать данными внутри регистров, а также записывать данные из своих регистров во внешнюю память или передавать на внешние устройства.

Компьютерная программа размещается в памяти. Области памяти, куда могут быть загружены как код, так и данные, имеют адреса – целочисленные значения, отождествляемые с номерами ячеек памяти (обычно однобайтными). Специальный регистр, называемый «программный счётчик», хранит адрес инструкции, которая будет выполнена следующей. Значением этого регистра пользуется модуль управления микропроцессора.

Как будет упомянуто далее, доступ к аппаратным устройствам обычно также реализуется по адресам.

### Классический RISC-конвейер

Обычно набор инструкций микропроцессоров относят к RISC (Reduced Instruction Set Computer, компьютер с ограниченным набором инструкций) или CISC (Complex Instruction Set Computer, компьютер со сложным набором инструкций). Разделение очень простое: в RISC-архитектуре есть меньшее, чем в CISC, число инструкций и часто некоторые простые действия приходится описывать несколькими RISC-командами, в то время как в CISC-архитектуре может быть всего одна команда, реализующая требуемое действие. Но, при этом, для RISC-архитектуры характерно фиксированное время исполнения каждой команды в тактах и фиксированный размер каждой команды, чего нельзя сказать о командах CISC-архитектуры.

Для обработки какой-либо инструкции, типичный микропроцессор с RISC-архитектурой выполнит конвейер операций, состоящий из пяти стадий:

1. **извлечение инструкции, instruction fetch** (IF)
2. **декодирование инструкции, instruction decode** (ID)
3. **выполнение инструкции, instruction execute** (EX)
4. **доступ к памяти, memory access** (MEM)
5. **запись результата, write back** (WB)

Машинная инструкция извлекается из области памяти, на которую ссылается регистр – счётчик инструкций. В ходе декодирования инструкции микропроцессор понимает, что он должен сделать и какие ещё данные ему необходимо извлечь для выполнение операции. Затем инструкция выполняется, чаще всего при этом задействуется блок АЛУ. Если инструкция подразумевает работу с памятью, выполняется обращение к памяти. И затем, если после выполнения инструкции формируется результат, он записывается в один из регистров.

Этот конвейер (англ. pipeline) может быть выполнен как единой последовательностью для какой-либо инструкции, так и конвейеризован. Конвейеризация – процесс, в ходе выполнения которого происходит разделение выполнения инструкции на последовательно выполняемые шаги, за счёт чего можно достичь распараллеливания выполнения этапов обработки: выполнив для первой команды стадию IF, её результат передаётся для дальнейшего выполнения на стадию ID, и микропроцессор может сразу же начать выполнение второй инструкции (снова выполнить её извлечение из памяти).

# Глава 2. Инструменты разработчика

## Введение

### Обзор главы

Эта глава посвящена настройке окружения и инструментов разработчика для написания программного кода на языке Assembler для процессора RISC-V и его запуска на компьютере. Существует достаточно большое количество различных программ и утилит, позволяющих писать и тестировать небольшие программы, которые достаточно сильно отличаются своей функциональностью. В этой главе мы рассмотрим небольшую часть таких инструментов, позволяющих писать код для микропроцессоров RISC-V, транслировать его в исполняемые модули, а также запускать и отлаживать его, работая при этом на обычных ПК с ОС Linux. В частности, будут рассмотрены: симулятор Ripes, эмулятор аппаратного обеспечения Qemu с набором инструментов GNU toolchain, одноплатный компьютер StarVision VisionFive.

Вам потребуется ОС Linux (предпочтительно – Debian) в качестве хоста для запуска симуляторов. Вы также можете использовать WSL[[1]](#footnote-1) (Windows Subsystem for Linux) или macOS, однако использование этих средств здесь рассматриваться не будет.

### Цели обучения

После прохождения этого раздела вы будете знать, как:

* устанавливать и использовать любое рассмотренное в разделе ПО;
* знать и уметь применять наборы инструментов GNU (GNU toolchain) для сборки приложений.

## Набор инструментов разработчика

### Обзор наборов инструментов

На рис.1. показан процесс того, как происходит преобразование исходного кода программы в исполняемый бинарный файл. Шаги этого процесса обычно скрыты внутри специального ПО, которое называется интегрированная среда разработки (integrated development environment, IDE). Работая в таких средах, вы просто печатаете кода в редакторе и сохраняете его в текстовом виде в файле исходного кода. Этот файл передаётся ассемблеру (или кросс-ассемблеру, специальной программе, позволяющей, находясь на компьютере одной архитектуры, компилировать бинарный код для совершенно другой архитектуры), который переводит его содержимое в так называемый объектный файл - бинарный код, в точности соответствующий машинному представлению инструкций из транслируемого файла, но при этом не содержащий дополнительную информацию, необходимую для запуска и исполнения программы операционной системой. Для преобразования объектного файла в исполняемый используется программа, которая называется компоновщик (или линкер, от англ. linker). Компоновщик принимает на вход объектный файл (или группу объектных файлов, если исходных файлов в проекте компилировалось несколько) и скрипт линковки – специальный набор указаний о том, как сегменты, описанные в исходном коде, должны размещаться в памяти при загрузке программы операционной системой и её исполнении. Результатом этой процедуры является исполняемый файл.

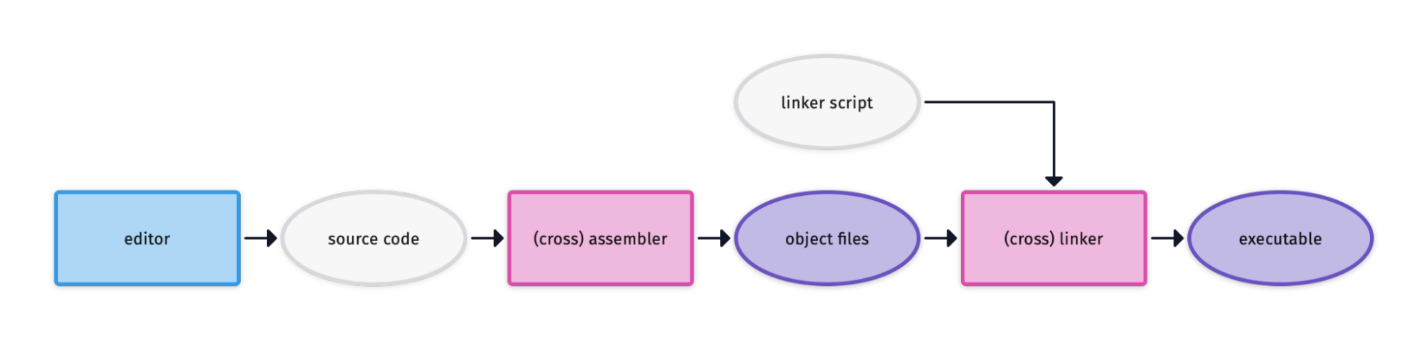


Рис.1. Процесс преобразования исходного кода в исполняемый

Набор инструментов (toolchain) включает в себя все описанные выше утилиты, конфигурационные файлы и прочие файлы и программы, необходимые для перевода текстового представления исходного кода в бинарный исполняемый файл.

### Симулятор Ripes

Ripes – симулятор, который позволяет отслеживать выполнение машинного (исполняемого) кода на процессорах с архитектурами RV32IMC и RV64IMC (суффиксы IMC показывают расширения архитектуры, добавляющие новую функциональность, и будут рассмотрены позднее в третьей части). Симулятор Ripes доступен по ссылке: <https://github.com/mortbopet/Ripes>, на сайте имеется возможность скачать исполняемые файлы для ОС Linux, Windows и macOS. На рис.2. показан пользовательский интерфейс симулятора Ripes. Слева расположен редактор программного кода на языке Assembler, по центру – сгенерированный машинный код, справа расположено коно, показывающее состояние регистров процессора, а внизу расположена консоль, в которую перенаправляется вывод программы и служебная информация. Помимо показанного интерфейса Ripes может визуализировать состояние процессора и его компонентов, кэш, память и простой ввод/вывод. Симулятор Ripes очень полезен для быстрого погружения в программирование на языке Assembler.

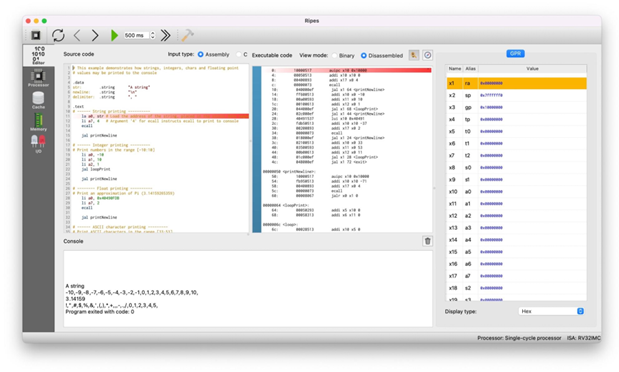


Рис.2. Интерфейс симулятора Ripes, запущенного под MacOS

### Эмулятор Qemu и набор инструментов GNU-toolchain для процессора RISC-V

Существуют различные способы обеспечения эмуляции систем на основе микропроцессоров RISC-V благодаря использованию Qemu (доступен по ссылке <https://www.qemu.org>) - эмулятора аппаратного обеспечения, который позволяет эмулировать как целые системы, так и отдельные программы в так называемом «пользовательском режиме» (англ. «user-mode»). В случае эмуляции целой системы операционная система запускается так, как будто она действительно работает на реальном (а не на эмулируемом) железе. При запуске приложений в пользовательском режиме (user-mode emulation) отдельная программа может быть запущена на текущей ОС, даже если она была скомпилирована под другую архитектуру.

В рассматриваемых примерах мы будем считать, что работа идёт на машине с установленной ОС Debian Linux. С более детальной информацией можно ознакомиться на сайте «Debian wiki for RISC-V» по адресу <https://wiki.debian.org/RISC-V>. Далее будет использоваться установка готовых пакетов с использованием менеджера пакетов Debian Advanced Packaging Tool (apt), тем не менее, по адресу <https://github.com/riscv-collab/riscv-gnu-toolchain> располагается инструкция о том, как можно собрать рассматриваемые утилиты из исходников.

Для установки Qemu на компьютер наберите в консоли следующий набор команд:

sudo apt install qemu-system-misc qemu-user-static binfmt-support opensbi u-boot-qemu

Теперь у вас появилась возможность эмулировать систему на процессоре RISC-V. Введите команду:

qemu-riscv64-static -help

Вы должны получить в ответ текст, который начинается со следующей строки:

usage: qemu-riscv64 [options] program [arguments...]

...

Следующим шагом идёт установка набора инструментов кросс-компилятора для сборки программ. Введите следующие команды:

sudo apt install gcc-riscv64-linux-gnu

sudo ln -s /usr/riscv64-linux-gnu/lib/ld-linux-riscv64-lp64d.so.1 /lib

И следом за ними команду:

riscv64-linux-gnu-as --version

GNU assembler (GNU Binutils for Debian) 2.35.2

...

Такой вывод говорит об успешной установке набора инструментов, позволяющего запускать эмуляцию программ в режиме пользователя.

После установки эмулятора и набора средств кросс-компилятора необходимо установить отладчик для процессоров с архитектурой RISC-V. В зависимости от версии ОС Debian Linux возможно потребуется поставить нестабильные версии пакетов (для более новых версий ОС).

Отладчик с поддержкой приложений для архитектуры RISC-V называется gdb-multiarch и может быть установлен следующей командой:

sudo apt install gdb-multiarch

### Пример использования набора инструментов

В качестве примера использования установленного выше набора инструментов необходимо написать исходный файл на языке Assembler. Для этого необходимо открыть текстовый редактор (например, nano или vi для работы в консоли, а также gedit, kate, sublime или code для работы в графическом режиме) и ввести следующий код, после чего необходимо сохранить его с именем example.s.

.text

.globl \_start

\_start:

addi x10, x0, 7

addi x17, x0, 93

ecall

Для сборки и запуска программы, а также проверки результата её исполнения необходимо ввести следующую последовательность команд:

riscv64-linux-gnu-as -o example.o example.s

riscv64-linux-gnu-ld -o example example.o

qemu-riscv64-static example

echo $?

7

Первая команда вызывает для написанного исходного кода компилятор программ на языке Assembler, который транслирует исходный код в машинные инструкции объектного файла. Вторая команда вызывает компоновщик, который дополняет код объектного файла не обходимыми инструкциями и генерирует исполняемый файл с именем example. Третья команда запускает скомпилированный бинарный исполняемый файл на исполнение в эмуляторе архитектуры RISC-V. Поскольку этот код ничего не делает, кроме как завершает работу с указанным кодом возврата, на экране в результате выполнения программы ничего не будет выведено. Для того, чтобы увидеть код возврата последней вызываемой команды или программы, воспользуемся командой командного интерпретатора bash (строка 4). В качестве результата в консоли должно напечататься число 7.

Помимо компилятора и компоновщика, набор инструментов также содержит утилиту objdump (файл riscv64-linux-gnu-objdump) - дизассемблер, который позволяет провести обратное описанным выше операциям действие и на основе скомпилированного бинарного кода получить набор инструкций на языке Assembler. Пример вызова утилиты дизассемблирования показан ниже:

riscv64-linux-gnu-objdump -sd example

example: file format elf64-littleriscv

Contents of section .text:

100b0 13057000 9308d005 73000000 ..p.....s...

Contents of section .riscv.attributes:

0000 412d0000 00726973 63760001 23000000 A-...riscv..#...

0010 05727636 34693270 305f6d32 70305f61 .rv64i2p0\_m2p0\_a

0020 3270305f 66327030 5f643270 3000 2p0\_f2p0\_d2p0.

Disassembly of section .text:

00000000000100b0 <\_start>:

100b0: 00700513 li a0,7

100b4: 05d00893 li a7,93

100b8: 00000073 ecall

Первая часть вывода objdump показывает содержимое файла в шестнадцатеричном виде (за это отвечает передаваемый программе ключ -s), вторая часть – собственно дизассемблированный код (ключ -d). К этому примеру мы ещё вернёмся позднее, сейчас он служит просто демонстрацией работоспособности установленных пакетов набора инструментов.

Наконец, можно использовать отладчик. Для этого введите следующие команды:

qemu-riscv64-static -g 1234 example &

gdb-multiarch example

...

Reading symbols from example...

(No debugging symbols found in example)

(gdb) target remote :1234

Remote debugging using :1234

0x00000000000100b0 in \_start ()

(gdb) display /3i $pc

1: x/3i $pc

=> 0x100b0 <\_start>: li a0,7

0x100b4 <\_start+4>: li a7,93

0x100b8 <\_start+8>: ecall

(gdb) si

0x00000000000100b4 in \_start ()

1: x/3i $pc

=> 0x100b4 <\_start+4>: li a7,93

0x100b8 <\_start+8>: recall

0x100bc: .2byte 0x2d41

(gdb) si

0x00000000000100b8 in \_start ()

1: x/3i $pc

=> 0x100b8 <\_start+8>: ecall

0x100bc: .2byte 0x2d41

0x100be: .2byte 0x0

(gdb) c

Continuing.

[Inferior 1 (process 1) exited with code 07]

После запуска эмулятора с подключением возможности удалённой отладки через порт 1234, запускается программа-отладчик gdb-multiarch. Отладчик подключается к удалённой системе благодаря команде target remote с указанием требуемого порта (1234). Команда display /3i $pc показывает 3 следующие за текущей инструкции, команда si предписывает выполнить одну инструкцию (сделать один шаг по коду), а команда c даёт указание продолжить выполнение программы. Чтобы выйти из отладчика, нужно ввести команду q.

### Запуск Visual Studio Code и PlatformIO вместе с Qemu и пакетом инструментов RISC-V GNU Toolchain

Помимо работы в командной строке, также возможно настроить работу в интегрированной среде разработки. В качестве примера рассмотрим, как можно использовать среду разработки Visual Studio Code (доступна по адресу <https://code.visualstudio.com>) с установленным расширением PlatformIO, пример работы которого показан на рис.3 (расширение доступно по адресу <https://platformio.org>).

Откройте терминал в среде VS Code, находясь внутри PlatformIO и выбрав пункт меню «New terminal», и введите следующую команду:

pio platform install <https://github.com/e1d1-academy/platform-linux_riscv.git>

Также можно импортировать тестовый проект, как показано в следующем разделе, и проверить, запускается ли он корректно.

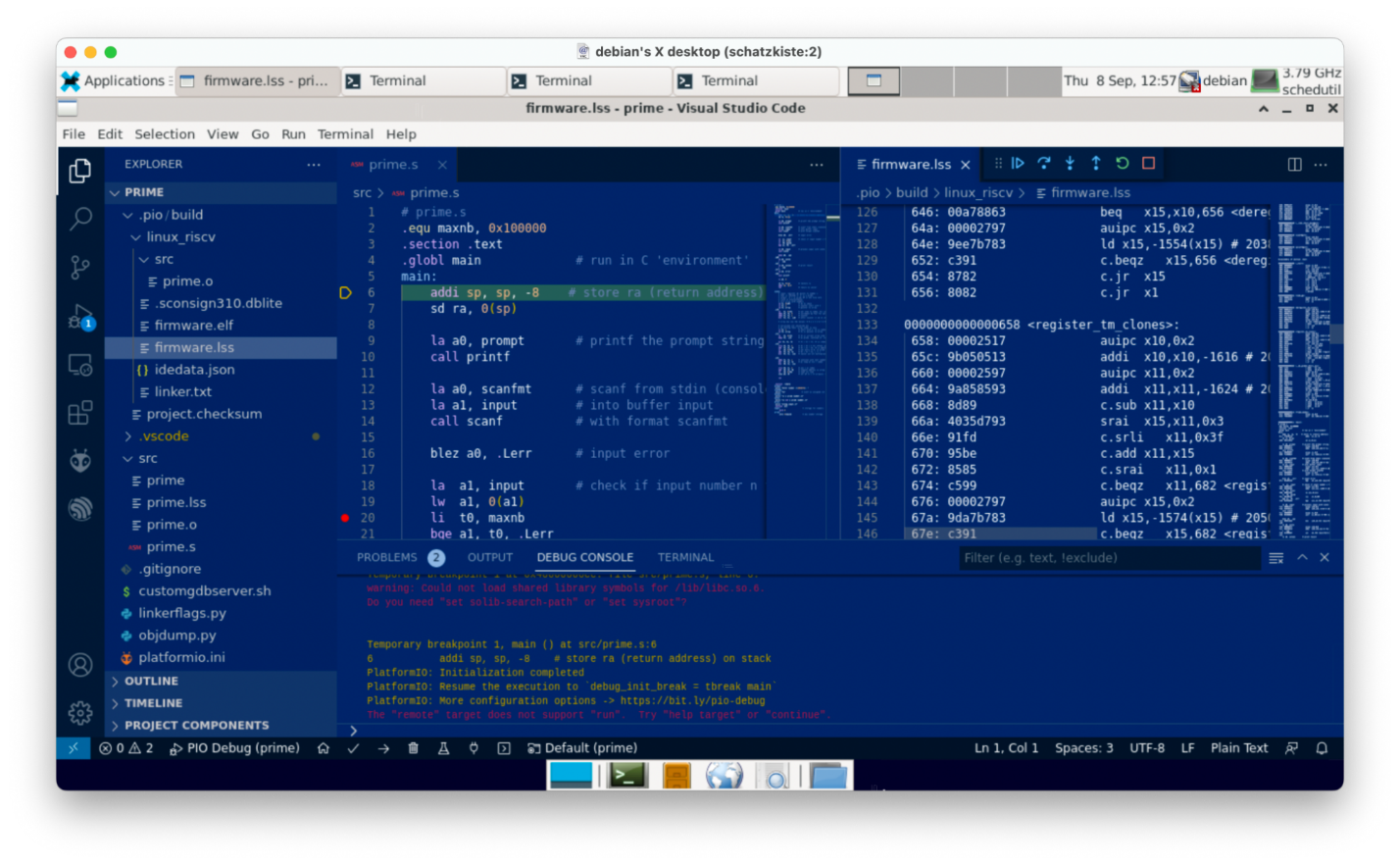


Рис.3. Пример интерфейса среды разработки VS Code с расширением PlatformIO, работающим в режиме отладки

Примеры к этому курсу выложены на GitHub, в каждом из каталогов примеров лежит файл platformio.ini. Эти файлы подготовлены для запуска примеров в Qemu с использованием пользовательского режима.

### Пример работы в Visual Studio Code с расширением PlatformIO

Для получения окружения для программирования на языке Assembler в расширении PlatformIO среды разработки VS Code наберите к консоли расширения PlatformIO команду:

https://github.com/e1d1-academy/platform-linux\_riscv.git

В расширении PlatformIO есть возможность работать под различны платформы, и если в диалоге вы наберёте «Linux», вам будет доступно окружение для ОС Linux для архитектуры RISC-V.

В появившемся списке вы можете кликнуть на требуемое окружение и увидите, что там есть раздел с примерами, которые можно импортировать и попробовать их в работе. Все примеры идут с конфигурационными файлами, которые позволяют сгенерировать шаблон приложения с вашим собственным кодом на языке Assembler. Этот код можно компилировать прямо из среды VS Code, а также запускать его как в обычном режиме, так и в режиме отладки.

В результате работы показанного примера на экран выводится фраза «Hello World!».

### 

### Мини-компьютер StarFive VisionFive и набор инструментов RISC-V GNU Toolchain

Одноплатный компьютер StarFive VisionFive поставляется с двухъядерным RISC-V процессором с архитектурой RV64IMAFDC. По умолчанию поставляется с ОС Fedora Linux и предустановленным набором инструментов. Этот одноплатный компьютер можно использовать как «живую» аппаратную платформу с процессором RISC-V и работать на нём без необходимости устанавливать эмуляторов, симуляторов и кросс-ассемблеров.

На рис.4 показан скриншот рабочего стола xfce4 с четырьмя открытыми терминалами, запущенного на плате VisionFive.

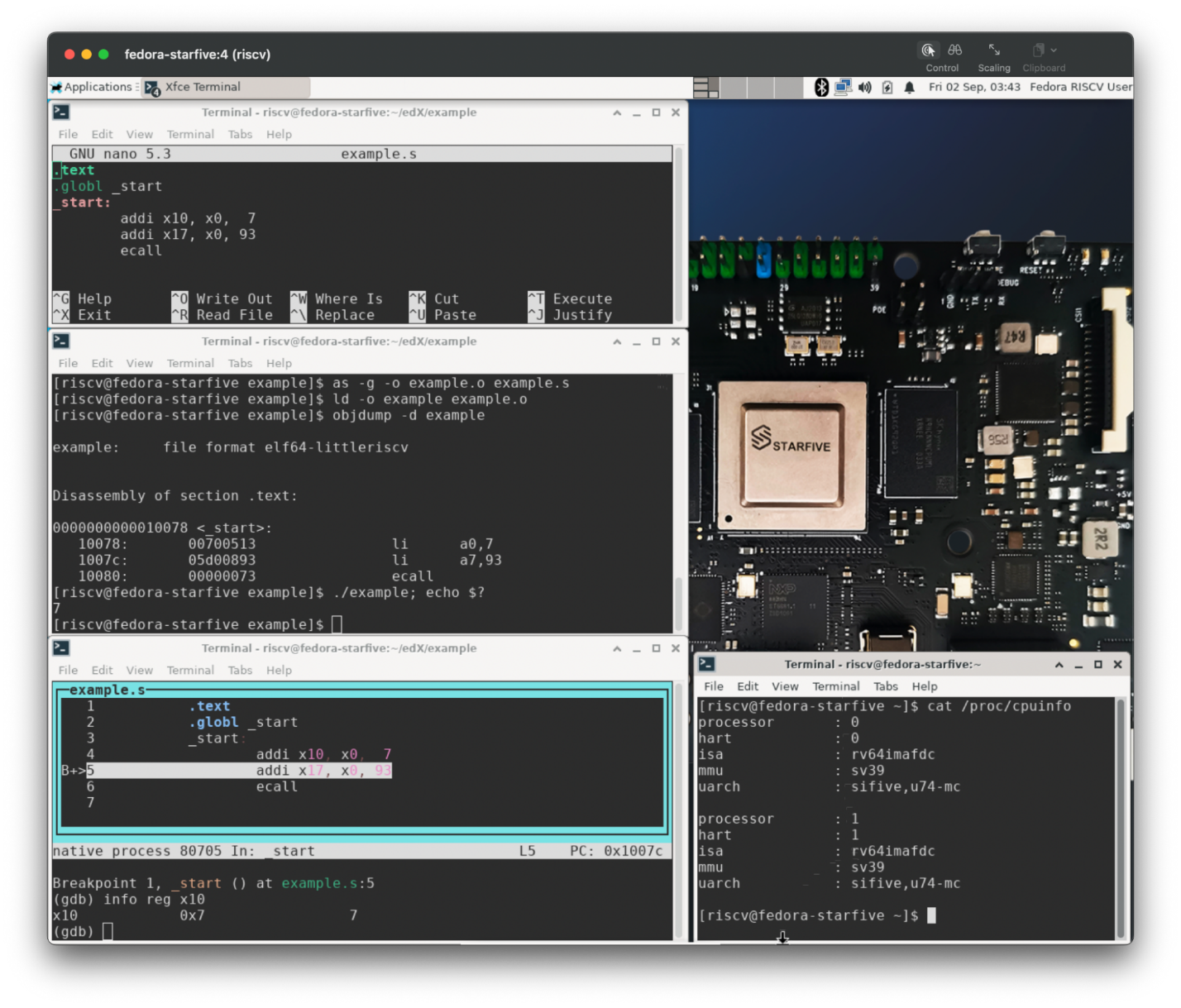


Рис.4. Рабочий стол Linux на плате VisionFile с запущенными терминалами с набором инструментов GNU toolchain

В левом верхнем углу расположен первый терминал, в нём открыт текстовый редактор nano с исходным кодом файла **example.s**. Чтобы воспроизвести это, наберите в консоли команду:

nano example.s

Затем вставьте скопированный программный код (или напишите его самостоятельно) и запишите содержимое файла на диск, нажав сочетание клавиш **^O** (Сtrl+o).

Во втором терминале, расположенном слева посередине, показан результат сборки и линковки с последующей распечаткой дизассемблированного кода, выполненной с использованием утилиты objdump, а также результат запуска программы – демонстрация возвращаемого программой кода возврата. Отладочная информация внесена в исполняемый бинарный файл благодаря ключу «-g» компилятора. Для воспроизведения этих результатов введите в консоли команды:

as -g -o example.o example.s  
ld -o example example.o  
objdump -d example  
./example

echo $?

В третьем терминале, расположенном слева внизу, показана работа отладчика. Для воспроизведения введите в консоли команду:

gdb example

После запуска отладчика были использованы следующие управляющие команды: «layout src» – показать исходный код, «b 5» - установить точку останова на строке 5, «r» - запустить код на исполнение до точки останова, «info reg x10» - показать содержимое регистра x10:

(gdb) layout src  
(gdb) b 5  
(gdb) r  
(gdb) info reg x10

Из отладчика можно выйти, набрав команду «q».

И, наконец, четвёртый терминал, расположенный справа, выводит информацию о процессоре.

**Исходные коды примеров:**

Исходные коды доступны в репозитории github по адресу: <https://github.com/U2654/LFD117x>

# Глава 3. Набор непривилегированных инструкций RISC-V

## Введение

### Обзор главы

Эта глава раскрывает основы непривилегированных целочисленных инструкций архитектуры RISC-V и содержит базовую информацию для написания программ на языке Assembler.

### Цели обучения

После освоения этой главы вы должны быть способны:

* понимать, что архитектура RISC-V определяется через набор инструкций и его расширения;
* понимать принцип перевода команд на языке Assembler в машинный код;
* знать основные категории целочисленных инструкций.

## Набор непривилегированных инструкций RISC-V

### Основной набор целочисленных инструкций

Набор непривилегированных инструкций архитектуры RISC-V описывает четыре базовые набора инструкций:

1. RV32I (32-х битные целочисленные)
2. RV32E (32-х битные встроенные)
3. RV64I (64-х битные целочисленные)
4. RV128I (128-х битные целочисленные)

Числа 32, 64, и 128 характеризуют размер регистра и адресного пространства. Например, 32-х битный регистр может хранить данные размером до 2^32 бит. Набор инструкций RV32I содержит всего 40 инструкций.

### Расширения и именования

Любой процессор RISC-V поддерживает как минимум один базовый набор инструкций. Например, обозначение RV32I означает, что процессор поддерживает 32-х битный целочисленный набор команд. Распространены дальнейшие расширения базового набора инструкций. Тип расширения, который поддерживает процессор, указывается в качестве суффикса к базовой аббревиатуре набора инструкций. Распространены следующие расширения:

| **Сокращение** | **Стандартное расширение** |
| --- | --- |
| M | целочисленные умножение и деление  integer multiplication and division |
| A | атомарные инструкции  atomic instructions |
| F | числа с плавающей запятой одинарной точности  single-precision floating point |
| D | числа с плавающей запятой двойной точности  double-precision floating point |
| Q | числа с плавающей запятой четверной точности  quad-precision floating point |
| C | сжатые инструкции  compressed instructions |
| V | векторные операции  vector operations |

Примеры:

* микропроцессор, построенный на архитектуре RV32IMAC, поддерживает базовый набор 32-х битных инструкций (RV32I), целочисленные умножение и деление (M), атомарные инструкции (A) и сжатые инструкции (C).
* микропроцессор, построенные на архитектуре RV64IMAFDC поддерживает базовый набор 64-х инструкций (RV64I), целочисленные умножение и деление(M), атомарные инструкции (A), числа с плавающей запятой одинарной (F) и двойной точности (D), а также сжатые инструкции (C).
* иногда можно встретить микропроцессоры, для которых полная информация зашита внутри процессора и может быть прочитана только после чтения специального регистра. В этом случае суффиксы будут оканчиваться на SU, например, RV64IMAFDCSU. Такое обозначение означает, что процессор поддерживает режим супервизора (S) и пользовательский режим (U).

### Регистры RISC-V

Базовый набор инструкций определяет наличие 32 регистров и программного счётчика. Регистры именуются от x0 до x31. Размер регистра соответствует размеру базовых инструкций, например, для процессора RV32I размер каждого из регистров будет составлять 32 бита. Регистр x0 хранит значение 0, которое нельзя поменять.

Расширения предоставляют большее число регистров. Например, расширения, связанные с вычислениями с плавающей запятой, предоставляют дополнительные регистры работы с дробной арифметикой.

Использование регистров языками более высокого уровня, чем Assembler, определяется бинарным интерфейсом для приложений (application binary interface, ABI). Этот интерфейс определяет то, как регистры должны использоваться в процессе трансляции программы компилятором из описания на высокоуровневом языке в машинный код. ABI предписывает использование дополнительных имён регистров и их назначение:

| **Регистр** | **Имя ABI** | **Описание** |
| --- | --- | --- |
| x0 | zero | Константный ноль  Zero constant |
| x1 | ra | Адрес возврата  Return address |
| x2 | sp | Указатель на область стека  Stack pointer |
| x3 | gp | Глобальный указатель  Global pointer |
| x4 | tp | Указатель потока  Thread pointer |
| x5-x7 | t0-t2 | Временные значения  Temporaries |
| x8 | s0 / fp | Сохраняемый регистр / Указатель на фрейм  Saved / Frame pointer |
| x9 | s1 | Сохраняемый регистр  Saved register |
| x10-x11 | a0-a1 | Аргументы функций и возвращаемые значения  Function args. / return values |
| x12-x17 | a2-a7 | Аргументы функций  Function arguments |
| x18-x27 | s2-s11 | Сохраняемые регистры  Saved registers |
| x28-x31 | t3-t6 | Временные значения  Temporaries |
| pc | - | Программный счётчик  Program counter |

Архитектура RISC-V предполагает выполнение арифметических операций через регистры: для модификации данных, находящихся в памяти, необходимо загрузить данные из памяти в один из регистров, выполнить операцию над данными, и переместить результат из регистра обратно в память.

## Обзор инструкций

### Простой пример

Для понимания принципа использования инструкций, рассмотрим простой пример, в котором используется только одна инструкция: **addi rd, rs1, imm**, которая добавляет константное целое **imm** (в диапазоне 12-ти битного знакового: -2048…2047) к содержимому регистра **rs1** и записывает результат в регистр **rd**. Другими словами выполняется операция rd = rs1 + imm. После получения инструкции программный счётчик инкрементируется и будет содержать адрес следующей инструкции, что можно условно записать как pc = cp + 4, поскольку размер инструкции в машинном коде равен 4 байтам.

Инструкция **addi x1, x0, 1** добавляет число **1** к значению **0**, которое хранится в регистре **x0** (в регистре x0 всегда хранится значение 0) и пишет результат, равный 1, в регистр **x1**. Машинный код инструкции **addi x1, x0, 1** – это (в шестнадцатеричном представлении) 0x00100093. Префикс **0x** перед числом показывает, что число задано в шестнадцатеричной системе счисления. Далее мы увидим, как именно работает кодирование чисел. По аналогии, инструкция **addi x1, x1, 1** добавляет значение 1 к содержимому регистра **x1** и записывает результат обратно в регистр **x1**. Машинный код этой инструкции - 0x00108093.

Теперь представим, что эти машинные коды располагаются в памяти начиная с адреса 0x0. Поскольку каждая инструкция занимает 4 байта, первая инструкция располагается по адресу 0x0, а вторая – по адресу 0x4. Программный счётчик указывает на начало исполняемого кода, следовательно, он равен значению 0x0:

Адрес Машинный код Инструкции

0x0 0x00100093 addi x1, x0, 1

0x4 0x00108093 addi x1, x1, 1

После запуска этих двух строк кода микропроцессор начинает выполнение с адреса 0x0. Как уже было объяснено выше, в результате выполнения первой строки в регистре x1 будет получено число 1, при этом программный счётчик будет инкрементирован до значения 0x4. Выполнение второй строки приведёт к увеличению значения регистра x1 на 1. Так как его значение было равно 1 в результате предыдущей операции, регистр после выполнения второй строки примет значение, равное 2.

На рис.5 показан пример того, как этот исходный код запускается в симуляторе Ripes.

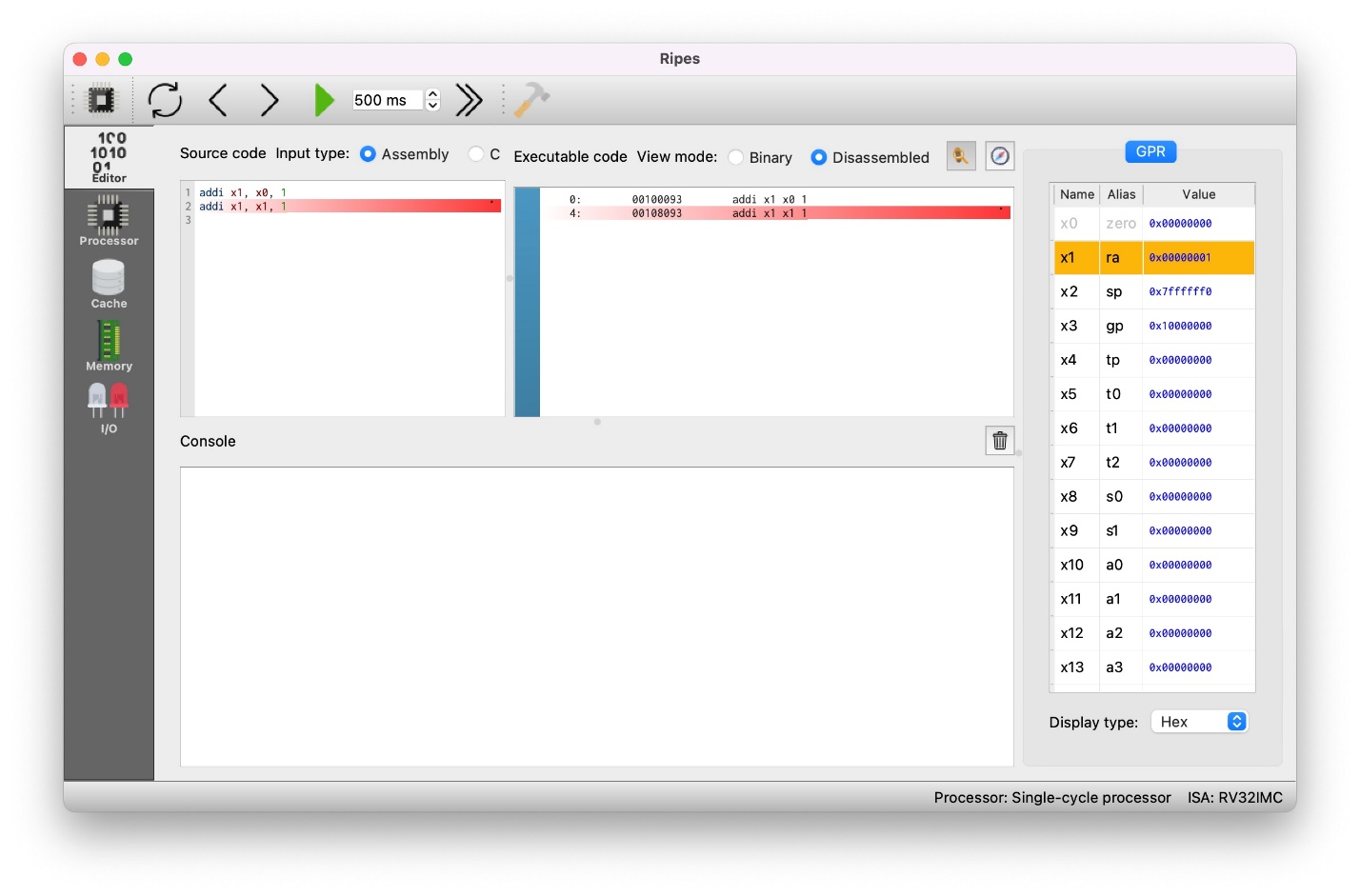
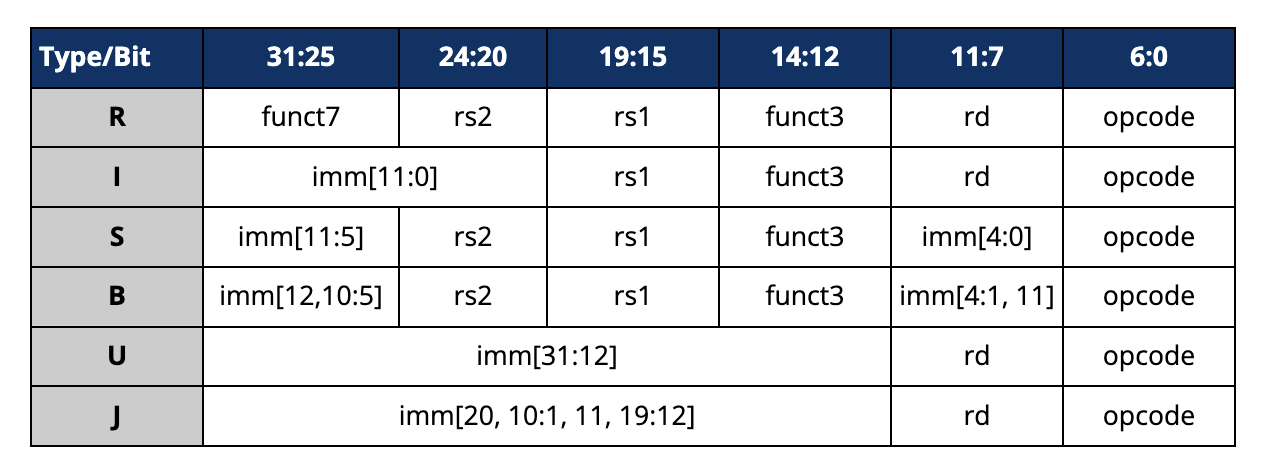


Рис.5. Запуск рассмотренного примера в симуляторе Ripes

### Кодирование инструкций

Базовые целочисленные инструкции в микропроцессорах архитектуры RISC-V разделены на пять типов, каждый из которых имеет свой тип кодирования. Типы соответствуют задачам, выполняемым машинной инструкцией: R-type (register, регистр), I-type (immediate, непосредственное значение), S-type (store, хранение), B-type (branch, ветвление), U-type (upper immediate, верхнее непосредственное значение), и J-type (jump, переход). Таблица ниже показывает принцип кодирования:



Рассмотрим предыдущий пример. Машинный код инструкции **addi x1, x0, 1** - это значение 0x00100093. Побайтное представление кода может быть получено из таблицы выше, зная, что код операции равен 0x13, значение funct3 равно 0x0, и что код типа I (целочисленная операция).

Таким образом, машинный код инструкции **addi x1, x0, 1** в побитовом представлении формируется следующим образом:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **addi x1, x0, 1** | 0000 0000 0001 | 0000 0 | 000 | 0000 1 | 001 0011 |

Поскольку каждые 4 бита могут быть объединены в шестнадцатеричное число, машинный код получается равным 0x00100093.

### Арифметические и логические операции (с непосредственными значениями)

Приведённая ниже таблица содержит инструкции, которые используют непосредственные данные. Программный счётчик обновляется после выполнения каждой инструкции и не приводится.

| **Инструкция** | **Тип** | **Формат** | **Код** | **funct3** | **Описание** |
| --- | --- | --- | --- | --- | --- |
| **addi** | ADD Immediate | I | 0010011 | 0x0 | rd = rs1 + imm |
| **xori** | XOR Immediate | I | 0010011 | 0x4 | rd = rs1 ^ imm |
| **ori** | OR Immediate | I | 0010011 | 0x4 | rd = rs1 | imm |
| **andi** | AND Immediate | I | 0010011 | 0x7 | rd = rs1 & imm |
| **slli** | Shift Left Logical Imm. | I | 0010011 | 0x1 | imm[11:5]=0x00, rd = rs1 << imm[4:0] |
| **srli** | Shift Right Logical Imm. | I | 0010011 | 0x5 | imm[11:5]=0x00, rd = rs1 << imm[4:0] |
| **srai** | Shift Right Arith. Imm. | I | 0010011 | 0x5 | imm[11:5]=0x20, rd = rs1 >> imm[4:0] |
| **slti** | Set Less Than Imm. | I | 0010011 | 0x2 | rd = (rs1 < imm)? 0:1 |
| **sltiu** | Set Less Than Imm. Un. | I | 0010011 | 0x3 | rd = (rs1 < imm)? 0:1 |

Как уже было показано выше, инструкции этого типа берут непосредственное значение imm, в качестве источника используется регистр rs1, в качестве регистра назначения - rd.

Поскольку инструкция **addi rd, rs1, imm** использует знаковое непосредственное значение imm, она может сработать как для сложения (если imm меняется от 0 до 2047), так и для вычитания (imm меняется от -2048 до -1).

Инструкции **xori**, **ori** и **andi** **rd**, **rs1**, **imm** выполняют побитовые операции xor (исключающее или), or (логическое сложение) и and (логическое умножение). Например, пусть в регистре x1 лежит значение 0x9, которому в побитовом представлении соответствует число 0b1001, тогда инструкция **xori x1, x1, 0x3** выполнит побитовое исключающее или над содержимым регистра x1 и числом 0x3, что можно представить следующим образом:

x1 = x1 xor 0x3 = 0b1001 xor 0b0011 = 0b1010 = 0xa

Инструкции **slli**, **srli**, и **srai** выполняют операции побитового сдвига с использованием непосредственного значения, задающего шаг сдвига. В качестве такого непосредственного значения может выступать положительное пятибитное число, представляющее значения от 0 до 31.

Код операции логического (побитового) сдвига влево синтаксически описывается следующим образом: **slli rd, rs1, imm**, и в процессе выполнения сдвигает каждый бит регистра, задаваемого как rs1, на imm позиций влево. Результат при этом записывается в регистр rd. При каждом сдвиге на 1 бит младший значащий бит устанавливается в 0, а значение старшего значащего бита теряется.

По аналогии с этим выполняется команда побитового сдвига вправо: **srli rd, rs1, imm**. Значение регистра rs1 побитового сдвигается вправо, результат записывается в регистр, задаваемый как rd. При этом в старший значащий бит при каждом сдвиге записывается значение 0, а значение младшего значащего бита теряется.

Результат выполнения команды srai в плане заполнение старшего бита отличается для арифметического сдвига вправо для команды **srai rd, rs1, imm**. Информация о старшем значащем бите для этой операции сохраняется. Если до сдвига он был равен 0, старший бит заполняется 0 после сдвига. Если он был равен 1, он заполняется 1. Такой сдвиг является знако-расширяющим для значений в rs1, которые записаны в дополнительном коде.

Рассмотрим пример. Пусть значение регистра x1 равно 0x9, двоичное представление которого записывается как 0b1001. Тогда в результате выполнения инструкции **slli x1, x1, 0x2** выполнится логический сдвиг содержимого регистра x1, который можно описать следующим кодом:

x1 = x1 << 0x2 = 0b1001 << 2 = 0b100100 = 0x24

Оставшиеся две инструкции **slti rd, rs1, imm** и **sltiu rd, rs1, imm** сравнивают содержимое регистра-источника с непосредственным значением и устанавливают значение регистра-приёмника в 0 в случае, если значение в регистре-источнике меньше непосредственного значения imm. Отличие между этими двумя командами в том, что команда slti использует значения в дополнительном коде и сравнение происходит со знаком.

### Арифметические и логические операции (с регистрами)

Следующая категория R-формата включает в себя арифметические и логические операции, которые используют три регистра: два регистра-источника и регистр-приёмник. Большинство этих инструкций по своему действию аналогичны рассмотренным выше. К этим инструкциям относятся:

| **Инструкция** | **Тип** | **Формат** | **Код** | **funct3** | **funct7** | **Описание** |
| --- | --- | --- | --- | --- | --- | --- |
| **add** | ADD | R | 0110011 | 0x0 | 0x00 | rd = rs1 + rs2 |
| **sub** | SUB | R | 0110011 | 0x0 | 0x20 | rd = rs1 - rs2 |
| **xor** | XOR | R | 0110011 | 0x4 | 0x00 | rd = rs1 ^ rs2 |
| **or** | OR | R | 0110011 | 0x6 | 0x00 | rd = rs1 | rs2 |
| **and** | AND | R | 0110011 | 0x7 | 0x00 | rd = rs1 & rs2 |
| **sll** | Shift Left Logical | R | 0110011 | 0x1 | 0x00 | rd = rs1 << rs2 |
| **srl** | Shift Right Logical | R | 0110011 | 0x5 | 0x00 | rd = rs1 >> rs2 |
| **sra** | Set Right Arith. | R | 0110011 | 0x5 | 0x20 | rd = rs1 >> rs2 |
| **slt** | Set Less Than | R | 0110011 | 0x2 | 0x00 | rd = (rs1 < rs2)? 0:1 |
| **sltu** | Set Less Than Un. | R | 0110011 | 0x3 | 0x00 | rd = (rs1 < rs2)? 0:1 |

Рассмотрим дополнение предыдущего примера с учётом новых инструкций:

Адрес Машинный код Инструкции

0x0 0x00100093 addi x1, x0, 1

0x4 0x00108093 addi x1, x1, 1

0x8 0x00108133 add x2, x1, x1

В последней строке примера содержимое регистра x1 добавляется к самому себе, результат записывается в регистр x2. В результате выполнения всех операций в регистре x2 содержится число 4.

Все эти инструкции в отдельном рассмотрении не нуждаются, поскольку являются функциональными аналогами операций из предыдущего раздела. Исключением является инструкция sub, но, как нам кажется, её функциональность является очевидной и в рассмотрении не нуждается.

### Загрузка и сохранение

Для того, чтобы получить доступ к памяти, реализован ряд инструкций, которые относятся к I-формату и предназначены для загрузки данных из памяти в регистры, а также ряд инструкций, относящихся к S-формату (S-Format, Save) для сохранения данных из регистров в память. Инструкции указаны в таблице:

| **Инструкция** | **Имя** | **Формат** | **Код** | **funct3** | **Описание** |
| --- | --- | --- | --- | --- | --- |
| **lb** | Load Byte | I | 0000011 | 0x0 | rd = M[rs1+imm][7:0] |
| **lh** | Load Half | I | 0000011 | 0x1 | rd = M[rs1+imm][15:0] |
| **lw** | Load Word | I | 0000011 | 0x2 | rd = M[rs1+imm][31:0] |
| **lbu** | Load Byte Un. | I | 0000011 | 0x0 | rd = M[rs1+imm][7:0] |
| **lhu** | Load Half Un. | I | 0010011 | 0x0 | rd = M[rs1+imm][15:0] |
| **sb** | Store Byte | I | 0010011 | 0x0 | M[rs1+imm][7:0] = rs2[7:0] |
| **sh** | Store Half | I | 0100011 | 0x1 | M[rs1+imm][15:0] = rs2[15:0] |
| **sw** | Store Word | I | 0100011 | 0x2 | M[rs1+imm][31:0] = rs2[31:0] |

Рассмотрим следующий пример. Предполагается, что в памяти по адресу 0x10000000находится значение 0x02.

Адрес Машинный код Значение

0x0 0x00100093 addi x1, x0, 0x1

0x4 0x01c09093 slli x1, x1, 28

0x8 0x00008103 lb x2, 0(x1)

Первые две инструкции записывают в регистр x1 значение 0x10000000. Для этого в регистр x1 записывается число 1, после чего сдвигается влево на 28 позиций, в результате чего в регистр x1 формируется значение 0x10000000. Третья инструкция загружает байт из памяти по адресу, задаваемому регистром x1.

Синтаксис вызова этой команды следующий: **lb rd, imm(rs1)**. Это можно развернуть в следующее представление: rd = M[rs1+imm][7:0]. Значение одного байта, задаваемого конструкций bits[7:0], загружается в регистра, определяемый как rd (register destination) из памяти по адресу, взятому из регистра, определяемого как rs1 (Register Source), к которому добавляется непосредственное значение смещения imm.

Из приведённого примера видно, что задавать адрес в регистре с использованием описанных инструкций довольно неудобно. Для исправления этого был разработан ряд инструкций U-формата (U-Format, Upper):

| **Инструкция** | **Имя** | **Формат** | **Код** | **funct3** | **Описание** |
| --- | --- | --- | --- | --- | --- |
| **lui** | Load Upper Imm. | U | 0110111 | - | rd = imm << 12 |
| **auipc** | Add Upper Imm. to PC | U | 0010111 | - | rd = PC + (imm << 12) |

Инструкция **lui rd, imm** задаёт значение регистра, определяемого как rd, равным непосредственному значению imm, сдвинутому на 12 бит влево. А инструкция **auipc rd, imm** устанавливает значение регистра, определяемого как rd, в результат сложения значения регистра программного счётчика PC и непосредственного значения imm, сдвинутого влево на 12 бит.

Разберём следующий пример:

Адрес Машинный код Значение Комментарий

0x0 0x100000b7 lui x1, 0x10000 Загружает 0x10000000 в регистр x1

0x4 0x10000117 auipc x2, 0x10000 Загружает 0x10000004 в регистр x2

Инструкция **lui x1, 0x10000** задаёт регистру x1 значение 0x10000 << 12, таким образом, x1 = 0x10000000, поскольку сдвиг влево на 12 бит «добавляет» три нуля в шестнадцатеричном представлении. Инструкция **auipc x2, 0x10000** устанавливает содержимое регистра x2 равным сумме текущего значения PC (счётчика команд), которое в данный момент равно 0x04 с прибавленным значением 0x10000 << 12. Таким образом, значение регистра x2 после выполнения этого кода будет равно 0x10000004.

### Управление

Каждая из предыдущих рассмотренных инструкций устанавливает программный счётчик PC на адрес следующей инструкции, расположенной в памяти: pc = pc + 4. Для реализации возможности контроля за процессом исполнения используются инструкции B-формата (B-Format, Branching), которые реализуют механизм ветвления. Они используются для условного перехода к инструкции, адрес которой указывается непосредственно в составе соответствующих команд в случае, если условие выполняется:

| **Инструкция** | **Тип** | **Формат** | **Код** | **funct3** | **Описание** |
| --- | --- | --- | --- | --- | --- |
| **beq** | Branch == | B | 1100011 | 0x0 | if (rs1 == rs2) pc+=imm |
| **bne** | Branch != | B | 1100011 | 0x1 | if (rs1 != rs2) pc+=imm |
| **blt** | Branch < | B | 1100011 | 0x4 | if (rs1 < rs2) pc+=imm |
| **bge** | Branch >= | B | 1100011 | 0x5 | if (rs1 >= rs2) pc+=imm |
| **bltu** | Branch < Un. | B | 1100011 | 0x6 | if (rs1 < rs2) pc+=imm |
| **bleu** | Branch >= Un. | B | 1100011 | 0x7 | if (rs1 >= rs2) pc+=imm |

Инструкция ветвления сравнивает значения регистров rs1 и rs2 способом, определяемым самой инструкцией. Если условие истинно, программный счётчик увеличивается на непосредственное значение imm. В выражении **blt rs1, rs2, imm** проверяется, что rs1 меньше, чем rs2, и если это так, программный счётчик PC изменяется следующим образом: pc = pc + imm. В противном случае он принимает значение адреса следующей инструкции в памяти (pc = pc +4).

Инструкции **blt** и **bge** учитывают знак числа, инструкции **bltu** и **bleu** – нет, воспринимают числа как беззнаковые.

Рассмотрим пример:

Адрес Машинный код Значение Комментарий

0x00 0xffb00093 addi x1, x0, -5 x1 = -5

0x04 0x00500113 addi x2, x0, 5 x2 = 5

0x08 0x0020c463 blt x1, x2, 0x8 if (x1 < x2) pc = pc + 8

0x0c 0x00100193 addi x3, x0, 1 skipped if (x1 < x2): x3 = 1

0x10 0x00200193 addi x3, x0, 2 x3 = 2

Инструкция по адресу 0x08 проверяет условие, что x1 < x2, то есть -5 < 5. Поскольку выражение истинно, программный счётчик устанавливается в значение 0x10 и инструкция по адресу 0x0c пропускается (не выполняется). Если бы инструкция по адресу 0x08 была **bltu**, отрицательное значение, хранящиеся в регистре x1 (-5) интерпретировалось бы как положительное значение, равное 0xFFFFFFFB (представление положительного числа, в двоичном коде идентичного отрицательному числу -5), а значит условие бы не выполнилось.

Помимо инструкций для условного перехода, есть инструкции для безусловного перехода: **jalr** и **jal**. Эти инструкции позволяют переходить на требуемый адрес путём модификации программного счётчика PC, однако при этом сохраняют адрес возврата, который равен PC+4 на момент вызова инструкции. Такой подход позволяет вернуться в то место, откуда произошёл переход. Именно так реализуется модульность разрабатываемых программ.

Инструкции записываются в разных форматах:

| **Инструкция** | **Тип** | **Формат** | **Код** | **funct3** | **Описание** |
| --- | --- | --- | --- | --- | --- |
| **jal** | Jump and Link | J | 1101111 | - | rd = PC+4; PC += imm |
| **jalr** | Jump and Link Register | I | 1100111 | 0x0 | rd = PC+4; PC = rs + imm |

Инструкция **jal rd, imm** добавляет непосредственное значение к программному счётчику PC, PC += imm, и пишет адрес инструкции, следующей за **jal** в регистр, задаваемый как rd. Инструкция **jalr rd, rs, imm** устанавливает программный счётчик в значение, равное результату сложения регистра, задаваемого как rs, и непосредственного значения imm, pc = rs + imm, и пишет адрес возврата в регистр, задаваемый как rd.

Рассмотрим пример:

Адрес Машинный код Значение Комментарий

0x00 0x00c000ef jal x1,0xc set pc to 0x0 + 0xc = 0xc,

x1 = pc + 4 = 0x4

0x04 0x00000013 addi x0, x0, 0 no operation

0x08 0x00010093 addi x1, x2, 0 set x1 = x2

0x0c 0x00008167 jalr x2, x1, 0 set pc to x1,

x2 = pc + 4 = 0x10

0x10 0x00100093 addi x1, x0, 1 x1 = 1

Программа работает следующим образом. Она начинается с инструкции **jal x1, 0xc**, которая расположена по адресу 0x00. В результате своего выполнения адрес возврата пишется в регистр x1 (следующая инструкция за инструкцией по адресу 0x00 размещается по адресу 0x04, именно это число помещается в регистр x1) и осуществляется переход по адресу 0x0c. По адресу 0x0c размещается инструкция **jalr x2, x1, 0**, которая пишет уже свой адрес возврата (значение 0x10) в регистр x2 и осуществляет переход по адресу, который лежит в регистре x1 (там размещается адрес возврата для первого выполненного перехода, это адрес 0x04). Программа продолжает выполняться и исполняется инструкция, расположенная по адресу 0x04, которая ничего не делает. Следующая за ней инструкция, расположенная по адресу 0x08) записывает значение 0x10 (адрес возврата для команды jalr) в регистр x1. После этого, вторая итерация выполнения команды **jalr** по адресу 0xc передаёт управление по адресу 0x10 (значение регистра x1). Итого, программный счётчик PC последовательного принимает следующие значения: 0x00, 0x0c, 0x04, 0x08, 0x0c, и 0x10.

### Системный интерфейс

Две инструкции, **ecall** и **ebreak** задаются в I-формате (I-format, immediate), и обе используются для доступа к системным функциям. Системные функции обычно требуют привилегированного доступа и наличия инструкций, которые лежат вне пользовательского режима доступа.

| **Инструкция** | **Тип** | **Формат** | **Код** | **funct3** | **Описание** |
| --- | --- | --- | --- | --- | --- |
| **ecall** | Environment Call | I | 1110011 | 0x0 | imm = 0, rd = rs1 = 0, передать управление системе |
| **ebreak** | Environment Break | I | 1110011 | 0x0 | imm = 1, rd = rs1 = 0, передать управление отладчику |

Инструкция **ecall** запрашивает системную сервисную операцию, так называемый системный вызов. Согласно бинарному интерфейсу для приложений ABI, идентификатор и дополнительные параметры предоставляются через предварительно настроенные регистры, при этом значение регистра x17 задаёт тип системного вызова. В следующем примере показан вызов функции выхода в ОС Linux. Системный вызов exit говорит ОС Linux о том, что ОС должна закончить выполнение программы и выгрузить программу из памяти.

Адрес Машинный код Значение Комментарий

0x00 0x05d00893 addi x17, x0, 93 set x17 = 93,

Linux system call exit

0x04 0x00000073 ecall call system

Инструкция **ebreak** используется отладчиком для отладки программ. Отладчик временно подменяет инструкцией **ebreak** код в том месте, где программа должна остановиться. Ситуация, когда вызов инструкции **ebreak** реализуется программистом вручную, нетипична.

### Упорядочение памяти

Наконец, инструкция **fence** используется для синхронизации доступа к памяти между различными процессорами, использующими общую область памяти. Инструкция **fence** разделяет программный код на код, идущий до неё (предшествующий набор, predecessor set) и инструкции после неё (последующий набор, successor set). Команда полезна для систем, использующих более одного процессорного ядра RISC-V.

Инструкция **fence** заботится о том, чтобы каждая инструкция из предшествующего ей набора была выполнена исполняющим ядром (и так для каждого ядра) перед тем, как будет выполнена первая инструкция из последующего за ней набора. Инструкция **fence** очень важна для реализации синхронизации процессов в операционных системах, в том числе в таких механизмах, как механизмы блокировки. Однако эти механизмы выходят за пределы рассматриваемого курса и упоминаются лишь для полноты картины. Обычная инструкция **fence** имеет следующий I-формат:

| **Инструкция** | **Тип** | **Формат** | **Код** | **funct3** | **Описание** |
| --- | --- | --- | --- | --- | --- |
| **fence** | Fence | I | 0001111 | 0x0 | rd, rs1 зарезервированы.  Для всех типов доступа к памяти imm = 0b000011111111. |

### ‘M’-расширение команд

В качестве примера расширений, рассмотрим M-расширение, реализующее операции умножения и деления. Инструкции приведены в таблице:

| **Инструкция** | **Тип** | **Формат** | **Код** | **funct3** | **funct7** | **instruction** |
| --- | --- | --- | --- | --- | --- | --- |
| **mul** | Multiply | R | 0110011 | 0x0 | 0x01 | rd = (rs1 \* rs2)[31:0] |
| **mulh** | Multiply High | R | 0110011 | 0x1 | 0x01 | rd = (rs1 \* rs2)[63:32] |
| **mulsu** | Multiply High Sign/Uns. | R | 0110011 | 0x3 | 0x01 | rd = (rs1 \* rs2)[63:32] |
| **mulu** | Multiply Unsigned | R | 0110011 | 0x3 | 0x01 | rd = (rs1 \* rs2)[63:32] |
| **div** | Divide | R | 0110011 | 0x4 | 0x01 | rd = rs1 / rs2 |
| **divu** | Divide Unsigned | R | 0110011 | 0x5 | 0x01 | rd = rs1 / rs2 |
| **rem** | Remainder | R | 0110011 | 0x6 | 0x01 | rd = rs1 % rs2 |
| **remu** | Remainder Unsigned | R | 0110011 | 0x7 | 0x01 | rd = rs1 % rs2 |

Поскольку перемножение двух (потенциально больших) чисел может превысить размер регистра (для RV32 этот размер составляет 32 бита), младшая часть результата умножения возвращается с использованием инструкции mul, а старшая часть – с использованием инструкций **mulh**, **mulsu**, **mulu** в зависимости от того, какая комбинация чисел в плане знака используется. Например:

Адрес Машинный код Значение Комментарий

0x00 0x000100b7 lui x1, 0x10 x1 = 0x00010000

0x04 0x00108093 addi x1, x1, 1 x1 = 0x00010001

0x08 0x00080137 lui x2, 0x80 x2 = 0x00080000

0x0c 0x022081b3 mul x3, x1, x2 x3 = 0x00080000,

lower 32 bits of mult.

0x10 0x02209233 mulh x4, x1, x2 x4 = 0x00000008,

higher 32 bits of mult.

Деление на нуль, результат которого не определён, записывает в регистр-приёмник значение -1 (или, другими словами, выставляет все биты в 1). Это значит, что перед выполнением операции деления программисту необходимо самостоятельно проверять делитель и использовать операторы ветвления для обработки исключительных ситуаций.

### Замечания касательно архитектуры RV64

В целом, для архитектуры RV64 набор инструкций такой же, как и для архитектуры RV32. Однако для RV64 регистры имеют размер в 64 бита, а не в 32, и операции выполняются над 64-битными числами. Более того, для RV64 реализуется ряд инструкций, специфичных именно для 64-битной архитектуры, такие инструкции имеют в конце букву 'w'. Такие инструкции работают только с 32-битными числами, причём используются младшие 32 бита то 64-битных чисел. Примерами таких инструкций являются **addiw**, **addw**, **sllw**, **mulw**.

# Глава 4. Программирование на языке Assembler

## Введение

### Обзор главы

Набор инструкций архитектуры (ISA) RISC-V, который обсуждался ранее, является основой для программирования на языке Assembler. Однако не очень удобно реализовывать программу исключительно на уровне ISA. По этой причине компилятор языка Assembler для RISC-V предоставляет различные псевдо-инструкции и директивы, которые облегчают разработку программ.

### Цели обучения

К концу этой главы вы будете:

* понимать и уметь использовать метки и директивы;
* понимать и уметь использовать псевдо-инструкции;
* знать, почему и как использовать бинарный интерфейс приложений (application binary interface, ABI);
* уметь реализовывать поток выполнения программ.

### Метки в языке Assembler

Первое, что облегчает разработку кода на языке Assmebler, это метки. Метки служат для идентификации мест в программном коде, они используются в качестве смещений и непосредственных значений адресов.

Под меткой понимается символьная последовательность, оканчивающаяся двоеточием. Также существует особая разновидность меток - числовые метки, – это локальные метки, которые могут быть заданы с суффиксами 'f' для перехода вперёд и 'b' для перехода назад.

Пример показан ниже:

1:

j 1b

j 1f

1:

Первый переход выполнится на метку 1 выше кода, потому что указан суффикс ‘b’, второй переход выполнится на метку 1 ниже кода, потому что указан суффикс ‘f’.

Компилятор языка Assembler преобразует метки в непосредственные значения адресов.

Рассмотрим следующий пример:

addi x1, x0, 1

beq x1,x0, there

addi x1, x1, 1

there:

addi x1, x1, 1

Компилятор переводит программу в машинные коды и вставляет смещения, ассоциируемые с метками:

0x0: 00100093 addi x1 x0 1

0x4: 00008463 beq x1 x0 8 <there>

0x8: 00108093 addi x1 x1 1

0000000c <there>:

0xc: 00108093 addi x1 x1 1

### Адресация с использованием меток

Если мы посмотрим на базовый набор RV32I ISA, можно увидеть, что адресация по всему адресному пространству может потребовать более одной инструкции: инструкции **auipc** и **lui** используются для загрузки старшей части непосредственного значения адреса в регистр, а для загрузки младшей части в регистр используется инструкция **addi**. Таким образом, требуется некий механизм разделения адресов меток на старшую и младшую составляющие.

Функции переразмещения **%hi(symbol)** и **%lo(symbol)** разделяют адрес метки на младшую и старшую части. Компоновщик (линковщик) размещает программу в памяти и назначает адреса соответствующим символам. Таким образом, код:

lui x1, %hi(there) #absolute higher 20 bits

addi x1, x1, %lo(there) #absolute lower 12 bits

there:

транслируется в код:

0x10078: lui x1,0x10

0x1007c: addi x1,x1,128 # 10080 <there>

Адрес метки **there** после размещения имеет адрес 0x10080. Функции **%hi** и **%lo** используются для задания корректных адресов в регистре x1.

Ещё одним примером функций переразмещения являются функции адресации, относительной значения программного счётчика. К ним относятся функции **%pcrel\_hi(symbol)** и **%pcrel\_lo(label)**, которые работают вместе с инструкциями **auipc** и **addi**. Однако, поскольку адресация относительная, они используются отличным от глобальной адресации способом. Так, компилятор сгенерирует из такого кода:

1: auipc x1, %pcrel\_hi(there)#relative higher 20 bits

addi x1, x1, %pcrel\_lo(1b)#relative lower 12 bits label 1 backwards

there:

следующий код:

1:

0x10078: auipc x1,0x0

0x1007c: addi x1,x1,8 # 10080 <there>

В этом примере также адрес метки **there** равен 0x10080. Инструкция **auipc** должна добавить 0x0 к значению программного счётчика 0x10078 для получения старшей части адреса метки ‘there’. Результат инструкции **auipc** - число 0x10078 в регистре x1. Инструкция **addi** добавляет число 8 к регистру x1, поскольку 8 – это младшая часть разницы адресов меток **there** и **1**.

Однако работа с функциями переразмещения вручную - работа громоздкая, поэтому чаще всего вместо них используют псевдоинструкции.

### Директивы языка Assembler

Директивы языка предоставляют компилятору информацию о том, как должен интерпретироваться текст, следующий за директивой: это данные или код. Все директивы начинаются с точки.

В таблице ниже приводится перечень наиболее часто употребляемых директив:

| **Директива** | **Аргументы** | **Описание** |
| --- | --- | --- |
| **.text** |  | Переключиться на секцию .text |
| **.data** |  | Переключиться на секцию .data |
| **.rodata** |  | Переключиться на секцию .rodata |
| **.bss** |  | Переключиться на секцию .bss |
| **.section** | .text, .data, .rodata, .bss | Переключиться на секцию, задаваемую аргументом |
| **.equ** | name, value | определить имя для константного значения |
| **.ascii** | “string” | начало строки без терминального нуля |
| **.asciz** | “string” | начало строки с терминальным нулём |
| **.string** | “string” | то же самое, что и .asciz |
| **.byte** | expression [,expression]\* | 8-битные значения, разделённые запятыми |
| **.half** | expression [,expression]\* | 16-битные значения, разделённые запятыми |
| **.word** | expression [,expression]\* | 32-битные значения, разделённые запятыми |
| **.dword** | expression [,expression]\* | 64-битные значения, разделённые запятыми |
| **.zero** | integer | байты, заполненные нулями |
| **.align** | integer | значение, выровненное по заданной степени 2 |
| **.globl** | symbol\_name | внести symbol\_name в таблицу символов |

Эти директивы очень важны, когда пишется программа, которая собирается для исполнения внутри некоторого окружения, например, встроенной системы или операционной системы.

Директива **.text** предоставляет компилятору информацию, что следующий текст является набором машинных инструкций. Директива **.data** говорит о том, что начинается область с объявлением данных, которые должны быть проинициализированы и могут меняться в процессе выполнения программы. Директива **.rodata** предназначена для объявления инициализированных данных, доступных только для чтения (константные значения). Директива **.bss** служит для объявления модифицируемых, но не инициализированных данных. Все эти области памяти могут быть объявлены с помощью директивы **.section** с последующим указанием типа секции (.text, .data, .rodata и т.д.).

Имя для константного значения может быть задано с помощью директивы **equ**. Данные для ASCII-строк задаются директивами **.ascii**, **.asciz**, и **.string**. Разница между ними в том, что директивы **.asciz** и **.string** добавляют терминальный нулевой байт в конце строки, а директива **.ascii** - нет. Текст располагается следом за директивной.

Директивы **.byte**, **.half**, **.word**, и **.dword** служат для задания одного или более числовых значений, которые указываются следом за директивами. Указанные директивы отличаются размером (в байтах) описываемых значений.

Директива **.zero** объявляет массив байт, которые заполняются нулями, при этом первое число, идущее за директивой**.zero** задаёт количество таких элементов.

Директива**.align** выравнивает в памяти данные, следующие за числом, задаваемым как 2 в степени аргумента параметра. Выравнивание достигается путём вставки нулевых байтов.

Директива **.globl symbol\_name** определяет идентификатор **symbol\_name**, который обычно задаётся как значение, видимое компоновщика. Символ **\_start** требуется для задания точки входа в программу. В данном руководстве мы не рассматриваем синтаксис скриптов компоновщика, просто используем скрипт по умолчанию, который поставляется вместе с набором инструментов сборки Linux Gnu toolchain. Тем не менее, скрипты компоновщика очень важны, когда идёт разработка для различных окружений или встраиваемых систем, поскольку такие скрипты могут определять расположение программы в памяти устройств.

Также используется такая конструкция, как точка (.), которая подменяет текущий адрес.

### Пример на директивы языка Assemler

Рассмотрим пример. Символ решётка '#' задаёт комментарий:

# define exit as 93

.equ exit, 93

# program code

.section .text

# export \_start for linker

.globl \_start

\_start:

li a7, exit

ecall

# data: init one word (16-bit value) with 1 and read/write

.section .data

counter:

.word 1

# rodata: constant text string

.section .rodata

text\_begin:

.asciz "Text"

text\_end:

# current address minus address of text\_begin = length of text

.byte .-text\_begin

# non initialized block with same size as the text

.section .bss

# start next part by address aligned to multiple of 2^2 = 4

.align 2

copy\_begin:

.zero text\_end-text\_begin

Если этот пример сохранить в файле с названием example.s, то его можно будет собрать в исполняемый модуль в формате ELF (Executable and Linkable Format, ELF), и проанализировать результат сборки и компоновки с помощью утилиты objdump:

riscv64-linux-gnu-as -o example.o example.s

riscv64-linux-gnu-ld -o example example.o

riscv64-linux-gnu-objdump -f -d -Mno-aliases,numeric example

example: file format elf64-littleriscv

architecture: riscv:rv64, flags 0x00000112:

EXEC\_P, HAS\_SYMS, D\_PAGED

start address 0x00000000000100e8

Disassembly of section .text:

00000000000100e8 <\_start>:

100e8: 05d00893 addi x17,x0,93

100ec: 00000073 ecall

Команда **objdump** с ключами **-f -d** показывает дизассемблированный файл (-d) с заголовком файла (-f). Опции, указанные после ключа **-Mno-aliases,numeric** говорят, что утилита objdump должна печатать только базовые инструкции и не выводить псевдоинструкции и печатать номера регистров, а не их имена в синтаксисе ABI (что будет рассмотрено позже). В листинге показан только программный код, содержащийся в секции**.text**. После указания ключа **-t** , утилита **objdump** печатает все секции исполняемого файла:

riscv64-linux-gnu-objdump -t example

example:     file format elf64-littleriscv

SYMBOL TABLE:  
00000000000100e8 l    d  .text 0000000000000000 .text  
00000000000100f0 l    d  .rodata 0000000000000000 .rodata  
00000000000110f6 l    d  .data 0000000000000000 .data  
00000000000110fc l    d  .bss 0000000000000000 .bss  
0000000000000000 l    d  .riscv.attributes 0000000000000000 .riscv.attributes  
0000000000000000 l    df \*ABS\* 0000000000000000 example.o  
000000000000005d l       \*ABS\* 0000000000000000 exit  
00000000000100f6 l       .data 0000000000000000 counter  
00000000000100f0 l       .rodata 0000000000000000 text\_begin  
00000000000100f5 l       .rodata 0000000000000000 text\_end  
00000000000110fc l       .bss 0000000000000000 copy\_begin  
00000000000118f6 g       \*ABS\* 0000000000000000 \_\_global\_pointer$  
00000000000110fa g       .data 0000000000000000 \_\_SDATA\_BEGIN\_\_  
00000000000100e8 g       .text 0000000000000000 \_start  
0000000000011108 g       .bss 0000000000000000 \_\_BSS\_END\_\_  
00000000000110fa g       .bss 0000000000000000 \_\_bss\_start  
00000000000110f6 g       .data 0000000000000000 \_\_DATA\_BEGIN\_\_  
00000000000110fa g       .data 0000000000000000 \_edata  
0000000000011108 g       .bss 0000000000000000 \_end

Секция **.section .text** начинается с адреса 0x100e8, секция **.rodata** – с адреса 0x100f0, секция **.data** – с адреса 0x110f6, и секция **.bss** – с адреса 0x110fc. Каждой метке присваиваются адреса, то есть метка **copy\_begin** располагается в секции **.bss** по адресу 0x110fc.

В итоге можно просмотреть содержимое бинарного файла в шестнадцатеричном формате, задав утилите **objdump** ключи **-F -s**, где ключ -F нужен для вывода дополнительной информации о файле:

riscv64-linux-gnu-objdump -F -s example

example: file format elf64-littleriscv

Contents of section .text: (Starting at file offset: 0xe8)

100e8 9308d005 73000000 ....s...

Contents of section .rodata: (Starting at file offset: 0xf0)

100f0 54657874 0005 Text..

Contents of section .data: (Starting at file offset: 0xf6)

110f6 01000000 ....

Contents of section .riscv.attributes: (Starting at file offset: 0xfa)

0000 412d0000 00726973 63760001 23000000 A-...riscv..#...

0010 05727636 34693270 305f6d32 70305f61 .rv64i2p0\_m2p0\_a

0020 3270305f 66327030 5f643270 3000 2p0\_f2p0\_d2p0.

Первые два двойных слова в секции **.text** являются представлением машинного кода программы. Следует обратить внимание, что в выводе дизассемблера порядок байт инвертирован, то есть написано 9308d005 вместо 05d00893. Показанное содержимое отображает то, как данные хранятся в памяти (или в файле), а не то, как 32-битное значение или инструкция составлены в формате RISC-V little endian. В формате представления little endian 32-битная инструкция составляется таким образом, что младшие адреса занимают старшие байты, а старшие адреса – младшие байты. Таким образом, четыре байта со значениями 0x93 0x08 0xd0 0x05 , расположенные по адресу 0x100e8, берутся и размещаются в 32-битное представление как 0x05d00893. Напротив, формат Big endian предписывает иной порядок следования байт в памяти.

Секция **.rodata** начинается после двух 32-битных инструкций по адресу 0x100f0 (что соответствует 8-ми байтному смещению от начала программы: 0x100e8 + 0x8) и содержит байты 0x54657874 0x0005. Значения 0x54 0x65 0x78 0x74 являются побайтовым представлением в таблице ASCII-символов четырёх символов: 'T' 'e' 'x' 't'. Они заканчиваются нулевым байтом (0x00) и байтом длины, равным 0x05 – это длина текста включая терминальный ноль. Затем следует секция **.data**, которая содержит 32-битное представление числа 1, которое кодируется в формате Little Endian. Следующая секция описывает атрибуты RISC-V.

Секция **.bss** не содержит данных, поэтому её содержимое не отображается. Тем не менее в таблице символов было видно, что секция **.bss** начинается с адреса 0x110fc, который является следующим адресом за адресом секции **.data** (заканчивается по адресу 0x110fa) с добавлением двух байт для выравнивания в памяти до 4 байт.

В итоге мы получили представление о том, как компилятор и компоновщик располагают данные в формате ELF.

### Псевдо-инструкции

Компилятор языка Assembler транслирует программу в машинный код. В рамках курса в основном рассматривается GNU assembler. Компилятор понимает не только инструкции RISC-V, определяемые стандартом ISA, но также понимает псевдо-инструкции и псевдо-директивы.

Псевдо-инструкции являются частью языка Assembler и могут быть странслированы в машинный код. Однако, в отличие от инструкций из набора ISA, псевдо-инструкции гораздо проще использовать. Обычно псевдо-инструкция может быть странслирована в более чем одну базовую инструкцию набора ISA. Кроме того, псевдо-инструкции решают проблему использования функций ручного перемещения памяти. В следующих таблицах показаны псевдо-инструкции и то, в какие базовые инструкции они преобразуются.

### Псевдо-инструкции для загрузки, сохранения и дополнения

| **Псевдо-инструкция** | **Набор базовых инструкций** | **Описание** |
| --- | --- | --- |
| **la rd, symbol** | **auipc rd, symbol[31:12]** **addi rd, symbol[11:0]** | Загрузить адрес (не позиционно-независимый код - non-PIC) |
| **la rd, symbol** | **auipc rd, symbol@GOT[31:12]** **l{w|d} rd, symbol[11:0](rd)** | Загрузить адрес (позиционно-независимый код - PIC) |
| **lla ra, symbol** | **auipc rd, symbol[31:12] addi rd, rd, symbol[11:0]** | Загрузить локальный адрес |
| **lga rd, symbol** | **auipc rd, symbol@GOT[31:12]** **l{w|d} rd, symbol@GOT[11:0](rd)** | Загрузить глобальный адрес |
| **l{b|h|w|d} rd, symbol** | **auipc rd, symbol[31:12]** **l{b|h|w|d} rd, symbol[11:0](rd)** | Загрузить глобальное значение |
| **s{b|h|w|d} rs, symbol, rd** | **auipc rd, symbol[31:12]** **s{b|h|w|d} rs, symbol[11:0](rd)** | Сохранить глобальное значение |
| **nop** | **addi x0, x0, 0** | Нет операции |
| **li rd, imm** | **Различные варианты инструкций** | Загрузить непосредственное значение |
| **mv rd, rs** | **addi rd, rs, 0** | Скопировать значение регистра |
| **not rd, rs** | **xori rd, rs, -1** | дополнение до 1  (инверсия битов) |
| **neg rd, rs** | **sub rd, x0, rs** | дополнение до 2  (инверсия битов с добавлением 1) |
| **negw rd, rs** | **subw rd, x0, rs** | дополнение до 2 (слово) |

Положение позиционно-независимого кода (Position Independent Code, PIC) в адресном пространстве зависит только от настроек компоновщика (а не от архитектуры) и полезно для генерации бинарного кода библиотек, которые могут использоваться другими программами. Такой код обычно используется совместно с глобальной таблицей смещений (Global Offset Table, GOT), которая хранится в исполняемом модуле и позволяет операционной системе загружать библиотеки при старте вызывающих программ по различным адресам памяти. Тема, связанная с этим вопросом, является достаточно сложной и не будет рассмотрена в рамках данного курса.

Для самостоятельно изучения вопросов, связанных с форматом ELF, можно посоветовать ознакомиться с документом RISC-V ELF Specification, доступным по адресу <https://github.com/riscv-non-isa/riscv-elf-psabi-doc/blob/master/riscv-elf.adoc>.

Отдельно стоит отметить, что псевдо-инструкция **lga** не поддерживается в наборе инструментов GNU toolchain версии 2.39 и ниже.

Рассмотрим на примере следующего кода, как он транслируется с точки зрения компилятора языка Assembler:

.text

.globl \_start

\_start:

la x1, counter #load address of counter

addi x1, x1, 4 #go to next word address += 4

lw x2, 0(x1) #load value from address, the 2

li x1, 2

lw x3, counter #load word from address counter, the 0

add x3, x2, x1 #add: x3 = x2 + x1

sw x3, counter, x2 #save x3, use x2 for address

.data

counter:

.word 0, 2

Disassembly of section .text:

00000000000100e8 <\_start>:

100e8: 00001097 auipc x1,0x1

100ec: 02808093 addi x1,x1,40 # 11110 <\_\_DATA\_BEGIN\_\_>

100f0: 00408093 addi x1,x1,4

100f4: 0000a103 lw x2,0(x1)

100f8: 00200093 addi x1,x0,2

100fc: 00001197 auipc x3,0x1

10100: 0141a183 lw x3,20(x3) # 11110 <\_\_DATA\_BEGIN\_\_>

10104: 001101b3 add x3,x2,x1

10108: 00001117 auipc x2,0x1

1010c: 00312423 sw x3,8(x2) # 11110 <\_\_DATA\_BEGIN\_\_>

Адрес метки **counter** в памяти равен 0b11110. Псевдо-инструкции транслируются в базовые инструкции следующим образом:

la x1, counter -> auipc x1,0x1; addi x1, x1, 40

li x1, 2 -> addi x1, x0, 2

lw x3, counter -> auipc x3, 0x1; lw x3, 20(x3)

sw x3, counter, x2 -> auipc x2, 0x1; sw x3, 8(x2)

### Псевдо-инструкции для условной установки битов

Таблица ниже показывает псевдо-инструкции и базовые инструкции для расширения и условной установки битов:

| **Псевдо-инструкция** | Набор базовых инструкций | Описание |
| --- | --- | --- |
| **sext.{b|h|w} rd, rs** | **Может быть различный набор инструкций** | sign extend |
| **zext.{b|h|w} rd, rs** | **Может быть различный набор инструкций** | zero extend |
| **seqz rd, rs** | **sltiu rd, rs, 1** | rd = (rs == 0)? 1:0 |
| **snez rd, rs** | **sltu rd, x0, rs** | rd = (rs != 0)? 1:0 |
| **sltz rd, rs** | **slt rd, rs, x0** | rd = (rs < 0)? 1:0 |
| **sgtz rd, rs** | **slt rd, x0, rs** | rd = (rs > 0)? 1:0 |

### Псевдо-инструкции для условного ветвления

В таблице ниже приведены псевдо-инструкции для условного ветвления. Первая часть из них упрощает ветвление с использованием операций сравнения с нулём без использования регистра x0. Вторая часть предлагает удобные функции для сравнения за счёт переупорядочивания параметров базовых инструкций, например, вызов команды **ble rs, rt, imm** эквивалентен вызову **bge rt, rs, imm**.

| **Псевдо-инструкция** | Набор базовых инструкций | Описание |
| --- | --- | --- |
| **beqz rs, imm** | **beq rs, x0, imm** | if (rs == 0) PC+=imm |
| **bnez rs, imm** | **bne rs, x0, imm** | if (rs != 0) PC+=imm |
| **blez rs, imm** | **bge x0, rs, imm** | if (rs <= 0) PC+=imm |
| **bgez rs, imm** | **bge rs, x0, imm** | if (rs >= 0) PC+=imm |
| **bltz rs, imm** | **blt rs, x0, imm** | if (rs < 0) PC+=imm |
| **bgtz rs, imm** | **blt x0, rs, imm** | if (rs > 0) PC+=imm |
| **bgt rs, rt, imm** | **blt rt, rs, imm** | if (rs > rt) PC+=imm |
| **ble rs, rt, imm** | **bge rt, rs, imm** | if (rs <= rt) PC+=imm |
| **bgtu rs, rt, imm** | **bltu rt, rs, imm** | if (rs > rt) PC+=imm, unsign. |
| **bleu rs, rt, imm** | **bgeu rt, rs, imm** | if (rs <= rt) PC+=imm, unsign. |

### Псевдо-инструкции для безусловного перехода

Наконец, псевдо-инструкции для безусловного перехода упрощают программирование за счёт того, что в их составе не нужно указывать имена регистров. А переход на дальние адреса реализуется командами **call** и **tail**.

| **Псевдо-инструкция** | Набор базовых инструкций | Описание |
| --- | --- | --- |
| **j imm** | **jal x0, imm** | PC += imm |
| **jal imm** | **jal x1, imm** | x1 = PC+4; PC += imm |
| **jr rs** | **jalr x0, rs, 0** | PC = rs |
| **jalr rs** | **jalr x1, rs, 0** | x1 = PC+4; PC = rs |
| **ret** | **jalr x0, x1, 0** | PC = x1 |
| **call imm** | **auipc x6, imm[31:12]** **jalr x1, x6, imm[11:0]** | x1 = PC+4; PC = imm |
| **tail imm** | **auipc x6, imm[31:12]** **jalr x0, x6, imm[11:0]** | PC = imm |

### Бинарный интерфейс для приложений (ABI) и пользовательский режим

До сих пор обращение к регистрам происходило по их именам. Однако, существует бинарный интерфейс для приложений (Application Binary Interface, ABI), подробнее про него можно почитать по ссылке: <https://github.com/riscv-non-isa/riscv-elf-psabi-doc>. Он представляет собой ряд правил, описывающих, как и какие регистры должны быть использованы для решения задач. Среди прочего, ABI предоставляет псевдонимы в том числе и для регистров.

Кроме того, одним из соглашений, регламентируемых ABI, является то, как устроен механизм сохранения значений регистров при переходе в подпрограмму: в момент вызова адрес возврата сохраняется в регистре, а программный счетчик устанавливается на начало функции. Функции состоит из инструкций, которые, собственно, и выполняются функцией; после выполнения функции управление передаётся инструкции, следующей следом за инструкцией вызова. Чтобы вернуться, функция использует сохранённый адрес возврата, а значит должно быть соглашение о том, какой из регистров используется для хранения адреса возврата в случае, если над программой работает большое число программистов. Кроме того, важно соглашение о том, какие регистры можно считать измененными или неизменяемыми при выходе из функции. В таблице ниже приведены псевдонимы и разъяснение соглашения о хранении. Например, регистр x1 имеет псевдоним ra, который обозначает адрес возврата, и вызывающая функция должна сохранить значение регистра x1 до вызова функции и восстановить его после возврата функции.

| Регистр | Псевдоним ABI | Описание | Кто сохраняет значение |
| --- | --- | --- | --- |
| x0 | zero | Константный ноль  zero constant | - |
| x1 | ra | Адрес возврата  return address | Вызывающий код |
| x2 | sp | Указатель на стек  stack pointer | Вызываемая функция |
| x3 | gp | Глобальный указатель  global pointer | Не должен использоваться в пользовательском режиме |
| x4 | tp | Указатель потока  thread pointer | Не должен использоваться в пользовательском режиме |
| x5-x7 | t0-t2 | Временные значения  temporaries | Вызывающий код |
| x8 | s0 / fp | Сохраняемый регистр / Указатель на фрейм  saved / Frame pointer | Вызываемая функция |
| x9 | s1 | Сохраняемый регистр  saved register | Вызываемая функция |
| x10-x11 | a0-a1 | Аргументы функций / Возвращаемые значения  function args. / return values | Вызывающий код |
| x12-x17 | a2-a7 | Аргументы функций  function arguments | Вызывающий код |
| x18-x27 | s2-s11 | Сохраняемые регистры  saved registers | Вызываемая функция |
| x28-x31 | t3-t6 | Временные значения  temporaries | Вызывающий код |
| pc | - | Программный счётчик  program counter | - |

Указатель на стек, глобальный указатель и указатель потока используются на системном уровне. Глобальный указатель и указатель потока не должны быть использованы никем, кроме операционной системы, а указатель на стек может быть использован для сохранения и восстановления значения.

В данном курсе мы рассматриваются только непривилегированные инструкции ISA. Архитектура RISC-V предусматривает три уровня привилегий: пользовательский (User, U), супервизора (Supervisor, S) и аппаратный (Machine, M). Системные вызовы, такие как, например, функции API операционной системы, выполняются либо уровне супервизора, либо на аппаратном уровне. Для использования функций операционной системы в пользовательском режиме необходимо выполнить системные вызовы строго придерживаясь ABI.

### Стек

Стек – это область памяти, к которой можно обращаться посредством указателя на стек (Stack Pointer, регистр sp). Указатель на стек обычно управляется системой, так что регистр sp на момент запуска программы уже является проинициализированным. Стек в памяти растёт от старших адресов к младшим и заполняется данными сверху вниз.

Для сохранения данных на стеке используется термин «затолкнуть в стек» (push), а для извлечения - «вытолкнуть из стека» (pop).

В примере ниже представим, что регистр sp хранит значение 0xff0:

li t0, 0xbeabdeaf # t0 = 0xbeabdeaf

addi sp, sp, -4 # увеличить стек на 4 байта

sw t0, 0(sp) # затолкнуть t0 (0xbeabdeaf) в стек

В результате выполнения этой операции указатель на стек хранит значение 0xfec, и значение 0xbeabdeaf сохраняется в памяти по адресу 0xfec. Значение можно извлечь и восстановить значение указателя на стек следующим кодом:

lw t0, 0(sp) # вытолкнуть 0xbeabdeaf из стека в t0

addi sp, sp, 4 # урезать значение стека обратно

### Реализация контроля за ходом выполнения программ

Программа начинает выполнение с инструкции, определяемой программным счётчиком. Программный счётчик определяет процесс выполнения программы. Без использования инструкций ветвления и переходов, программа будет исполняться инструкция за инструкцией в том порядке, в котором они расположены в памяти. Использование инструкций ветвления и переходов позволяет контролировать процесс выполнения программ, они позволяют реализовывать операции проверки условий и циклы, которые являются основными действиями в императивном программировании. Использование безусловных переходов (совместно с механизмом сохранения адреса возврата согласно ABI) позволяет использовать функции (процедуры), которые являются основой процедурного программирования. Поскольку эти функциональности определяют то, как мы можем писать код, рассмотрим их реализацию с соответствующими инструкциями языка Assembler.

### Условия

Цель условия – проверить, является ли проверяемое выражение истинным (true) или ложным (false), и, в зависимости от получаемого результата, выполнить переход по одной из веток кода. Такой механизм также известен как оператор if-then-else.

Реализация if-then-else в языке Assembler реализуется с помощью инструкций условного ветвления. Если условие (не) верно, управление передаётся на соответствующую ветку кода. Зачастую логика оригинального оператора if-then-else переворачивается таким образом, что код веки «else» выполняется до кода ветки «then», как показано в примере:

# если условие истинно, тогда выполнить ‘code for then’, иначе ‘code for else’

# если (t0 == t1), то переходим на ветку ‘then’

beq t0, t1, then

# иначе

... # код для else

# перепрыгнуть через then-часть

j end

then:

... # код для then

end:

Простой вариант конструкции if-then может быть реализован с использованием одной инструкции ветвления:

# если t0=t1 - выполнить ‘code for then’, если t0!=t1 – игнорировать ‘code for then’

bne t0, t1, skip

... # код для then

skip:

...

Более сложные условия реализуются вложенными конструкциями.

### Циклы

Циклы также реализуются условными инструкциями. Конструкция while-do повторяет do-часть зацикленно до тех пор, пока проверяемое условие истинно. Для конструкции while-do условие проверяется в начале инструкций, которые необходимо повторить. Если условие не выполняется, зацикливаемый код пропускается (не выполняется). Например:

# инициализация t0 и t1

# цикл должен выполниться 4 раза

li t0, 0

li t1, 4

while:

# если t0 == t1 перейти на end, выйти из цикла

beq t0, t1, end

... # код цикла

# инкремент: t0 = t0 + 1

addi t0, t0, 1

# повторять, пока условие выше (t0 == t1) - истиина

j while

end:

### Функции

Любая функция реализуется путём сохранения адреса возврата, перехода на код функции, его выполнения и возврата из функции, при этом необходимо позаботиться о том, чтобы использовались корректные регистры для сохранения адреса возврата. Лучшее решение – слепо следовать ABI, которые регламентирует использование регистров как для передачи аргументов, так и для реализации механизма возврата:

# аргумент в a0

li a0, 0

# вызов функции func, адрес возврата в ra

jal ra, func

# код, выполняемый после возврата из func

# ...

# код функции

func:

... # в коде функции нельзя менять ra

# возращаемое значение в a0, в примере – число 1

li a0, 1

# вернуться по адресу из ra

ret

Инструкции, которые сохраняют значения регистров на стеке в начале кода функции называют «пролог»; инструкции, которые выталкивают значения из стека обратно в регистры в конце работы функции называют «эпилог». Пролог и эпилог служат для реализации возможности работы с регистрами внутри тела функции. Простой пример пролога и эпилога показан в следующем разделе в примере использования рекурсии.

### Рекурсия

Рекурсивная функция – функция, которая вызывает сама себя. Рекурсивные функции обычно используют локальные переменные, значения которых хранятся в регистрах и которые необходимо сохранять перед вызовом функции и восстанавливать после возврата из неё. Рекурсивные функции – источник большого числа ошибок, поскольку при вызове рекурсивных функций происходит постоянное заталкивание значений регистров в стек и выталкивание значений регистров из стека и при отсутствии контроля может произойти переполнение стека в случае, если размера выделенной под стек памяти окажется недостаточно.

Рассмотрим вычисление n-ного элемента последовательности a(n) = a(n-1) + 3 при начальном значении a(0) = 2. Члены последовательности: a(0) = 2, a(1) = a(0) + 3 = 5, a(2) = a(1) + 3 = (a(0)+3) + 3 = 8 и так далее. Вычисляющая функция вызывает сама себя до тех пор, пока не будет достигнуто начальное значение a(0). Ниже приводится пример программы с рекурсивным вызовом функции **compute**, вычисляющей a(n) до тех пор, при этом считается, что аргумент n должен храниться в регистре a0. Результат возвращается в регистре a0.

.globl \_start

\_start:

li a0, 5 # вычислить для n = 5

call compute

# выход

li a7, 93

ecall

compute:

# занять место в стеке под регистр ra

# в RV64 регистры 64 бита, следовательно 8 байт

addi sp, sp, -8

sd ra, 0(sp)

# проверка на окончание рекурсии; да? тогда выходим

beq a0, x0, compend

# иначе посчитать a(n-1)

addi a0, a0, -1

# рекурсивный вызов

call compute

# рекурсивно посчитать a(n) = a(n-1) + 3

addi a0, a0, 3

# выйти

j compret

compend:

# если (n ==0), вернуть a(0) = 2

li a0, 2

compret:

# восстановить ra

ld ra, 0(sp)

# осводобить стек

addi sp, sp, 8

ret

# Глава 5. Использование системных библиотек

## Введение

### Обзор главы

В этой главе приводятся некоторые примеры того, как писать программы под ОС Linux и использованием взаимодействия с кодом на языке Assembler. Рассмотренные действия выполнятся на компьютере с Intel-архитектурой с установленной ОС Linux и установленным набором инструментов для кросс-компиляции. Исполняемые файлы запускаются под эмулятором Qemu в режиме пользователя. Несмотря на это, программы могут быть собраны и запущены на процессоре с архитектурой RISC-V как это было показано в главе 2.

### Цели обучения

После изучения этой главы вы должны быть способны:

* писать программы на языке Assembler с использованием системных вызовов;
* писать программы на языке Assembler с использованием библиотек, написанных на C.

## Использование системы и библиотек

### Взаимодействие с системой

Операционная система предоставляет интерфейс для использования её ресурсов и сервисов путём выполнения системных вызовов. О том, чтобы получаемый в результате компиляции и линковки бинарный код соответствовал целевому окружению, заботится скрипт компоновщика.

ОС Linux предоставляет интерфейс для системных вызовов. Стоит отметить, что несмотря на то, что приведённые примеры были выполнены и проверены в ОС Debian, они должны работать и в других версия ОС Linux.

Введите следующую команду:

man syscall.2

И посмотрите на строки и столбцы с текстом «riscv»:

Arch/ABI Instruction System Ret Ret Error Notes

call # val val2

-------------------------------------------------------------------

riscv ecall a7 a0 a1 -

...

Arch/ABI arg1 arg2 arg3 arg4 arg5 arg6 arg7 Notes

--------------------------------------------------------------

riscv a0 a1 a2 a3 a4 a5 -

Видно, что регистр a7 используется для задания номера системного вызова, регистры a0 и a1 используются для хранения возвращаемых значений, а аргументы передаются через регистры с a0 по a5.

Есть документация на различные системные вызовы:

man syscalls.2

NAME

syscalls - Linux system calls

SYNOPSIS

Linux system calls.

DESCRIPTION

The system call is the fundamental interface between an application and the Linux kernel.

...

System call Kernel Notes

------------------------------------------------------------------------

write(2) 1.0

...

В качестве примера рассмотрим системный вызов для вывода значений **write**. Чтобы получить информацию о параметрах системного вызова **write**, необходимо ввести в консоли:

man write.2

NAME

write - write to a file descriptor

SYNOPSIS

#include <unistd.h>

ssize\_t write(int fd, const void \*buf, size\_t count);

DESCRIPTION

write() writes up to count bytes from the buffer starting at buf to the file referred to by the file descriptor fd.

...

Системный вызов **write** позволяет выводить информацию в поток стандартного вывода. Файловый дескриптор со значением 1 является стандартным выводом, что, во-первых, является общим соглашением, а во-вторых, описано в заголовочном файле /usr/include/unistd.h.

Но эта информация – для программистов на языке C. Для программистов на языке Assembler необходимо знать численное значение номера системного вызова. В файле /usr/include/asm-generic/unistd.h содержится следующая строчка, задающая идентификатор для системного вызова:

grep write /usr/include/asm-generic/unistd.h

/\* fs/read\_write.c \*/

#define \_\_NR\_write 64

\_\_SYSCALL(\_\_NR\_write, sys\_write)

…

Кроме того, необходимо оформить правильный выход из программы. Это можно сделать с использованием системного вызова **exit**. Системный вызов **exit** имеет номер 93, что можно выяснить способом, аналогичным показанному выше.

Обладая этой информацией, теперь есть возможность написать программу, которая выводит традиционное сообщение «Hello World!». Для этого нам нужно загрузить регистры необходимыми параметрами и выполнить корректные системные вызовы. Параметрами системного вызова **write** являются файловый дескриптор, который необходимо поместить в регистр a0, адрес начала выводимого сообщения, который необходимо поместить в регистр a1 и длина сообщения в байтах, которую нужно поместить в регистр a2. Номер системного вызова необходимо помещать в регистр a7, а сам системный вызов осуществляется с помощью инструкции **ecall**:

# file hello.s

.equ write, 64

.equ exit, 93

.section .text

.globl \_start

\_start:

li a0, 1

la a1, msgbegin

lbu a2, msgsize

li a7, write

ecall

li a0, 0

li a7, exit

ecall

.section .rodata

msgbegin:

.ascii "Hello World!\n"

msgsize:

.byte .-msgbegin

Соберём и запустим программу:

riscv64-linux-gnu-as hello.s -o hello.o

riscv64-linux-gnu-ld hello.o -o hello

./hello

Hello World!

### Взаимодействие с библиотеками

Помимо прямых системных вызовов, операционная система предоставляет целый набор библиотек, которые могут быть выполнены в двух вариантах: для динамической или статической компоновки. Используя связывание (линковку) с библиотеками, можно в своих программах использовать предоставляемый ими набор функций. Интерфейс обычно представляет собой описание функции для языка C, доступ к нему можно получить, путём линковки в окружении компилятора C.

Построение интерфейса с библиотеками на языке С требует соблюдения соглашений о вызовах RISC-V (RISC-V Calling Conventions, документ доступен по ссылке <https://github.com/riscv-non-isa/riscv-elf-psabi-doc/blob/master/riscv-cc.adoc>). Имеется возможность указать компилятору/компоновщику путём задания соответствующих аргументов, какой вариант соглашения ABI нужно использовать. Например, вариантом ABI по умолчанию для архитектуры RV64G является LP64D. ABI регламентирует размер типов данных, использованных компилятором, порядок передачи аргументов в функции, и т.д.

Компоновщик в процессе работы добавляет специальный служебный настроечный код для работы с библиотеками С, который запускается до разработанной программы на языке Assembler и затем передаёт управление на точку входа программы. В предыдущих примерах точкой входа был участок кода с меткой \_start, поскольку до сих пор сборка в окружении языка C не использовалась, однако при сборке в окружении языка C необходимо, чтобы имя метки было \_main.

В следующем примере программы на языке Assembler будут использованы функции printf и scanf стандартной библиотеки glibc, получить информацию по использованию которых можно в стандартное документации man printf.3 и man scanf.3. Демонстрируемая далее программа считает хэш по алгоритму djb2 (подробнее про алгоритм можно почитать по ссылке: http://www.cse.yorku.ca/~oz/hash.html) за авторством Д. Бернштейна. Хэш-функция вычисляет значение фиксированной длины для задаваемого в строковом виде набора данных. Алгоритм djb2 начинается с значения hash(0) = 5381 и вычисляет каждое следующее значение по формуле:

hash(i+1) = 33 \* hash(i) ^ character(i),

где character(i) соответствует символу на i-той позиции во входной строке. Вычисления начинаются с самого первого символа и заканчиваются на последнем символе.

Следует обратить внимание, что программа написана для набора инструкций RV64I, поэтому для компиляции для набора инструкций RV32 вызовы **addiw** и **slliw** должны быть заменены на вызовы **addi** и **slli**. Оригинальная хэш-функция оперирует 32-битными числами (как и в приведённом ниже примере), однако алгоритм применим и для 64-битных чисел.

# djb2.s

.section .text

.globl main # run in C 'environment'

main:

addi sp, sp, -8 # store ra (return address) on stack

sd ra, 0(sp)

la a0, prompt # printf the prompt string

call printf

la a0, scanfmt # scanf from stdin (console)

la a1, input # into buffer input

call scanf # with format scanfmt

la a0, input # process input with djb2

call djb2

mv a1, a0

la a0, result # print result

call printf

li a0, 0

ld ra, 0(sp) # restore ra

addi sp, sp,8

ret # return to caller

djb2: # compute djb2

li t1, 5381 # init hash = 5381

djb2\_loop:

lb t0, 0(a0) # process every char of input

beqz t0, dbj2\_end # until zero appears

mv t2, t1

slliw t2, t2, 5 # t2 = hash << 5 = 32 \* hash

addw t1, t1, t2 # t1 = 32 \* hash + hash = 33 \* hash

addw t1, t1, t0 # t1 = 33 \* hash + char

addi a0, a0, 1 # next iteration

j djb2\_loop

dbj2\_end:

mv a0, t1 # return hash value

ret

.section .rodata

prompt:

.asciz "Enter text: "

scanfmt:

.asciz "%127[^\n]" # scanf max 127 chars and end with return

result:

.asciz "Hash is %lu\n" # write out the parameter as long unsigned

.section .bss

input: # storage for input

.zero 128

Файл djb2.s необходимо слинковать компилятором C, а собранный в результате бинарный исполняемый файл должна обладать информацией о том, где располагаются динамические библиотеки. Этого можно достичь, задействовав переменную окружения QEMU\_LD\_PREFIX как показано ниже:

riscv64-linux-gnu-as djb2.s -o djb2.o

riscv64-linux-gnu-gcc djb2.o -o djb2

export QEMU\_LD\_PREFIX=/usr/riscv64-linux-gnu/

qemu-riscv64-static djb2

Enter text: hallo

Hash is 261095189

Можно просмотреть исполняемый файл с помощью утилиты objdump и увидеть, что исполняемый файл отличается от предыдущих примеров за счёт настройки среды окружения C.

### Подсчёт символов

Следующая программа подсчитывает количество символов, слов и строк, задаваемых в стандартном вводе. В ней используется функция getchar библиотеки C, которая передаёт ASCII-коды вводимых символов. При подсчете слов предполагается, что слово представляет собой по крайней мере букву алфавита или десятичную цифру. Код перевода строки в Linux имеет ASCII-код 0xa, ASCII-коды десятичных цифр лежат в диапазоне от 0x30 до 0x39 для цифр от 0 до 9, буквы латинского алфавита в верхнем регистре имеют коды от 0x41 до 0x5a для букв от A до Z, и коды от 0x61 до 0x7a для букв нижнего регистра от a до z. Для детального анализа некоторых инструкций в приведённом исходном коде заданы номера строк.

00 # wordcount.s

01 .section .text

02 .globl main # run in C 'environment'

03 main:

04 addi sp, sp, -40 # store ra (return address) and saved regs on stack

05 sd ra, 0(sp)

06 sd s0, 8(sp)

07 sd s1, 16(sp)

08 sd s2, 24(sp)

09 sd s3, 32(sp)

10

11 li s0, 0 # counter chars

12 li s1, 0 # counter line feeds

13 li s2, 0 # counter words

14 li s3, 0 # indicator if current input is in word

15

16 loop:

17 call getchar # get input from stdin in a0

18 bltz a0, end # if end of file (eof is -1) goto end

19

20 addi s0, s0, 1 # count characters

21

22 li t0, 0xa # is linefeed (ascii 0xa)?

23 bne a0, t0, nolf # no -> continue

24 addi s1, s1, 1 # yes -> count

25 nolf:

26

27 # is this a word: char digit or alphabet?

28 addi t0, a0, -0x30 # digits go from 0x30 to 0x39

29 li t1, 0x9 # if (char-0x30) >= 0 and <= 0x9 then digit

30 bleu t0, t1, aldi # trick: treat negative value as unsigned

31 # value (or neg. as unsign.) > 0x9, continue

32 andi t0, a0, ~0x20 # 0x60 to range > 0x40, lower to upper cast

33 addi t0, t0, -0x41 # letter go from 0x41 to 0x5a

34 li t1, 0x19 # (char-0x41) >= 0 and <= 0x19, then alphabet

35 bleu t0, t1, aldi # trick again

36 # reached here, then not in word (anymore)

37 add s2, s2, s3 # count word, indicator is one if word else 0

38 li s3, 0 # clear indicator

39 j loop

40

41 aldi:

42 li s3, 1 # char is part of word, indicate for word counter

43 j loop

44 end:

45 la a0, result # print result

46 mv a1, s1

47 mv a2, s2

48 mv a3, s0

49 call printf

50

51 li a0, 0

52

53 ld s3, 32(sp) # restore saved regs.

54 ld s2, 24(sp)

55 ld s1, 16(sp)

56 ld s0, 8(sp)

57 ld ra, 0(sp) # restore ra

58 addi sp, sp,40

59 ret # return to caller

60

61 .section .rodata

62 result:

63 .asciz "Lines: %u Words: %u Chars: %u\n" # write out result

Инструкции на строках 04-09 и 53-58 сохраняют значения регистров ra и s0-s3 согласно ABI, регистры s0-s2 используются в роли счётчиков для подсчёта символов, слов и строк соответственно. Регистр s3 принимает значение 1, если входной символ представляет собой букву слова. Все эти регистры изначально инициализируются нулями в строках 11-14.

В цикле со строки 16 вызывается функция getchar. Если в результате вызова getchar было получено значение -1 (признак конца файла или конца водимых данных), происходит выход из цикла (строка 18). Если getchar вернула ASCII-код символа, увеличивается значение счётчика символов (строка 20). Если getchar вернула символ перевода строки, увеличивается счётчик строк (строки 22-24). Считанный символ проверяется на то, что он является числом (строки 27-30) или буквой (строки 32-36) и в случае, если он попадает в один из этих диапазонов, значение регистра s3 выставляется равным 1 (наличие 1 в регистре s3 говорит о том, что обнаружено слово) и осуществляется переход на новую итерацию цикла считывания (строки 42-43). Если же прочитанный символ не входит в требуемые диапазоны, значение регистра r3 добавляется к счётчику слов, и его значение становится равным 0.

Строки 28-30 и 32-35 определяют, находится ли символ в заданном диапазоне. Первая проверка – на попадание в диапазон от 0x30 до 0x39 (цифры), это достигается путём вычитания из кода символа значения 0x30 и проверки на то, что результат лежит в диапазоне от 0 до 9. При этом выполняется операция если-меньше-или-равно (bleu). Строки 33-35 ведут себя похожим образом для диапазона от 0x41 до 0x5a за исключением того, что в строке 32 очищается бит 5 рассматриваемого символа инструкцией andi с инверсией значения 0x20. Очищая этот бит, все значения, большие или равные 0x60, отображаются в диапазон от значения 0x40, а значения ниже 0x60 не затрагиваются. Это позволяет выполнить проверку в строках 33-35 также и для значений в диапазоне от 0x61 до 0x7a.

riscv64-linux-gnu-as -o wordcount.o wordcount.s

riscv64-linux-gnu-gcc -o wordcount wordcount.o

export QEMU\_LD\_PREFIX=/usr/riscv64-linux-gnu/

qemu-riscv64-static wordcount

word Word .... hello

next line should be two lines

Lines: 2 Words: 9 Chars: 60

### Проверка чисел на простоту

Следующий код запрашивает число и проверяет его на простоту, результат этой проверки выводится на экран. Программа реализует алгоритм «решето Эратосфена». C-функция scanf принимает аргумент – строку «%u», которая читает беззнаковое целое. Метки, начинающиеся с .L, являются локальными и не экспортируются в таблицу символов.

00 # prime.s

01 .equ maxnb, 0x100000

02 .section .text

03 .globl main # run in C 'environment'

04 main:

05 addi sp, sp, -8 # store ra (return address) on stack

06 sd ra, 0(sp)

07

08 la a0, prompt # printf the prompt string

09 call printf

10

11 la a0, scanfmt # scanf from stdin (console)

12 la a1, input # into buffer input

13 call scanf # with format scanfmt

14

15 blez a0, .Lerr # input error

16

17 la a1, input # check if input number n fits

18 lw a1, 0(a1)

19 li t0, maxnb

20 bge a1, t0, .Lerr

21

22 la a0, input # process input with sieve

23 call sieve

24

25 bnez a0, .Lp1

26 .Lp0:

27 la a0, outno

28 j .Lpp

29 .Lp1:

30 la a0, outyes # print result

31 j .Lpp

32 .Lerr:

33 la a0, error

34 .Lpp:

35 call printf

36

37 li a0, 0

38

39 ld ra, 0(sp) # restore ra

40 addi sp, sp,8

41 ret # return to caller

42

43 sieve:

44 # input: register a0 points to number n

45 # that is checked if it is a prime.

46 # output: if n is prime a0 is one else zero

47 # sieve of Erastosthenes

48 # init array with numbers

49 lw t1, 0(a0) # nb to check

50 li t2, 2 # counter start with 2

51 la t3, array # pointer to array

52 .Ls0:

53 sw t2, 8(t3) # set item to index, 8() is begin with index 2

54 addi t3, t3, 4 # increment by four for word size

55 addi t2, t2, 1 # counter

56 ble t2, t1, .Ls0 # until counter == nb to check

57

58 # array has now the values: 0 0 2 3 4 5 6 7 8 9 10...

59

60 # non-primes are cancelled out

61 # by setting their array items to zero

62 li t2, 2 # start with 2, t2 is index i

63 la t3, array # t3 is pointer to array

64 .Ls1:

65 lw t4, 8(t3) # t4 is current array item (offset by 2)

66 beqz t4, .Ls3 # no prime, continue at .Ls3

67

68 mul t4, t2, t2 # t4 = t2 \* t2; t4 is index j

69 .Ls2:

70 slli t5, t4, 2 # t5 = t4 \* 4 for offset (words) in array

71 add t5, t3, t5 # t5 = t3 + t5; t5 is address in array for j

72 sw x0, 0(t5) # set entry to 0, no prime nb, array[j] = 0

73 add t4, t4, t2 # t4 = t4 + t2; j += i

74 ble t4, t1, .Ls2 # cancel out all multiples of i for i < n

75

76 .Ls3:

77 addi t2, t2, 1 # continue with next number

78 mul t0, t2, t2 # as long as n\*n > index

79 ble t0, t1, .Ls1

80

81 slli t0, t1, 2 # use n as index

82 add t0, t3, t0 # compute address in array

83 lw t0, 0(t0) # load its item

84 snez a0, t0 # set a0 to 1 if array[n] != 0

85

86 ret

87

88

89 .section .rodata

90 prompt:

91 .asciz "Enter number (<1048576): "

92 scanfmt:

93 .asciz "%u" # scanf an unsigned int number

94 outyes:

95 .asciz "is a prime number.\n"

96 outno:

97 .asciz "is not a prime number.\n"

98 error:

99 .asciz "wrong input.\n"

100 .section .bss

101 input: # storage for numbers

102 .word 0

103 array:

104 .zero 4\*maxnb # max number storage

Строки 01-41 реализуют с вводом числа и выводом результата. Решето Эратосфена реализовано в строках 43-86. Строки 89-104 описывают данные, с которыми происходит работа.

Первая часть проверки чисел на простоту инициализирует массив (строки 49-56). Основная часть состоит в исключении всех кратных чисел, начиная с числа два (строки 62-79). Для этого обрабатывается каждый элемент массива и если он уже исключён (равен 0), берется следующий элемент (строки 64-66). В противном случае аннулируются все кратные числа, начиная с квадрата рассматриваемого числа (строки 68-74, внутренний цикл) и заканчивая самим введенным числом. Затем обрабатывается следующий элемент массива (строки 64-79, внешний цикл). Это продолжается до тех пор, пока рассматриваемое число меньше или равно введенному числу (строки 77-79). Элемент массива, соответствующий введённому числу, проверяется в конце на предмет того, равен он нулю (число не простое) или нет (число- простое) (строки 81-84).

riscv64-linux-gnu-as -o prime.o prime.s

riscv64-linux-gnu-gcc -o prime prime.o

export QEMU\_LD\_PREFIX=/usr/riscv64-linux-gnu/

qemu-riscv64-static prime

Enter number (<1048576): 1034123

is a prime number.

# Глава 6. Примеры

## Введение

### Обзор главы

В этой главе рассматриваются более продвинутые темы программирования под RISC-V на примере использования структур данных (реализуется связный список) и работы с числами с плавающей запятой. Цель этой главы – показать, как реализуются базовые алгоритмы, применяемые в вычислительной технике.

### Цели обучения

После прочтения этой главы вы должны быть способны:

* реализовывать структуры данных и использовать функции malloc и free стандартной библиотеки языка C;
* писать программы, работающие с числами с плавающей запятой.

## Примеры

### Связный список

Связанный список – простейшая структура данных. Связный список состоит из элементов, содержащих данные, и указателей на другие аналогичные элементы. Указатель – специальная переменная, которая хранит адрес в памяти, в данном случае – адрес элемента списка. Элементы списка упорядочены в том смысле, что у каждого элемента есть потомок, связанный с текущим элементом по указателю. Есть первый элемент, называемый головой списка – указатель на самый первый элемент списка. Есть последний элемент, называемый хвостом списка - задаётся элементом без потомка, указатель на потомка у такого элемента равен нулю (нулевой указатель).

Рассмотрим список, состоящий из двух элементов, имеющих значения 0x32 и 0x12. Первый элемент расположен в памяти по адресу 0x32c0, а второй – по адресу 0x32a0. Каждый элемент содержит значение и указатель на следующий элемент (он может быть нулевым). Голова списка указывает на самый первый элемент: head = 0x32c0:

Адрес Значение Указатель

0x32c0: 0x32 0x32a0

0x32a0: 0x12 0x0000

Самый простой способ вставить элемент в список – затолкнуть его в голову, при этом для каждого нового элемента требуется выделить место в памяти, что обычно делается динамически через менеджер памяти и область динамической памяти, называемой «куча». В процессе удаления элемента из списка (выталкивание) должна освободиться память. Для решения задач выделения и освобождения памяти используются C-функции **malloc** и **free**.

Элемент списка на языке Assembler может быть задан с помощью полей, которые предоставляют доступ к значению элемента списка и указателю на следующий элемент. Это делается в примере:

# linkedlist\_struct.s

# offset to value

.equ node\_val, 0

# offset to pointer(address) of next element

.equ node\_next, 8

# size of one element

.equ node\_size, 16

# calls:

# linkedlist\_push

# input> a0: head, a1: value

# output< a0: head or -1 if error

# linkedlist\_pop

# input> a0: head

# output< a0: head, a1: value

# linkedlist\_print

# input> a0: head

В этом примере для работы со связным список реализованы три функции: для добавления элемента в список (операция заталкивания, push), для удаления элемента из списка (операция выталкивания, pop), и для печати значений всех элементов списка от головы до хвоста. Функция push создаёт новый элемент, при этом бывшая голова списка становится потомком для только что созданного элемента, а голова (указатель на начало списка) теперь указывает на созданный элемент. Функция pop инвертирует это действие, голова списка начинает указывать на потомок текущего элемента, а сам элемент, ранее бывший головой списка, удаляется. Функция print печатает все элементы.

# linkedlist.s

.include "src/linkedlist\_struct.s"

.section .text

.globl linkedlist\_push

linkedlist\_push:

addi sp, sp, -24

sd ra, 16(sp)

sd a0, 8(sp)

sd a1, 0(sp)

# alloc memory

li a0, node\_size

call malloc

beqz a0, .L0err

# value

ld t1, 0(sp)

sd t1, node\_val(a0)

# insert as new head head

ld t0, 8(sp)

sd t0, node\_next(a0)

# || val | next ->|| -> || ... | ... ||

j .L0exit

.L0err:

li a0, -1

.L0exit:

ld ra, 16(sp)

addi sp, sp, 24

ret

.globl linkedlist\_pop

linkedlist\_pop:

addi sp, sp, -16

sd ra, 8(sp)

sd s0, 0(sp)

# if head is zero

beqz a0, .L1err

# return value

ld a1, node\_val(a0)

# pointer to next element will be new head

ld s0, node\_next(a0)

# free memory

call free

# return new head

mv a0, s0

j .L1exit

.L1err:

li a0, -1

.L1exit:

ld s0, 0(sp)

ld ra, 8(sp)

addi sp, sp, 16

ret

.globl linkedlist\_print

linkedlist\_print:

addi sp, sp, -24

sd ra, 16(sp)

sd a0, 8(sp)

sd a0, 0(sp)

.L2loop:

beqz a0, .L2exit

sd a0, 0(sp)

ld a1, node\_val(a0)

la a0, .L2prompt

call printf

ld a0, 0(sp)

ld a0, node\_next(a0)

j .L2loop

.L2exit:

ld a0, 8(sp)

ld ra, 16(sp)

addi sp, sp, 24

ret

.section .rodata

.L2prompt:

.asciz "%u \n"

Использование связного списка показано в основном коде. Элементы добавляются, удаляются, печатаются.

# main.s

.include "src/linkedlist\_struct.s"

.section .text

.globl main

main:

addi sp, sp, -8

sd ra, 0(sp)

# new head of list

li s0, 0

mv a0, s0

li a1, 0x12

call linkedlist\_push

li a1, 0x23

call linkedlist\_push

li a1, 0x45

call linkedlist\_push

call linkedlist\_pop

li a1, 0x56

call linkedlist\_push

call linkedlist\_print

call linkedlist\_pop

call linkedlist\_pop

call linkedlist\_pop

ld ra, 0(sp)

addi sp, sp, 8

ret

### Подсчёт среднего значения

Следующий пример показывает использование регистров для работы с числами с плавающей запятой. Она начинается с чтения дробных чисел с помощью функции **scanf** (строки 09-19) в память (по метке buffer). Числа с плавающей запятой занимают 4 байта. Программа в ходе своей работы подсчитывает сумму всех значений, для этой цели используются специальные регистры для работы с числами с плавающей запятой ft0, ft1 и ft3. Целочисленные значения могут быть сконвертированы в числа с плавающей запятой инструкцией **fcvt.s.w** (конвертирует знаковое целое длинной в слово в дробное значение), это происходит на строках 24 и 25. Второй цикл (строки 26-36) вычисляет сумму путём загрузки чисел с плавающей запятой из памяти (строка 34) и суммируя числа с сохранением результата в регистре ft1 (строка 35). Среднее значение вычисляется благодаря операции деления (строка 38). В итоге, результат печатается с использованием функции **printf**. Функция **printf** требует аргумент в виде числа двойной точности, загруженного в регистр a1, эта операция реализуется в строчках 41 и 42. Программе нужны расширения F и D, описанные в спецификации. Программе на стандартный ввод можно перенаправить данные из файла используя возможности перенаправления потоков операционной системы: **average < numbers.txt**.

00 # average.s

01 .equ maxNb, 100 # maximal numbers to be read

02 .section .text

03 .globl main # run in C 'environment'

04 main:

05 addi sp, sp, -16 # store ra (return address) and saved regs on stack

06 sd ra, 0(sp)

07 sd s0, 8(sp)

08

09 li s0, 0 # counter numbers: i

10 .L0input: # read in numbers

11 la a0, scanfmt

12 la a1, buffer

13 slli t0, s0, 2 # next input in buffer+4\*i

14 add a1, a1, t0

15 call scanf # read float number

16 blez a0, .L0avg # if no number could be read anymore, continue..

17 addi s0, s0, 1 # count numbers read in s0: i = i + 1

18 li t0, maxNb # if i < maxNb, continue reading

19 blt s0, t0, .L0input

20

21 .L0avg:

22 beqz s0, .L0err # check if at least one number is read, else exit

23

24 fcvt.s.w ft0, s0 # divider: convert from int in s0 to float

25 fcvt.s.w ft1, x0 # sum: init with zero, convert from x0

26 .L0avgloop:

27 beqz s0, .L0out # count down s0 to zero: i = i - 1

28 addi s0, s0, -1

29

30 slli t1, s0, 2 # compute address to float number in memory

31 la t0, buffer

32 add t0, t0, t1

33

34 flw ft2, 0(t0) # load from memory to float register ft2

35 fadd.s ft1, ft1, ft2 # sum in ft1

36 j .L0avgloop

37 .L0out:

38 fdiv.s ft1, ft1, ft0 # average in ft1

39

40 la a0, resultfmt # print result using printf

41 fcvt.d.s ft1, ft1 # need to prepare for printf, expand to double size

42 fmv.x.d a1, ft1 # move from ft1 to a1

43 call printf # and print

44

45 .L0err:

46 li a0, 0

47

48 ld s0, 8(sp)

49 ld ra, 0(sp) # restore ra

50 addi sp, sp,16

51 ret # return to caller

52

53 .section .rodata

54 scanfmt:

55 .asciz "%f"

56 resultfmt:

57 .asciz "Average: %f\n"

58

59 .section .bss

60 buffer:

61 .zero 4\*maxNb

1. WSL (Windows Subsystem for Linux) - подсистема ОС Windows, которая позволяет разработчикам запускать среду GNU/Linux с большинством программ командной строки, служебных программ и приложений непосредственно в Windows без каких-либо изменений и необходимости использовать традиционную виртуальную машину или двойную загрузку. Подробнее: <https://learn.microsoft.com/ru-ru/windows/wsl/about> [↑](#footnote-ref-1)