**СКВОЗНОЙ ЛАБОРАТОРНЫЙ ПРАКТИКУМ ПО ТЕХНОЛОГИЯМ RISC-V**

Авторы: Заславский Марк Маркович, Костичев Сергей Валентинович, Буренева Ольга Игоревна, Морозов Сергей Михайлович, Терехина Софья Анатольевна

2023

Данный лабораторный практикум использует материалы учебных курсов The Linux Foundation:

* Introduction to RISC-V (LFD110x), <https://training.linuxfoundation.org/training/introduction-to-riscv-lfd110x/>,
* Foundations of RISC-V Assembly Programming (LFD117x), <https://training.linuxfoundation.org/training/foundations-of-risc-v-assembly-programming-lfd117x/>,
* RISC-V Toolchain and Compiler Optimization Techniques (LFD113x), <https://training.linuxfoundation.org/training/risc-v-toolchain-and-compiler-optimization-techniques-lfd113x/>,
* Building Applications with RISC-V and FreeRTOS (LFD112x), <https://training.linuxfoundation.org/training/building-applications-with-risc-v-and-freertos-lfd112x/>,
* Microcontroller Applications with RISC-V (LFD115x), <https://training.linuxfoundation.org/training/microcontroller-applications-with-risc-v-lfd115x/>,

распространяемых под лицензией [CC BY 4.0](https://creativecommons.org/licenses/by/4.0/). Разработка лабораторного практикума выполнена авторским коллективом (Заславский Марк Маркович, Костичев Сергей Валентинович, Буренева Ольга Игоревна, Морозов Сергей Михайлович, Терехина Софья Анатольевна) по заказу [Альянса RISC-V](https://riscv-alliance.ru/), допускается к использованию под лицензией [CC BY 4.0](https://creativecommons.org/licenses/by/4.0/deed.ru).

# 

# Оглавление

# 

[**Оглавление 2**](#_rddko13oa933)

[**ПРОГРАММА ПРАКТИКУМА 5**](#_lua9pef5xzq)

[Состав практикума 5](#_h31sk9u40fpu)

[Для кого предназначен данный практикум 5](#_dqknggj4d077)

[Как адаптировать задания практикума для различных условий 5](#_wrdso27ka78x)

[**Лабораторная работа № 1. ЗНАКОМСТВО С РАБОЧЕЙ СРЕДОЙ Syntacore Kit 6**](#_7r1yk9u3m53r)

[1.1 Цель и задачи 6](#_6gi5kw1t2p42)

[1.2. Основные теоретические сведения 6](#_8n0mgox3yeyh)

[1.2.1. Syntacore Kit 6](#_t2eoy355ydv2)

[1.2.2. Компиляция и запуск программ в Syntacore Kit 7](#_8p26g3oxzgy4)

[1.2.2.1 Запуск через qemu-riscv64 11](#_h8r21sl4wjbt)

[1.2.2.2 Копирование программы в RISC-V ОС 11](#_8ntq49kd2xe9)

[1.2.2.3 Создание приложений в Syntacore SC-IDE 11](#_91m0no8tfarc)

[1.2.3. Получение сведений об аппаратном обеспечении в Linux 16](#_c1gtn7fpjmvr)

[1.3. Задание к лабораторной работе 17](#_d5pb25wrkc6y)

[1.3.1. Описание последовательности выполнения работы 18](#_9udaxfjspqob)

[1.3.2. Пример выполнения задания на защиту 18](#_u3r52v3mnin7)

[1.4. Вопросы для контроля 19](#_4mi8w4oj2vvq)

[**Лабораторная работа № 2. КОМПИЛЯЦИЯ И ЗАПУСК АССЕМБЛЕРНОЙ ПРОГРАММЫ 20**](#_isuchike6kmc)

[2.1 Цель и задачи 20](#_scwksc69frsq)

[2.2. Основные теоретические сведения 20](#_96tq9z134vb)

[2.2.1 Структура ассемблерных программ 21](#_ubx66ahow10d)

[2.2.2 Компиляция ассемблерных программ 23](#_7nnnut8j4fzz)

[2.3. Задание к лабораторной работе 24](#_kbibskguzyeb)

[2.3.1. Описание последовательности выполнения работы 25](#_yh4et396zzd7)

[2.3.2. Пример выполнения задания на защиту 25](#_h7r9k2w27rgx)

[2.4. Вопросы для контроля 26](#_5lsbpznjfdri)

[**Лабораторная работа № 3 ИЗУЧЕНИЕ АРХИТЕКТУРЫ RISC-V НА ПРИМЕРЕ АССЕМБЛЕРНОЙ ПРОГРАММЫ 26**](#_ek399j2qklsv)

[3.1 Цель и задачи 26](#_b8853qxs36yk)

[3.2. Основные теоретические сведения 26](#_62vg6lda1rt4)

[3.2.1 Базовый набор регистров и альтернативные имена в стандарте ABI (application binary interface) 28](#_1wplin3s9ddc)

[3.3. Задание к лабораторной работе 30](#_xmpvdiilr9tm)

[3.3.1. Описание последовательности выполнения работы 30](#_xudum7y8yvb7)

[3.3.2. Пример выполнения задания на защиту 30](#_z3m7vxx4knp3)

[3.4. Вопросы для контроля 31](#_lnob65v69toh)

[**Лабораторная работа №4 ЛОГИЧЕСКИЕ И АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ, ОПЕРАЦИИ СРАВНЕНИЯ И СДВИГА 31**](#_b3m2tob599hv)

[4.1 Цель и задачи 31](#_cd7jdgt9acqb)

[4.2. Основные теоретические сведения 31](#_ufne8pg0bqqh)

[4.2.1 Логические и арифметические операции I-type 31](#_7xpkfqkd59wd)

[4.2.2 Логические и арифметические операции R-type 33](#_2ggmywwpyim4)

[4.3. Задание к лабораторной работе 34](#_igm2gvbe7i8p)

[4.3.1 Описание последовательности выполнения работы 35](#_mbwvp3ncce3r)

[4.3.2 Пример выполнения задания на защиту 35](#_fat9g8g3huff)

[4.4. Вопросы для контроля 36](#_bsz0bw2dq2t7)

[**Лабораторная работа № 5 РЕЖИМЫ АДРЕСАЦИИ 36**](#_paxt6ae5471)

[5.1 Цель и задачи 36](#_au5b20fsnd14)

[5.2. Основные теоретические сведения 37](#_fzf8t648dnr6)

[5.2.1 Регистровая адресация 37](#_majfu6sr0sj)

[5.2.2 Непосредственная адресация 37](#_uikb02wro0v8)

[5.2.3 Базовая адресация 38](#_8gdmb89fy21z)

[5.2.4. Адресация относительно счетчика команд 39](#_6abr6mlypcrc)

[5.3. Задание к лабораторной работе 40](#_7i43w1mv8ap8)

[5.3.1. Описание последовательности выполнения работы 41](#_3s8fxjhj5fn5)

[5.3.2. Пример выполнения задания на защиту 41](#_qnti7p46yhr1)

[5.4. Вопросы для контроля 42](#_g86kxfjh4in1)

[**Лабораторная работа № 6 АДРЕСАЦИЯ ЭЛЕМЕНТОВ МАССИВА, ОРГАНИЗАЦИЯ ЦИКЛА 42**](#_l4j1dv5wbfsi)

[6.1 Цель и задачи 42](#_78o6fkcj2kq7)

[6.2. Основные теоретические сведения 43](#_367tc1c8lyqv)

[6.2.1 Хранение массивов в памяти 43](#_yzpo864acpbo)

[6.2.2 Организация циклов 44](#_4bxtvkiit0a)

[6.3. Задание к лабораторной работе 47](#_ibkm519k1alo)

[6.3.1 Описание последовательности выполнения работы 48](#_yj46ucl13an4)

[6.3.2 Пример выполнения задания на защиту 48](#_lfsysz7j3l04)

[6.4. Вопросы для контроля 50](#_6f4jgcdk49dn)

[**Лабораторная работа №7 ВЫЗОВ ПОДПРОГРАММЫ, РАБОТА СО СТЕКОМ 51**](#_8drq1shq2bon)

[7.1. Цель и задачи 51](#_wonf20dv5ro2)

[7.2. Основные теоретические сведения 52](#_vxkiqiz9aysz)

[7.2.1 Директивы размещения данных в памяти 52](#_8f1hycuj7iw)

[7.2.2 Системные вызовы 53](#_b8oe02sqh444)

[7.2.3 Подпрограммы 56](#_bkh8euc8tjhz)

[7.2.4 Стек 59](#_g59fsalnbedd)

[7.3 Задание к лабораторной работе 60](#_p3ku5qb2z2ue)

[7.3.1 Описание последовательности выполнения работы 60](#_tkm34z9csykd)

[7.3.2 Пример выполнения задания на защиту 60](#_uqbo1j69sa2p)

[7.4 Вопросы для контроля 62](#_cvgg4e8og00e)

[**Лабораторная работа №8 ОПТИМИЗАЦИИ НА ЭТАПЕ КОМПИЛЯЦИИ 62**](#_8tn2v71rqwis)

[8.1 Цель и задачи 62](#_kp7896lrfvhk)

[8.2. Основные теоретические сведения 62](#_qsblarwpwz92)

[8.2.1 Оптимизации на уровне компилятора 63](#_2tpy4hj3msyq)

[8.2.2 Инструментирование и профилирование 64](#_jpn21lo1hpdy)

[8.2.4 Измерение ускорения от оптимизаций 66](#_xu6a7rpjj2jj)

[8.2.5 Измерение уменьшения объема бинарного файла от оптимизаций 69](#_epund9e99ofw)

[8.3. Задание к лабораторной работе 69](#_nn3akeyg3xe0)

[8.3.1. Описание последовательности выполнения работы 71](#_vgmh9nn3admo)

[8.3.2. Пример выполнения задания на защиту 71](#_g3hze3prmo4p)

[8.4. Вопросы для контроля 72](#_p7nacjrjpnnu)

[**Лабораторная работа №9 МЕТОДЫ ОТЛАДКИ ПРОГРАММ 72**](#_x5vjsj6t07nl)

[9.1 Цель и задачи 72](#_z7m2d66a0wg0)

[9.2. Основные теоретические сведения 73](#_f4fod9tt201x)

[9.3. Задание к курсовой работе 76](#_yxjkfp2fqpbs)

[9.3.1. Описание последовательности выполнения работы 77](#_og7iwznfnxu2)

[9.3.2. Пример выполнения задания на защиту 78](#_bzld60edvgbr)

[9.4. Вопросы для контроля 80](#_9l7l2zry1j9r)

[Лабораторная работа №10 НАСТРОЙКА И ЗАПУСК ЗАДАЧ FreeRTOS 81](#_9c3twshgmxla)

[10.1 Цели и задачи 81](#_ee0wmkqarz42)

[10.2 Основные теоретические сведения 81](#_494nc1z72s4c)

[10.2.1 Сборка и запуск программ FreeRTOS 87](#_tlpkwj4qjwpd)

[10.3 Задание к лабораторной работе 89](#_dypc4tljqu5o)

[10.3.1 Описание последовательности выполнения работы 90](#_b9s3ee6m7s8o)

[10.3.2 Пример выполнения задания на защиту 90](#_wgotzjkj66tc)

[10.4 Вопросы для контроля 92](#_2xcmr6z3xxcz)

[Лабораторная работа №11 МЬЮТЕКСЫ FreeRTOS 92](#_b4n0u2jb3u13)

[11.1 Цели и задачи 92](#_nao8cgu61yue)

[11.2 Основные теоретические сведения 92](#_kdb60by0tv95)

[11.3 Задание к лабораторной работе 94](#_nqw7l8t6lbev)

[11.3.1 Описание последовательности выполнения работы 95](#_g4s15wfh78y2)

[11.3.2 Пример выполнения работы 95](#_1cbr6qll0vhl)

[11.5 Вопросы для контроля: 96](#_xu3uy81lwfw7)

[Лабораторная работа №12 ОЧЕРЕДИ FreeRTOS 97](#_dctr5qimv7lx)

[12.1 Цели и задачи 97](#_kwf677bpr5bl)

[12.2 Основные теоретические сведения 97](#_37sfogd57vj1)

[12.3 Задание к лабораторной работе 100](#_io9wlvhdgt1h)

[12.3.1 Описание последовательности выполнения работы 101](#_jd2p9tmwe9ug)

[12.3.2 Пример выполнения 101](#_m0dux1kf1pst)

[12.4 Вопросы для контроля 103](#_adklv7krspqp)

[**Лабораторная работа № 13 Знакомство со средой проектирования. Управление контактами GPIO 104**](#_jxmvndkcd7u1)

[13.1 Цели и задачи 104](#_8ncez9te5y8c)

[13.2 Основные теоретические сведения о работе с микроконтроллерами для RISC-V 104](#_p1hfa62zhper)

[13.2.1 Описание платы прототипирования CORE ESP32 111](#_2o3jnkxl2f83)

[13.2.2 Подготовка среды проектирования 113](#_53hzlsxs6uk2)

[13.2.3 GPIO 115](#_k3wywf49inoj)

[13.2.4 Ввод данных в периферийные устройства через GPIO матрицу 116](#_v8gsej4uxi8e)

[13.2.5 Вывод данных из периферийных устройств через GPIO матрицу 117](#_6tbfyyqogs67)

[13.3 Задания на лабораторную работу 120](#_q2pj1ca89ctx)

[13.3.1 Описание последовательности выполнения работы 120](#_hr502vc5p3jp)

[13.3.2 Пример выполнения задания на защиту 121](#_h7jhvtk3u3po)

[13.4 Вопросы для контроля 121](#_yaj5aaltkwne)

[**Лабораторная работа № 14 Работа с LED PWM контроллером 122**](#_28ltu261axkh)

[14.1 Цель и задачи 122](#_vs1i2alr89ta)

[14.2 Краткие теоретические сведения 122](#_gq8k21j016g)

[14.3 Задание к лабораторной работе 125](#_hswt30sf25as)

[14.2.1 Описание последовательности выполнения работы 125](#_b3dago74m5jl)

[14.2.2 Пример выполнения задания на защиту 125](#_n37kr3v2qg7q)

[14.4 Вопросы для контроля 127](#_t4bh86mpkbpm)

[**Лабораторная работа № 15 Настройка GPIO для ввода аналогового сигнала 127**](#_r8xrx77eudun)

[15.1 Цель и задачи 127](#_931g8hi59s6r)

[15.2 Краткие теоретические сведения 128](#_hno5ysmxzvaf)

[15.3 Задания к лабораторной работе 131](#_go5p26lnwsh5)

[15.3.1 Описание последовательности выполнения работы 131](#_3rbmtju1itk5)

[15.3.2 Пример выполнения задания на защиту 131](#_g90hqu23fn4t)

[15.4 Вопросы для контроля 133](#_i30up2l25z42)

[**СПИСОК ОСНОВНОЙ ЛИТЕРАТУРЫ 134**](#_2i3p0l66j4rg)

[**СПИСОК ДОПОЛНИТЕЛЬНОЙ ЛИТЕРАТУРЫ ДЛЯ САМОСТОЯТЕЛЬНОГО ИЗУЧЕНИЯ 136**](#_ffnxztd8kbo)

# 

# ПРОГРАММА ПРАКТИКУМА

## Состав практикума

Данный практикум покрывает пять больших тематик:

1. Введение в инструменты разработчика RISC-V на примере Syntacore Kit.
2. Язык ассемблера для RISC-V.
3. Оптимизации по на уровне компилятора и методы отладки для RISC-V.
4. Разработка ПО для операционных систем реального времени(ОСРВ) на примере FreeRTOS.
5. Программирование для микроконтроллеров на базе RISC-V.

Организация практикума именно преследует цель — **познакомить студента с особенностями архитектуры RISC-V на различных уровнях разработки**. Именно для этой цели большое внимание уделяется изучению ассемблера. С точки зрения авторов практикума, понимание архитектуры невозможно без умения самостоятельно составлять программы на данном языке. Остальные темы позволяют получить минимальные необходимые навыки для самых востребованных областей, где применяется RISC-V — микроконтроллеры и встраиваемая разработка.

## Для кого предназначен данный практикум

Практикум ориентирован на широкую аудиторию студентов технических вузов, обучающихся по специальностям, связанным с разработкой ПО и информационно-вычислительной техникой. Авторы ориентируют практикум на студентов второго курса, обладающих начальным опытом программирования в объеме первого года занятий. Для освоения практикума желательны, но не обязательны следующие навыки:

* Программирование на языке C.
* Работа в командной строке Linux.
* Базовые представления о многопоточном программировании.

## Как адаптировать задания практикума для различных условий

В ряде случаев уровень заданий лабораторных работ из практикума может оказаться простым для сильных групп или студентов, имеющих продвинутые знания по рассматриваемым тема. В таком случае авторы призывают преподавателей вносить изменения в формулировки заданий — например, усложнять требуемое поведения или включать дополнительные шаги. Для ряда лабораторных, где такое усложнение может быть неочевидно, даны прямые указания как именно можно изменить задание.

# 

# **Лабораторная работа № 1. ЗНАКОМСТВО С РАБОЧЕЙ СРЕДОЙ Syntacore Kit**

## **1.1 Цель и задачи**

Целью работы является освоение работы с виртуальным окружением лабораторного практикума (Syntacore Kit).

Для достижения поставленной цели требуется решить следующие задачи:

1) Изучить структуру виртуального окружения, научится запускать виртуальные машины.

2) Ознакомиться с основными командами, позволяющими проверить исправность среды.

3) Изучить инструменты кросс-компиляции программ.

4) Написать программу-пример для работы в виртуальной машине на базе RISC-V.

## 1.2. Основные теоретические сведения

### **1.2.1. Syntacore Kit**

Виртуализация — создание виртуального представления (виртуальных устройств) вычислительных ресурсов (процессоров, оперативной памяти, сетевых устройств и т.д.), абстрагированного от их физического воплощения [1]. Интерфейсы и поведение виртуальных устройств максимально повторяют реальные устройства, однако за счет своей программной основы их можно в любой момент изменять.

Syntacore Kit –- набор инструментов разработчика для платформы RISC-V, организованный в виде набора виртуальных машин. Внешне Syntacore Kit представляет собой образ виртуальной машины VirtualBox [2], внутри которого содержится ОС (операционная система) Ubuntu 22.04 с процессором x86\_64, необходимые инструменты разработчика (инструменты командной строки и Syntacore SC-IDE) , а также виртуальная машина QEMU с процессором RISC-V и ОС Debian Unstable с ядром Linux версии 5.18. В дальнейшем мы будем использовать следующие термины:

* **ОС хоста** — операционная система компьютера, на котором вы выполняете лабораторные работы
* **Гостевая ОС** — операционная система Ubuntu 22.04, работающая на виртуальной машине VirtualBox, запускаемой в ОС хоста.
* **RISC-V ОС —** операционная система Debian Unstable с ядром Linux версии 5.18 без графической оболочки, работающая на виртуальной машине QEMU (с виртуальным процессором RISC-V), запускаемой на гостевой ОС.

Syntacore Kit позволяет разработчикам создавать, компилировать и запускать тестировать программы для платформы RISC-V с помощью инструментов, расположенных в гостевой ОС. Для этого используется механизм кросс-компиляции – подхода, при котором код программы собирается для архитектуры, отличной от той, где запущен компилятор. Кросс-компиляторы входят в состав инструментов командной строки и Syntacore SC-IDE.

### **1.2.2. Компиляция и запуск программ в Syntacore Kit**

Ниже будет рассмотрена последовательность действий для запуска Syntacore Kit и компиляции программ на его виртуальных машинах. На момент начала лабораторной работы Syntacore Kit должен быть установлен в вашем компьютерном классе.

Если запуск выполнен успешно, то вы увидите интерфейс гостевой ОС (Рис 1.)

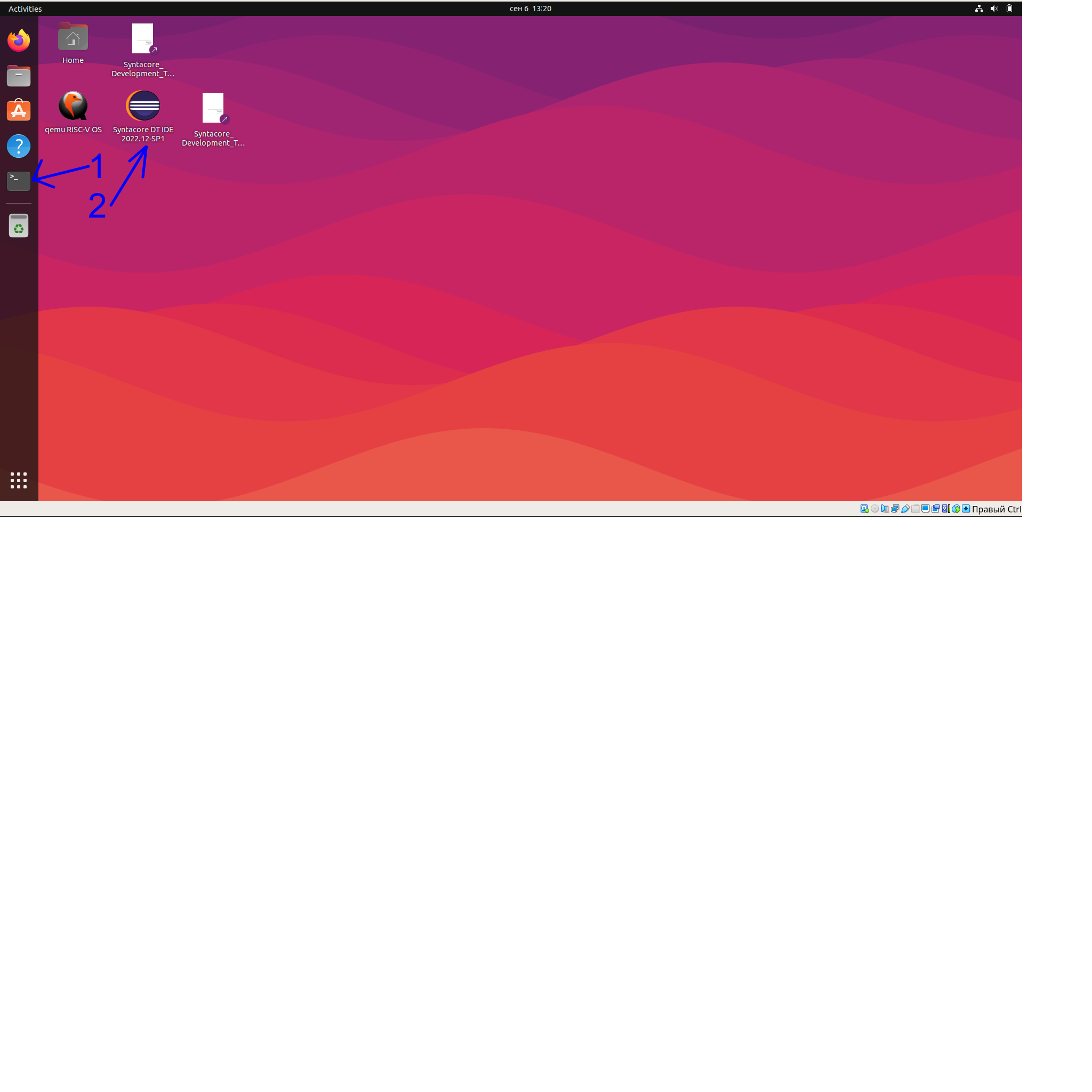


Рис. 1.1 Интерфейс гостевой ОС и основные элементы Syntacore Kit

Необходимые элементы Syntacore Kit для выполнения данной лабораторной работы:

1. Эмулятор терминала.
2. Ярлык запуска Syntacore SC-IDE.

Для того чтобы запустить RISC-V ОС, нажмите на ярлык №1 (Рис. 2.1). В открывшемся окне терминала выполните две команды

$ cd ~/riscv-debian-sandbox/artifacts

$ ./run.sh

После выполнения данных команд начнется процесс запуска виртуальной машины QEMU c RISC-V ОС. Полная загрузка RISC-V ОС может потребовать некоторого времени (порядка 1-2 минут). В случае успешного запуска в интерфейсе гостевой ОС отобразится следующее окно (Рис. 1.2).

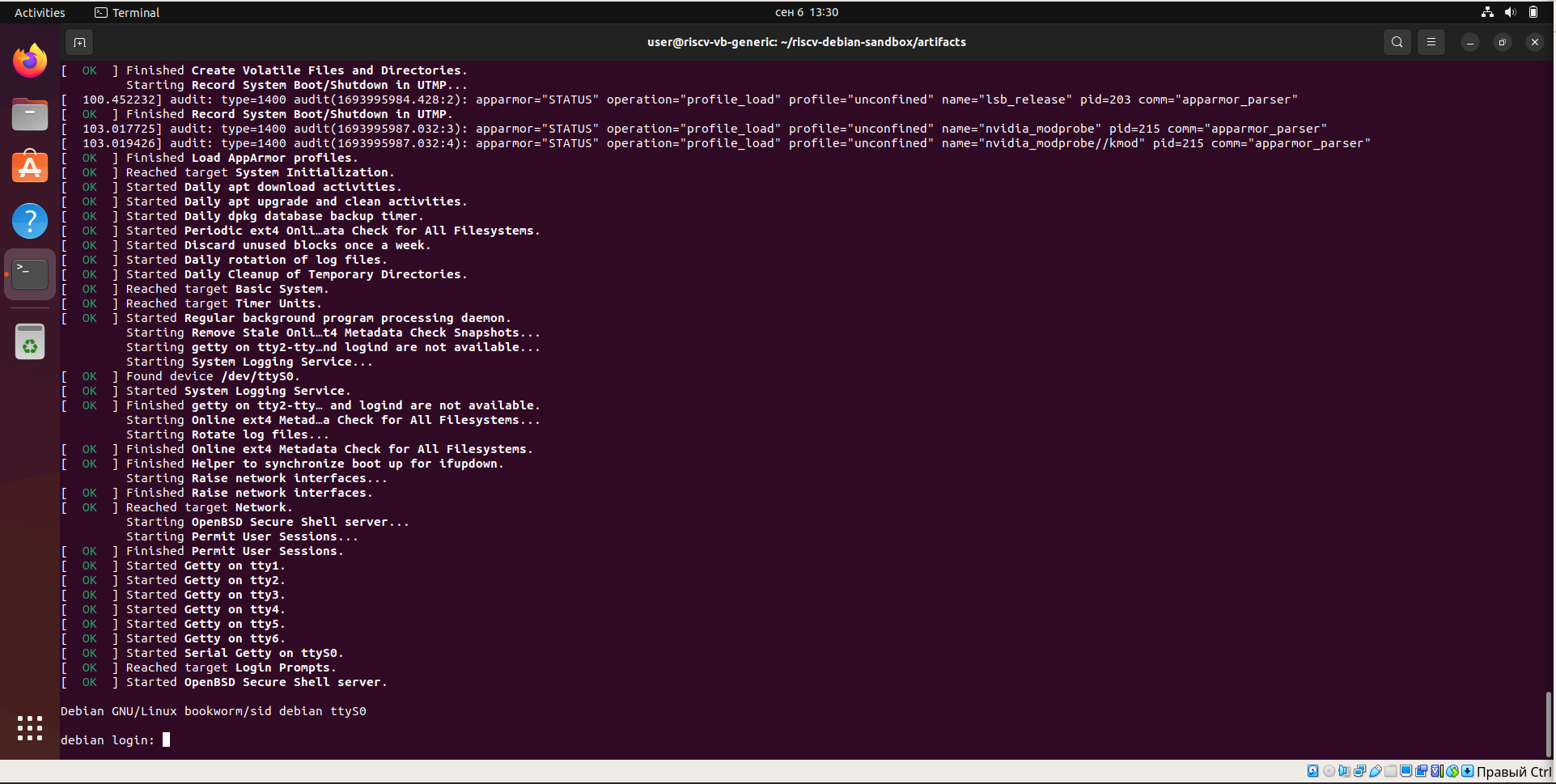


Рис. 1.2 Интерфейс авторизации RISC-V ОС

В окне вам будет предоставлен интерфейс авторизации RISC-V ОС. Для авторизации используйте логин **debian** и пароль **debian**. В случае успешной авторизации вам будет доступен интерфейс командной строки (Рис. 1.3).



Рис. 1.3 Интерфейс командной строки RISC-V ОС

В командной строке (Рис. 1.3) содержится минимальный инструментарий разработчика, позволяющий создавать и компилировать программы непосредственно в RISC-V ОС. Рассмотрим три основных способа создания RISC-V приложений в Syntacore Kit

**Создание приложений в терминале RISC-V ОС**

При данном способе и редактирование исходного кода, и его компиляция происходит в RISC-V ОС. Для сборки своего первого приложения выполните следующие команды:

$ cd ~/hello\_world\_test/ # Смена текущего каталога

$ gcc -o hello hello.c # Компиляция программы (флаг -o задает имя исполняемого файла

$ ./hello # запуск программы

В случае успешного выполнения в терминале должна отобразится строка “Hello from RISC-V”. Обратите внимание, что вы можете создавать новые программы или редактировать пример с помощью предустановленных текстовых редакторов в командной строке nano, vim и vi. Начинающим пользователям мы рекомендуем использовать nano.

**Создание приложений в терминале гостевой ОС**

Гостевая ОС имеет инструменты кросс-компиляции для сборки RISC-V приложений. Для начала работы в терминале создайте с помощью следующих команд файл hello.c:

$ nano hello.c

В открывшемся окне вставьте код программы выше с помощью буфера обмена. Затем нажмите сочетание клавиш CTRL + X (в английской раскладке). После чего, нажмите на клавиатуре Y (в английской раскладке) и клавишу Enter.

#include <stdio.h>

int main(void)

{

printf("Hello from RISC-V\n");

return 0;

}

Далее выполните команду кросс-компиляции:

$ riscv64-unknown-linux-gnu-gcc -o hello hello.c

Эта команда вызовет специальную версию компилятора gcc, которая создает приложения RISC-V. По итогу ее работы, на вашей файловой системе будет создан файл ./hello. Однако, если вы попробуете запустить его в том же терминале, то вас ждет ошибка:

$ ./hello

bash: ./hello: cannot execute binary file: Exec format error

Ошибка связана с тем, что архитектура исполняемого файла hello не соответствует архитектуре гостевой ОС. Поэтому для запуска созданной программы ее нужно необходимо запускать в RISC-V ОС. Для этого существует два способа: запуск через утилиту qemu-riscv64 и копирование файла в виртуальную машину. Для обоих способов необходимо чтобы у вас уже была запущена RISC-V ОС.

### 1.2.2.1 Запуск через qemu-riscv64

В ряде случаев, для запуска и отладки программ на архитектуре RISC-V не обязательно запускать полноценную виртуальную машину с RISC-V ОС. В этих случаях можно использовать утилиту qemu-riscv64, которая обеспечивает запуск легковесной эмуляции для одной программы.

Выполните следующую команду в терминале гостевой ОС:

$ qemu-riscv64 -L "$RISCV/sysroot" ./hello

Произойдет запуск и вы увидите сообщение “Hello from RISC-V”. Флаг -L устанавливает предварительно настроенное окружение для корректной работы RISC-V программ.

### 1.2.2.2 Копирование программы в RISC-V ОС

В терминале гостевой ОС выполните команду копирования по ssh[3]:

$ scp ./hello riscv-debian:/home/debian

Данная команда создала копию исполняемого файла ./hello из гостевой ОС в каталоге /home/debian RISC-V ОС. Теперь откройте окно терминала RISC-V ОС и выполните команды

$ cd ~

$ ./hello

Произойдет запуск и вы увидите сообщение “Hello from RISC-V”.

### 1.2.2.3 Создание приложений в Syntacore SC-IDE

Для запуска Syntacore SC-IDE нажмите на ярлык №2 (Рис. 1.4). В появившемся окне выберите пункт меню File -> New -> C/C++ project

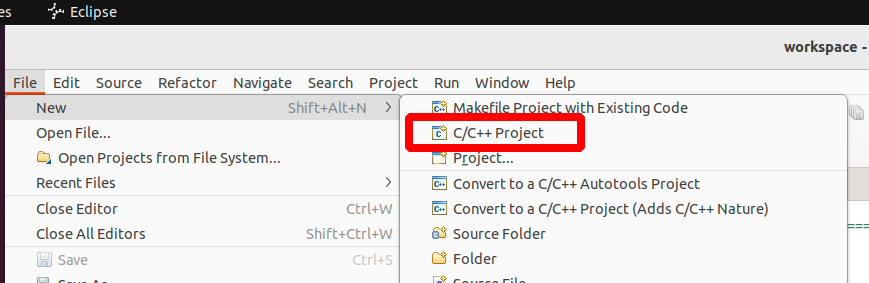


Рис. 1.4 Создание проекта в Syntacore SC-IDE

На экране (Рис. 1.5) появится диалог “New C/C++ project” . В нем необходимо выбрать опцию “C Managed build” и нажать Next.

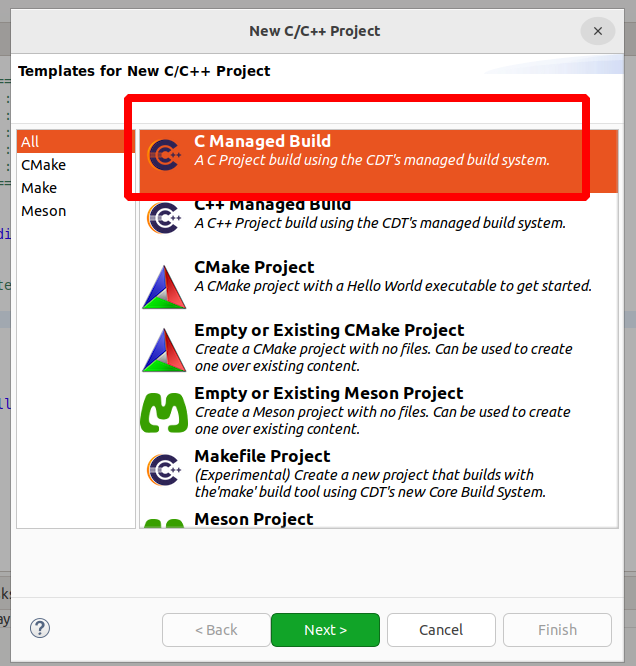


Рис. 1.5 Выбор типа сборки

На следующем шаге (Рис. 1.6) введите имя проекта (1), выберите Project type как “Syntacore RV32/64GC ANSI C/C++ Project” и “RISC-V Cross GCC” (2 и 3)

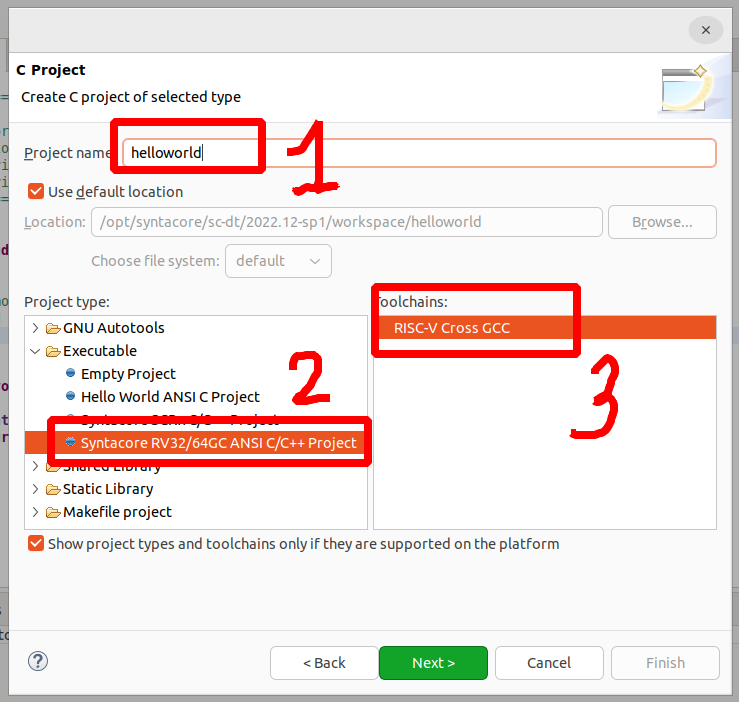


Рис. 1. 6 Настройки для работы с RISC-V

На следующем этапе (Рис. 1.7) убедитесь, что установлена опция “64 bit XLEN”.

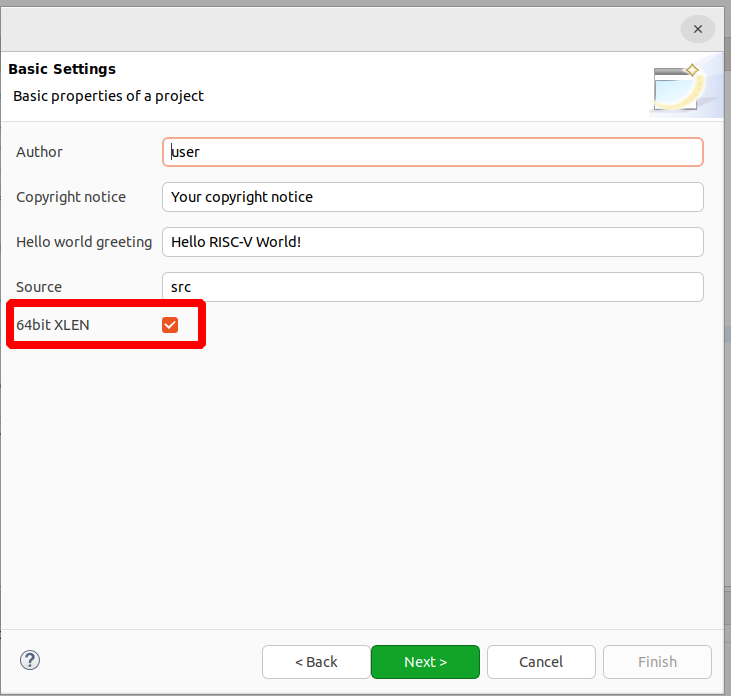


Рис. 1.7 Базовые настройки проекта

На следующем экране (Рис. 1.8) оставьте настройки по умолчанию и нажмите Next. На последнем экране выберите “Toolchain name” в соответствии с "RISC-V GCC/Linux (riscv64-unknown-linux-gnu-gcc)" и нажмите Finish.

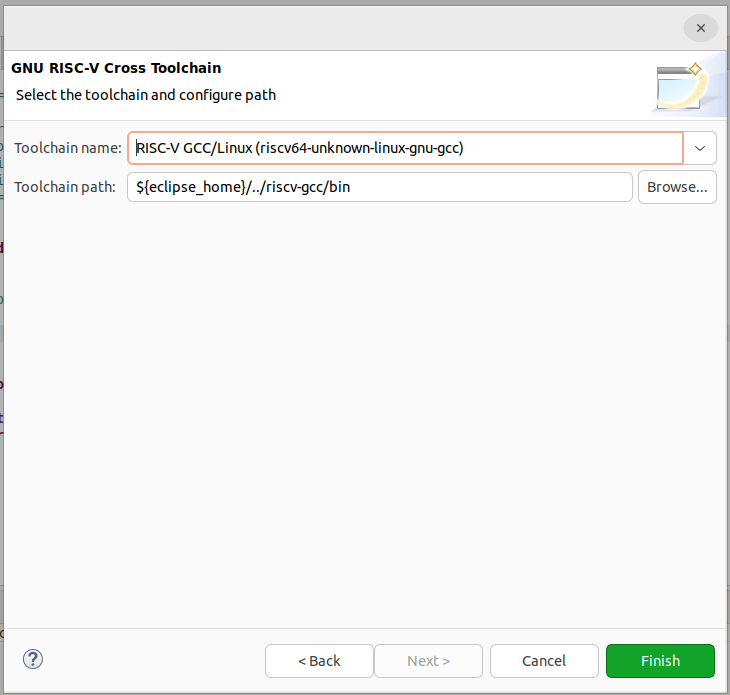


Рис. 1.8 Настройки тулчейна проекта

После этого (Рис. 1.9) у вас отобразится окно редактора файла main.c с кодом HelloWorld приложения. Для его запуска используйте команду

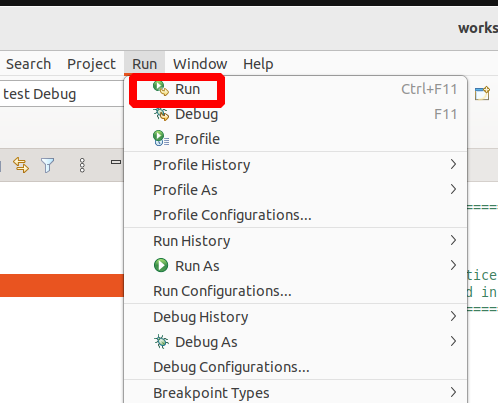


Рис. 1.9 Интерфейс запуска проекта.

Вывод запущенного приложения (Рис. 1.10) отобразится в нижней части среды разработки:

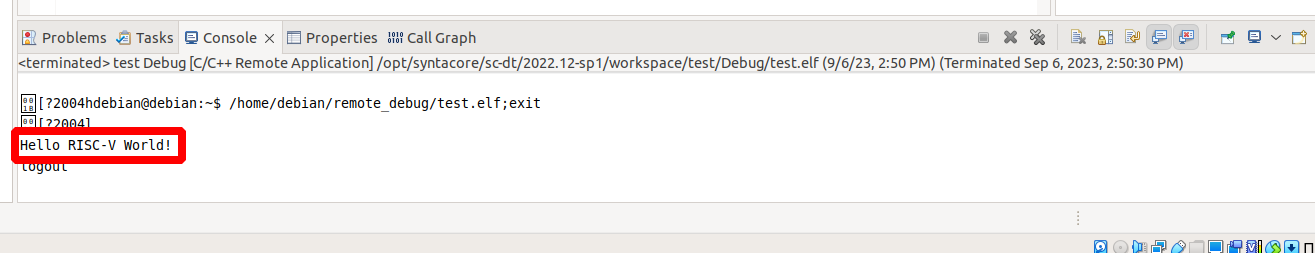


Рис. 1.10 Консоль запущенного приложения.

Обратите внимание, что помимо вывода самого приложения, выводятся также отладочные символы соединения с RISC-V ОС.

### **1.2.3. Получение сведений об аппаратном обеспечении в Linux**

Для выполнения данной лабораторной работы вам потребуется программно получать информацию о текущей аппаратной платформе. Для того чтобы узнать архитектуру, на которой работает Linux-система можно использовать команду arch:

$ arch # Гостевая ОС

x86\_64

$ arch # RISC-V ОС

riscv64

Расширенная информация о процессоре и ОЗУ содержится в procfs, в частности в /proc/cpuinfo и /proc/meminfo. С точки зрения ОС это текстовые файлы, содержимое которых можно вывести стандартными средствами (пример для гостевой ОС):

$ cat /proc/cpuinfo

processor : 0

vendor\_id : GenuineIntel

cpu family : 6

model : 142

model name : Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz

stepping : 9

cpu MHz : 2712.008

……

~$ cat /proc/meminfo

MemTotal: 2022664 kB

MemFree: 177740 kB

MemAvailable: 278060 kB

Buffers: 6136 kB

…..

## **1.3. Задание к лабораторной работе**

Напишите программу на языке С, которая последовательно выводит информацию о процессоре (каждый пункт списка выводится в новой строке):

* архитектура
* название модели
* общее количество ядер и потоков,
* для каждого ядра: в одну строку — название модели, размер физического адреса (биты), размер виртуального адреса (биты).

Программа должна проверять доступность источников данных и в случае невозможности получить информацию от ОС выводить код возврата 1 и соответствующее сообщение об ошибке.

Программу необходимо скомпилировать и запустить последовательно в гостевой ОС и в RISC-V ОС (либо через qemu-riscv64).

### **1.3.1. Описание последовательности выполнения работы**

Для выполнения поставленной задачи необходимо в первую очередь понять, с какими источниками данных о аппаратном обеспечении предстоит работать. В данном случае предлагается использовать чтение из служебного файла файл /proc/cpuinfo и захват вывода команды arch.

Далее необходимо составить универсальную программу, которая будет решать поставленную задачу вне зависимости от архитектуры процессора, под которую она скомпилирована. Авторы предлагают вынести в отдельные функции работу с /proc/cpuinfo и работу с командой arch. Обратите внимание на формат файла cpuinfo — каждое ядро процессора выводится отдельным блоком. Примеры кода можно посмотреть в разделе 1.2.

Далее необходимо скомпилировать и запустить программу в обоих виртуальных машинах. Для этого можно использовать как полностью консольный интерфейс (например, редактируя программу через терминал и редакторы vim или nano), так и графические среды разработки, включающие в себя кросс-компиляторы.

### **1.3.2. Пример выполнения задания на защиту**

Напишите программу, которая считывает общий объем оперативной памяти компьютера.

#include <stdio.h>

int main(void)

{

FILE \*meminfo = fopen("/proc/meminfo", "r");

char line[256];

while(fgets(line, sizeof(line), meminfo))

{

int ram;

if(sscanf(line, "MemTotal: %d kB", &ram) == 1)

{

fclose(meminfo);

printf(“%d”, ram);

}

}

fclose(meminfo);

return 0;

}

## **1.4. Вопросы для контроля**

1. Как в Syntacore Kit реализуется возможность запуска программ для архитектуры RISC-V?
2. В чем состоит назначение qemu-riscv64?
3. Как узнать размер кэш-памяти для одного ядра процессора в Linux?

# **Лабораторная работа № 2.** КОМПИЛЯЦИЯ И ЗАПУСК АССЕМБЛЕРНОЙ ПРОГРАММЫ

## **2.1 Цель и задачи**

Целью работы является освоение способов компиляции программ на ассемблере RISC-V.

Для достижения поставленной цели требуется решить следующие задачи:

1) Изучить примеры простейших ассемблерных программ.

2) Ознакомиться с основными командами для компиляции ассемблерных программ.

3) На примере HelloWorld приложений изучить основные структурные элементы ассемблерной программы.

## **2.2. Основные теоретические сведения**

Язык ассемблера представляет максимально близкий к аппаратному обеспечению способ для создания прикладных и системных программ. В таких программах практически отсутствуют привычные абстракции и программист вынужден оперировать реальными аппаратными структурами процессора и ОЗУ, а также учитывать особенности аппаратной архитектуры конкретной системы. Это приводит к тому, что языки ассемблера для разных процессорных архитектур не совместимы между собой, однако, общая логика построения программ и манипуляций с данными сохраняется. Ниже мы поясним необходимые элементы простейшей ассемблерной программы, подробное введение в ассемблер для RISC-V дается в книге [4], синтаксис команд описан в [5].

Ассемблерные программы представляют собой последовательность простейших команд для процессора, которые оперируют минимальными элементами данных и простейшими операциями с ними или другими устройствами. Для создания базового комфорта программисту, в большинстве языков ассемблера (в том числе и для RISC-V) предусмотрены такие структурные единицы как **псевдооператоры** и **метки**:

* Псевдооператоры позволяют разделить программу по секциям на статические данные (строковые и числовые константы) и исполняемый код, а также задать точку входа в программу (строку, с которой начинается выполнение программы). Для нашего примера нам потребуется две секции — **.global** (задает точку входа) и **.data** (задает секцию данных). Это не единственное назначение псевдооператоров, подробнее можно прочитать в [5].
* Метки помогают разделить программу на элементы, которые в ближайшем приближении напоминают функции в языке Си. Метки представляют собой именованные позиции в рамках исходного кода, к которым можно выполнить различные виды переходов в программе. Самый лучший аналог меток из языка Си это оператор GOTO [6], однако в отличии от Си, в ассемблере можно выполнять и условные переходы по меткам.

### **2.2.1 Структура ассемблерных программ**

В отличии от языков программирования высокого уровня, в ассемблере вместо переменных используются регистры – небольшие ячейки памяти, расположенные на микросхеме процессора. В каждой архитектуре процессора свой ограниченный набор регистров. Малый объем памяти, отводимый на регистры, компенсируется высокой скоростью работы с ними — она превосходит аналогичный показатель для ОЗУ и ПЗУ. Регистры могут иметь специальное назначение — так в примере простейшей программы ниже мы будем использовать четыре регистра (**a0, a1, a2, a7**), назначение которых подразумевает передачу аргументов для вызываемой функции системного вызова. В остальном работа с регистрами с некоторыми оговорками похожа на работу с переменными в языках программирования высокого уровня.

Управляющие инструкции в языке ассемблера выглядят очень просто по сравнению с командами и ключевыми словами языков высокого уровня. Как правило, команда представляет собой сокращение полного английского наименования совершаемого действия:

* la — Load Address (загрузить адрес в регистр),
* addi — Add an Immediate value to register (сложить константу со значением из регистра),
* ecall — Environment Call (произвести вызов системной процедуры). В случае ОС Linux (в данном случае RISC-V ОС) это будет означать вызов функции стандартной библиотеки.

Команды могут иметь от нуля до трех операндов (аргументов) для базового набора инструкций. В расширениях RISC-V может встречаться и большее число операндов.

* la — два операнда (регистр назначения и адрес области данных),
* addi — три операнда (регистр назначения для результата сложения, слагаемое №1, слагаемое №2) ,
* ecall — без операндов.

Простейший helloworld на ассемблере для RISC-V будет выглядеть так:

.global \_start # Точка входа для программы

# Настройка вызова функции write так, чтобы он вывел строчку в терминал

# В регистры a0-a2 записываются аргументы write

# В регистре a7 указывается номер write

\_start: addi a0, x0, 1 # 1 = Поток стандартного выовда (StdOut)

la a1, helloworld # Загрузка адреса контсанты helloworld в a1

addi a2, x0, 13 # Запись длины строки в a2

addi a7, x0, 64 # Запись номера функции write 64

ecall # Выполнение функции write

# Настройка вызова exit и вызов функции

addi a0, x0, 0 # Указание кода возврата для exit

addi a7, x0, 93 # Запись номера функции exit 93

ecall # Выполнение функции exit

.data

helloworld: .ascii "Hello World!\n"

Необходимо отдельно рассказать о разнице между системным вызовом exit и одноименной функцией стандартной библиотеки:

* Системный вызов только выходит из текущего потока управления, т.е. нити. За завершение процесса отвечает системный вызов exit\_group.
* В стандартной библиотеке функции exit(), \_Exit(), \_exit() используют именно вызов exit\_group.

### **2.2.2 Компиляция ассемблерных программ**

Для компиляции и запуска нам потребуется выполнить следующие команды в терминале RISC-V ОС (предполагаем, что исходный код сохранен в файле hello.s):

$ as -o hello.o hello.s # Запуск компилятора ассемблерных программ

$ ld -o hello hello.o # Запуск линковщика

$ ./hello

В случае успеха, в терминал будет выведена строка “Hello World!”.

Аналогичные операции можно проделать и из гостевой ОС, выполняя кросс-компиляцию, при этом потребуется использовать утилиты riscv64-linux-gnu-as и riscv64-linux-gnu-ld соответственно.

Для исследования скомпилированных ассемблерных программ удобно использовать утилиту objdump. Она позволяет выполнить обратное преобразование — из бинарного файла получить код на языке ассемблера:

$ objdump -d ./hello

Флаг -d в данном случае обозначает “дизассемблировать секции файла, в которых ожидается наличие кода“. Результат выполнения:

./hello: file format elf64-littleriscv

Disassembly of section .text:

00000000000100e8 <\_start>:

100e8: 00100513 li a0,1

100ec: 00001597 auipc a1,0x1

100f0: 02058593 addi a1,a1,32 # 1110c <\_\_DATA\_BEGIN\_\_>

100f4: 00d00613 li a2,13

100f8: 04000893 li a7,64

100fc: 00000073 ecall

10100: 00000513 li a0,0

10104: 05d00893 li a7,93

10108: 00000073 ecall

Вывод objdump имеет следующую структуру:

hex-код операции код операции ассемблерный код

В структуре восстановленного ассемблерного кода угадывается структура исходной программы, однако очевидно, что операции указанные нами изначально были преобразованы компилятором.

## **2.3. Задание к лабораторной работе**

Подготовьте программу, выполняющую с помощью языка ассемблера следующую задачу:

1. Вывести строку “Hello, I am <ваше ФИО> from group <номер вашей группы>”.
2. Вывести с помощью псевдографики изображение фигуры (размер фигуры по одной из сторон — 3 строки), составленное по следующему алгоритму:
   1. Если у вас четный номер в списке — вы выводите изображение квадрата, если нечетный — равносторонний прямоугольный треугольник.
   2. Символы, из которых необходимо составить фигуру, определяются остатком от деления последней цифры в номере вашей группы на 4:
      1. 0 - @
      2. 1 - #
      3. 2 - $
      4. 3 - %
3. Завершить работу программы.

Для вывода изображения используйте несколько последовательных вызовов write.

Программу необходимо скомпилировать и убедится в ее работоспособности на RISC-V ОС. С помощью утилиты objdump (в RISC-V ОС) необходимо дизассемблировать бинарный файл программы и зафиксировать в отчете общее количество строк, а также hex-код последней команды.

### **2.3.1. Описание последовательности выполнения работы**

Для выполнения поставленной задачи необходимо понять, как выглядит текст, выводимый вашей программой и фигура, которую вам необходимо нарисовать в вашей программе. Для определенности будем считать, что это квадрат из символов \*, размером 2х2. Для наглядности лучше всего изобразить эту фигуру:

\*\*

\*\*

Далее необходимо составить программу, которая будет выводить указанные строки. Учитывая пример фигуры выше, всего программа должна вывести три строки. Запишите эти строки в секцию .data. Соответственно, необходимо сделать три вызова write.

Далее необходимо скомпилировать и запустить программу в RISC-V ОС. Для этого можно использовать утилиты as и ld, либо применить кросс-компиляцию и запуск из гостевой ОС. После компиляци и запуска, программу необходимо проанализровать через objdump.

### **2.3.2. Пример выполнения задания на защиту**

Напишите программу, которая выводит две строки “I am an example string\n” и “Another string example\n”.

.global \_start

\_start: addi a0, x0, 1

la a1, string1

addi a2, x0, 25

addi a7, x0, 64

ecall

addi a0, x0, 1

la a1, string2

addi a2, x0, 25

addi a7, x0, 64

ecall

addi a0, x0, 0

addi a7, x0, 93

ecall

.data

string1: .ascii "I am example string\n"

string2: .ascii "Another string example\n"

## **2.4. Вопросы для контроля**

1. Какую задачу решает утилита ld из пакета binutils?
2. Какие задачи решают команды ассемблера li и auipc ?
3. Для чего предназначена секция .global?
4. Что такое метки?

# **Лабораторная работа № 3** ИЗУЧЕНИЕ АРХИТЕКТУРЫ RISC**-**V НА ПРИМЕРЕ АССЕМБЛЕРНОЙ ПРОГРАММЫ

## **3.1** **Цель и задачи**

Целью работы является получить представление о наборе команд RISC-V (базовый instruction set architecture (ISA), основные типы инструкций)

Для достижения поставленной цели требуется решить следующие задачи:

1) Ознакомиться с основными компонентами компьютера.

2) Понять взаимосвязь между SW и HW процессора (ISA – мост между SW и HW).

3) Понимать взаимосвязь между процессором и языком ассемблера.

4) Знать основные этапы компиляции программы и место ассемблера.

5) Изучить классический RISC – конвейер.

6) Понимать принцип кодирования инструкции на языке ассемблера в машинный код.

## **3.2. Основные теоретические сведения**

Место RISC-V в современных архитектурах согласно системе команд:

* CISC: одна инструкция = много последовательных операций.
* RISC: одна инструкция = операция, ограниченная по времени выполнения.
* VLIW: одна инструкция = много параллельных операция (или одна большая).

Принципы RISC:

* отсутствие вычислительно сложных инструкций,
* фиксированная длина инструкции,
* большое количество регистров общего назначения,
* ограничения на работу непосредственно с оперативной памятью как с медленным устройством.

Особенности инструкций:

* отсутствие регистра флагов в RISC-V.
* формат из 3-х операндов.
* дополнительные функции отдельных инструкций (add вместо mov).
* базовый набор и стандартные расширения.
* наличие псевдоинструкций.

В архитектуре RISC-V имеется обязательное для реализации небольшое подмножество команд (набор инструкций I — Integer) и несколько стандартных опциональных расширений. В базовый набор входят инструкции условной и безусловной передачи управления/ветвления, минимальный набор арифметических/битовых операций на регистрах, операций с памятью (load/store), а также небольшое число служебных инструкций.

Операции ветвления не используют каких-либо общих флагов, как результатов ранее выполненных операций сравнения, а непосредственно сравнивают свои регистровые операнды. Базис операций сравнения минимален, а для поддержки комплементарных операций операнды просто меняются местами.

### **3.2.1 Базовый набор регистров и альтернативные имена в стандарте ABI (application binary interface)**

Базовое подмножество команд использует следующий набор регистров (Таблица 3.1): специальный регистр x0 (zero), 31 целочисленный регистр общего назначения (x1 — x31), регистр счётчика команд (PC, используется только косвенно), а также множество CSR (Control and Status Registers, может быть адресовано до 4096 CSR).

| Номер | Название | Описание |
| --- | --- | --- |
| x0 | zero | Константа нуля (zero register) |
| x1 | ra | Адрес возврата (return address) |
| x2 | sp | Указатель стека (stack pointer) |
| x3 | gp | Глобальный указатель (global pointer) |
| x4 | tp | Указатель потока (thread pointer) |
| x5-x7 | t0-t2 | Временные переменные (temporary registers) |
| x8 | s0 / fp | Сохраняемая переменная /Указатель фрейма стека (saved register / frame pointer) |
| x9 | s1 | Сохраняемая переменная (saved register) |
| x10-x11 | a0-a1 | Аргументы функций/Возвращаемые значения (function arguments / return values) |
| x12-x17 | a2-a7 | Аргументы функций (function arguments) |
| x18-x27 | s2-s11 | Сохраняемые переменные (saved registers) |
| x28-x31 | t3-t6 | Временные переменные (temporary registers) |

Таблица 3.1 Набор регистров RISC-V (регистровый файл).

Для лучшей читаемости кода инструкции ассемблера обычно используют специальные имена, например s1, но они также могут использовать номер регистра (например, x9 для регистра номер 9). В нулевом регистре x0 всегда хранится константа 0; попытка записать в него другое значение игнорируется. Регистры от s0 до s11 (регистры 8–9 и 18–27) и от t0 до t6 (регистры 5–7 и 28–31) используются для хранения переменных; ra и регистры от a0 до a7 служат для вызовов функций. Регистры 2–4 носят имена sp, gp и tp. Нет операций над частями регистров, нет каких-либо выделенных «регистровых пар».

Регистр gp (global pointer) представляет собой специальную константу, указывающую в блок данных, где хранятся константы и глобальные переменные. Значение данного регистра инициализируется в начале работы программы (это делает линковщик, когда компонует программу) смещением 0x800 от начала данной секции, что позволяет получать доступ к содержимому секции используя 12-битное смещение со знаком, что значительно упрощает доступ к константам и глобальным переменным. Изменять значение регистра gp самостоятельно не рекомендуется.

Архитектура RISC-V использует только little-endian модель (Рис. 3.1) — первый байт операнда в памяти соответствует наименее значащим битам значений регистрового операнда. RISC-V применяет память с побайтовой адресацией. Это значит, что каждый байт памяти имеет уникальный адрес. Поскольку 32-битное слово состоит из четырех 8-битных байтов, то адрес каждого слова (word address) кратен 4. Старший байт (most significant byte,MSB) находится слева, а младший байт (least significant byte, LSB) – справа.



Рис. 3.1 Формат little-endian.

Инструкции базового набора имеют длину 32 бита.Формат 32-битной машинной команды (признаки – младшие биты всегда «11» и 2-4 биты ≠ «111» ).

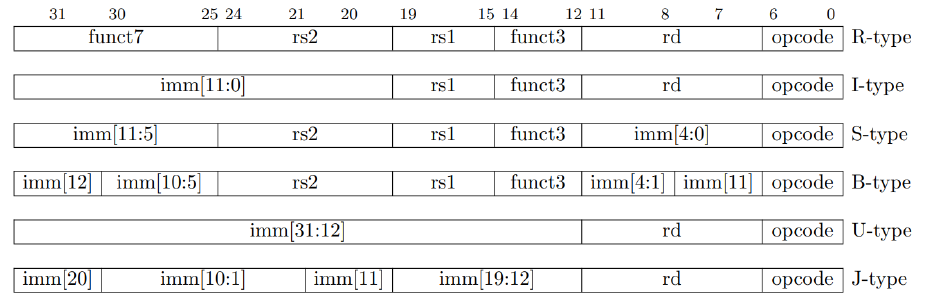


Рис. 3.2 Типы команд и их кодирование для RISC-V

Обозначения на Рис. 3.2:

* rs1 (5бит) — номер регистра в котором находится первый операнд
* rs2 (5бит) — номер регистра в котором находится второй операнд
* rd (5бит) — номер регистра в который будет записан результат
* opcode (7бит) +funct7+funct3 (10бит) определяют операцию
* imm – непосредственный операнд (константа) в дополнительном коде

Таким образом, в архитектуре RISC-V используются четыре формата инструкций: типа R, типа I, типа S/B и типа U/J.

## **3.3. Задание к лабораторной работе**

Напишите программу на ассемблере, которая пересылает константу 1 в регистр х1. К полученному результату прибавьте 1 и сохраните результат в регистре х1.

1. Скомпилируйте программу.
2. Представьте программу в машинном коде.
3. Выполните вручную декодирование программы в соответствии с типами используемых инструкций.
4. Повторите перечисленные выше действия, используя альтернативные имена регистров в стандарте ABI.

### **3.3.1. Описание последовательности выполнения работы**

Скомпилируйте исходную программу и убедитесь в ее работоспособности в RISC-V ОС, либо через qemu-riscv64.

С помощью утилиты objdump дизассемблируйте бинарный файл программы.

Используя информацию о том, как кодируются инструкции, выполните декодирование в виде таблицы.

### **3.3.2. Пример выполнения задания на защиту**

-----code-----

.text

.global \_st art

\_start:

addi x1, x0, 1

addi x1, x1, 1

addi a7, x0, 93 #set a7 = 93, Linux call exit

ecall

Disassembly of section .text:

00000000000100b0 <\_start>:

100b0**:** 00100093 addi x1, x0, 1

100b4: 00108093 addi x1, x1, 1

100b8: 05d00893addi a7, x0, 93

100bc: 00000073 ecall

-----end code-----

Инструкция addi относится к I-типу, opcode=0x13, funct3=0x0

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

## **3.4. Вопросы для контроля**

1.Что входит в базовый набор инструкций?

2.За что отвечают регистры a2-a7?

3. Поясните, как кодируются инструкции типа J.

# **Лабораторная работа №4** ЛОГИЧЕСКИЕ И АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ, ОПЕРАЦИИ СРАВНЕНИЯ И СДВИГА

## **4.1** **Цель и задачи**

Целью работы является разработка изучение логических и арифметическихопераций, операций сравнения и сдвига в рамках языка ассемблер RISC-V. Для этого необходимо выполнить следующие задачи:

1. Изучить логические и арифметические операции I-type.
2. Изучить логические и арифметические операции R-type.

## **4.2. Основные теоретические сведения**

### 4.2.1 Логические и арифметические операции I-type

| **Instruction** | **name** | **opcode** | **funct3** | **description** |
| --- | --- | --- | --- | --- |
| addi | ADD immediate | 0010011 | 0x0 | rd=rs1 + imm |
| xori | XOR immediate | 0010011 | 0x4 | rd=rs1 ^ imm |
| ori | OR immediate | 0010011 | 0x6 | rd=rs1 | imm |
| andi | AND immediate | 0010011 | 0x7 | rd=rs1 & imm |
| slli | Shift Left Logical Imm. | 0010011 | 0x1 | Imm(11:5)=0x00, rd=rs1<< imm(4:0) |
| srli | Shift Right Logical Imm. | 0010011 | 0x5 | Imm(11:5)=0x00, rd=rs1>> imm(4:0) |
| srai | Shift Right Arith Imm. | 0010011 | 0x5 | Imm(11:5)=0x20, rd=rs1>> imm(4:0) |
| stli | Set Less Than Imm. | 0010011 | 0x2 | rd=(rs1 < imm)?0:1 |
| sltiu | Set Less Than Imm. Unsigned | 0010011 | 0x3 | rd=(rs1 < imm)?0:1 |

Таблица 4.1 Операции I-type

Команды I-type (Таблица 4.1) принимают непосредственное значение imm (константу), исходный регистр rs1 и целевой регистр rd.

Поскольку инструкция **addi rd, rs1, imm** использует значение imm в качестве значения со знаком, она работает для арифметического сложения (imm = 0...2047) и вычитания (imm = -2048…-1), т.е. нет команды subi.

Инструкции **xori, ori** и **andi rd, rs1, imm** выполняют побитовую операцию xor, or, and соответственно. Например, пусть значение регистра x1 равно 0ⅹ9; в двоичной системе счисления (обозначается префиксом 0b) это 0b1001. Команда **xori x1, x1, 0ⅹ3** выполняет операцию xor над содержимым регистра x1, таким образом

x1 = x1 xor 0ⅹ3 = 0b1001 xor 0b0011 = 0b1010 = 0xa.

Инструкции **slli, srli и srai** выполняют операции сдвига, используя непосредственное значение в качестве количества сдвигов. Непосредственным значением является положительное число из 5 бит (0...31).

Логический сдвиг влево выполняется с помощью команды **slli rd, rs1, imm**. Она перемещает влево каждый бит значения исходного регистра rs1 на число бит, заданное непосредственным значением, и записывает результат в rd. При каждом сдвиге младший значащий бит устанавливается равным нулю, а старший значащий бит отбрасывается.

Логический сдвиг вправо выполняется аналогично с помощью **srli rd, rs1, imm**. Регистр rs1 побитово сдвигается вправо на immediate, и результат поступает в rd. Здесь самый старший бит заполняется нулем, а младший бит отбрасывается.

Обработка или заполнение старшего значащего бита отличается для арифметического сдвига вправо, выполняемого с помощью **srai rd, rs1, imm**. Самый старший бит (msb) сохраняется. Если msb равен 0 перед сдвигом, то msb заполняется значением 0 после сдвига. Если msb равен 1 перед сдвигом, то msb заполняется значением 1. Например, пусть значение регистра x1 равно 0ⅹ9, в двоичной системе счисления это 0b1001. Команда **slli x1, x1, 0ⅹ2** выполняет логический сдвиг содержимого регистра x1 влево таким образом

x1 = x1 << 0ⅹ2 = 0b1001 << 2 = 0b100100 = 0ⅹ24.

Оставшиеся две инструкции **slti rd, rs1, imm** и **sltiu rd, rs1,imm** сравнивают содержимое исходного регистра **rs1** с непосредственным значением (**imm**) и устанавливают в **rd** значение 0, если значение исходного регистра **rs1** < **imm**. Разница между обеими инструкциями заключается в том, что **slti** использует значения со знаком, и в сравнении используются значения со знаком. Примеры

addi x3, x1, 3 # x3 = x1 + 3

andi x3, x1, 3 # x3 = x1 & 3

slli x3, x1, 3 # x3 = x1 << 3

addi x3, x1, -3 # x3 = x1 - 3

### 4.2.2 Логические и арифметические операции R-type

Операции R-type (Таблица 4.2) включают арифметические и логические операции, которые используют два регистра в качестве источника и один регистр в качестве приемника. Большинство инструкций являются аналогом операций с непосредственными значениями.

| **Instruction** | **name** | **opcode** | **funct3** | **description** |
| --- | --- | --- | --- | --- |
| add | ADD | 0110011 | 0x0 | rd=rs1 + rs2 |
| sub | SUB | 0110011 | 0x0 | rd=rs1 - rs2 |
| xor | XOR | 0110011 | 0x4 | rd=rs1 ^ rs2 |
| or | OR | 0110011 | 0x6 | rd=rs1 | rs2 |
| and | AND | 0110011 | 0x7 | rd=rs1 & rs2 |
| sll | Shift Left Logical | 0110011 | 0x1 | rd=rs1<< rs2 |
| srl | Shift Right Logical | 0110011 | 0x5 | rd=rs1>> rs2 |
| sra | Shift Right Arith | 0110011 | 0x5 | rd=rs1>> rs2 |
| stl | Set Less Than | 0110011 | 0x2 | rd=(rs1 < rs2)?0:1 |
| sltu | Set Less Than Un | 0110011 | 0x3 | rd=(rs1 < rs2)?0:1 |

Таблица 4.2 Операции R-type.

Примеры

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

slt x3, x1, x2 # if x1 < x2 then x3 = 1 else x3 = 0

and x3, x1, x2 # x3 = x1 & x2

sll x3, x1, x2 # x3 = x1 << x2

Пример сложения I-type и R -type описан в Таблице 4.3.

| Адрес | Машинный код | Инструкция | Комментарий |
| --- | --- | --- | --- |
| 0x0 | 0x00100093 | addi x1, x0, 1 | # x1=1 |
| 0x4 | 0x00108093 | addi x1, x1, 1 | # x1=2 |
| 0x8 | 0x00108133 | add x2, x1, x1 | # x2=4 |

Таблица 4.3 Сложение операций.

Рассмотрим пример сложных вычислений a = ((b+3) >> c) - 1. Предположим, что a, b, c находятся в регистрах s2, s3 и s4 соответственно. Используйте регистр s5 для y0 и регистр s6 для y1.

addi s5, s3, 3 # y0 = b + 3

srl s6, s5, s4 # y1 = y0 >> c

addi s2, s6, -1 # a = y1 - 1

## **4.3. Задание к лабораторной работе**

1. Создать программу согласно варианту, которая

• выполняет заданные действия,

• выводит на экран сообщения о результатах/ошибках.

2. Откомпилировать программу и запустить на исполнение.

3. Отладить программу, проследить изменения в регистрах.

**Варианты заданий**

Напишите программы, которые вводят два целых значения x и y, вычисляют следующие выражения и выводят результат:

1. (x + 5) - (y - 7)
2. (x >> 2) + ((y - 1) << 3) (>> — логический сдвиг)
3. (x << y) - 10
4. (x >> y) + 10 (>> — арифметический сдвиг)
5. ((x << 2) - y + 5) >> 1 (>> — арифметический сдвиг)
6. x \* 6 - y \* 3 (выполнить умножение, используя сдвиг, сложение, вычитание)
7. 2 \* x \* x - 3 \* y + 4
8. (x + 5) / y + 10 / (y - 1)
9. (x / y) \* y + x % y
10. x & (-1 << 5)
11. x | (-1 >> 27) (>> — логический сдвиг)
12. Установите y-ий бит x в 1 (номера битов начинаются с 0)
13. Установите y-ий бит x в 0 (номера битов начинаются с 0)
14. x > y ? 0 : 1
15. (x == (y + 3)) ? 0 : 1
16. x < -5 & y > 10

### **4.3.1 Описание последовательности выполнения работы**

Скомпилируйте исходную программу и убедитесь в ее работоспособности в RISC-V ОС, либо через qemu-riscv64.

### **4.3.2 Пример выполнения задания на защиту**

Напишите программу, которая:

* очищает регистры t0, t1,t2 (использовать andi);
* загружает 0x1A352A9C и 0x1B2D4C6A соответственно в регистры t0, t1;
* добавляет в регистр t0 константу 0x1CB;
* складывает регистры t0, t1, результат помещает в t2.

Результат

andi t0, t0, 0 # Clear register t0

andi t1, t1, 0 # Clear register t1

andi t2, t2, 0 # Clear register t2

li t0, 0x1A352A9C # Loading register t0 with a value

li t1, 0x1B2D4C6A # Loading register t1 with a value

addi t0, t0, 0x1CB # Add t0 with an immediate value

add t2, t0, t1 # Add -- t0 with t1 and place the result in t2

## **4.4. Вопросы для контроля**

1. Как выполнить эти операции без использования инструкции умножения?

x \* 2

x \* 3

x \* 4

x \* 5

x \* 8

x \* 31

2. Как выполнить эти операции без использования инструкции деления?

x / 2

x / 3

x / 5

x / 8

x / 31

# **Лабораторная работа № 5** РЕЖИМЫ АДРЕСАЦИИ

## **5.1** **Цель и задачи**

Целью работы является разработка простой программы (программ) преобразования данных для приобретения практических навыков программирования на языке ассемблера и закрепления знаний по режимам адресации.

Для достижения поставленной цели требуется решить следующие задачи:

1.Изучить возможные способы адресации.

2.Ознакомиться с основными типами машинных команд RISC-V.

## **5.2. Основные теоретические сведения**

В данном практикуме адресация понимается как **способ доступа к данным в рамках той или иной управляющей инструкции**. Поскольку ассемблерные программы имеют мало выразительных синтаксических средств, то способы работы с данными могут существенно отличаться от программ на языке высокого уровня. В архитектуре RISC-V используются четыре режима адресации: регистровый, непосредственный, базовый и относительно счетчика команд. Первые три режима (регистровый, непосредственный и базовый) определяют способы чтения и записи операндов. Последний режим (относительно счетчика команд) определяет способ записи счетчика команд.

Далее каждый из способов адресации (доступа к данным) будет пояснен на примере операции add как одной из самых простых и интуитивно понятных. Эти объяснения обобщаются на все соответствующие типы инструкций.

### **5.2.1 Регистровая адресация**

При регистровой адресации (Рис. 5.1) регистры используются для всех операндов-источников и операндов-назначений (иными словами – для всех операндов и результата). **Все инструкции типа R используют именно такой режим адресации**.



Рис. 5.1 Регистровая адресация

Пример регистровой адресации:

add rd,rs1,rs2 # rd = rs1 + rs2

### **5.2.2 Непосредственная адресация**

При непосредственной адресации (Рис. 5.2) в качестве операндов наряду с регистрами используют константы (непосредственные операнды). **Этот режим адресации используют некоторые инструкции типа I**, такие как сложение с 12-битной константой (addi) и логическая операция andi.



Рис. 5.2 Непосредственная адресация

Пример непосредственной адресации:

addi rd,rs1,12 # rd = rs1 + 12

andi rd,rs1,-8 # rd = rs1 & 0xFF8

Чтобы использовать константы большего размера, используйте инструкцию непосредственной записи в старшие разряды lui (load upper immediate), за которой следует инструкция непосредственного сложения addi. Инструкция lui загружает 20-битное значение сразу в 20 старших битах и помещает нули в младшие биты:

lui s2, 0xABCDE # s2 = 0xABCDE000

addi s2, s2, 0x123 # s2 = 0xABCDE123

При использовании многоразрядных непосредственных операндов, если указанный в addi 12-битный непосредственный операнд отрицательный, старшая часть постоянного значения в lui должна быть увеличена на единицу. Помните, что знак addi расширяет 12-битное непосредственное значение, поэтому отрицательное непосредственное значение будет содержать все единицы в своих старших 20 битах. Поскольку в дополнительном коде все единицы означают число –1, добавление числа, у которого все разряды установлены в 1, к старшим разрядам непосредственного операнда приводит к вычитанию 1 из этого числа. Пример иллюстрирует ситуацию, когда мы хотим в s2 получить постоянное значение 0xFEEDA987:

lui s2, 0xFEEDB # s2 = 0xFEEDB000 (число, которое нужно записать в старшие 20 разрядов (0xFEEDA), предварительно увеличено на 1)

addi s2, s2, −1657 # s2 = 0xFEEDA987 (0x987 – это 12-битное представление числа -1657)(0xFEEDB000 + 0xFFFFF987 = 0xFEEDA987)

### **5.2.3 Базовая адресация**

Инструкции для доступа в память, такие как загрузка слова(чтение памяти) (lw) и сохранение слова(запись в память) (sw), используют базовую адресацию (Рис. 5.3). Эффективный адрес операнда в памяти вычисляется путем сложения базового адреса в регистре rs1 и 12-битного смещения с расширенным знаком, являющегося непосредственным операндом. Операции загрузки (lw) – это **инструкции типа I**, а операции сохранения (sw) – **инструкции типа S**.

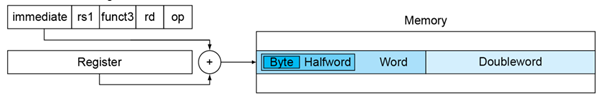


Рис. 5.3 Базовая адресация

Пример базовой адресации:

lw rd,36(rs1) # rd = M[rs1+imm][0:31]

Поле rs1 указывает на регистр, содержащий базовый адрес, а поле rd указывает на регистр-назначение. Поле imm, хранящее непосредственный операнд, содержит 12-битное смещение, равное 36. В результате регистр rd содержит значение из ячейки памяти rs1+36

sw rs2,8(rs1) # M[rs1+imm][0:31] = rs2[0:31]

Инструкция сохранения слова sw демонстрирует запись значения из регистра rs2 в слово памяти, расположенное по адресу rs1+8

### **5.2.4. Адресация относительно счетчика команд**

Инструкции условного перехода, или ветвления, используют адресацию относительно счетчика команд для определения нового значения счетчика команд (Рис. 5.4) в том случае, если нужно осуществить переход. Смещение со знаком прибавляется к счетчику команд (PC) для определения нового значения PC, поэтому тот адрес, куда будет осуществлен переход, называют адресом относительно счетчика команд.

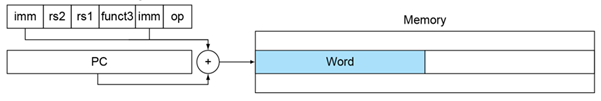


Рис. 5.4 Адресация относительно счетчика команд.

Инструкции перехода по условию (beq, bne, blt, bge, bltu, bgeu) **типа B** и jal (переход и связывание) **типа J** используют для смещения 13- и 21-битные константы со знаком соответственно. Самые старшие значимые биты смещения располагаются в 12- и 20-битных полях инструкций типа B и J. Наименьший значащий бит смещения всегда равен 0, поэтому он отсутствует в инструкции.

beq rs1,rs2,imm # if(rs1 == rs2) PC += imm

jal rd,imm #rd = PC+4; PC += imm

Инструкция jal может быть использована как для вызова функций, так и для простого безусловного перехода. Соглашение, используемое в RISC-V таково, что адрес возврата должен быть сохранён в адресе возврата ra ( x1).

Инструкция jal не имеет достаточного места для кодирования полного 32-битного адреса. Это означает, что вы не можете сделать переход куда-либо в коде, если ваша программа больше максимального значения смещения. Но если адрес перехода хранится в регистре, вы можете сделать переход на любой адрес (инструкция jalr типа I).

jalr rd,imm(rs1) # rd = PC + 4, PC = rs1 + imm

Большая разница состоит в том, что переход jalr не происходит относительно PC. Вместо этого он происходит относительно rs1. Для этого можно использовать addi для установки значения регистра rs1, но на практике поступают иначе. Для того, чтобы адрес был относителен к программному счётчику (РС) применяется специальная инструкция auipc. Инструкция auipc (Add Upper Immediate to Program Counter) типа U (сложить старшие разряды константы смещения с PC) также использует адресацию относительно счетчика команд.

auipc rd,imm # rd = PC + imm[31:12] << 12

Инструкция auipc является очень важной при работе с подпрограммами. Об этом подробнее будет рассказано в лабораторной работе №7.

## **5.3. Задание к лабораторной работе**

Напишите четыре программы на ассемблере, использующие различные режимы адресации.

1.Регистровая адресация. Сложить 2 константы n1=511 и n2=-10, заданные с помощью директивы .equ. Константы разместить в регистрах t0 и t1, а результат сложения в регистре t2.

2. Непосредственная адресация. Переслать константу n1=511 в регистр t0 и выполнить логическую операцию И с константой -8. Результат операции сохранить в регистре t1.

3.Базовая адресация. Сложить n целых чисел sum = a[0] + a[1] + a[2] + ... + a[n-1], результат записать в память.

Регистровый файл

| X1 | Addr of a[i] |
| --- | --- |
| X2 | n |
| X3 | sum |
| X10 | 100 |

Память

| 0 | a[0] |
| --- | --- |
| 4 | a[1] |
| …. | …. |
|  | a[n-1] |
|  |  |
| 100 | 0 |
| 104 | n |
| 108 | sum |

4. Адресация относительно счетчика команд. Найти c = max(a, b)

Регистровый файл до выполнения программы

| a0 | a |
| --- | --- |
| a1 | b |

Регистровый файл после выполнения программы

| a0 | c |
| --- | --- |

Каждую программу скомпилируйте программу и отладьте. Представьте результат в машинном коде.

В качестве варианта повышенной сложности, можно организовать работу всех четырех типов заданий в рамках одной программы.

### **5.3.1. Описание последовательности выполнения работы**

Скомпилируйте исходную программу и убедитесь в ее работоспособности в RISC-V ОС, либо через qemu-riscv64.

С помощью утилиты objdump дизассемблируйте бинарный файл программы

### **5.3.2. Пример выполнения задания на защиту**

.text

start:

.global \_start

n1 EQU 511

n2 EQU -10

# Регистровая адресация

addi t0, zero, n1

addi t1, zero, n2

add t2, t0, t1

# Непосредственная адресация

addi t0, zero, n1

andi t1, t0,-8

# Базовая адресация

# s10=100

lw s2, 0x0(s10)

lw s3, 0x4(s10)

add s4, zero, zero

loop:

lw s5, 0x0(s2)

add s4, s4, s5

addi s2, s2, 4

addi s3, s3, -1

bnez s3, loop

sw s4, 0x8(s10)

#Адресация относительно счетчика команд

max:

blt a0, a1, second # if a0 < a1 then a1 is larger

jal zero, done

second:

add a0, zero, a1 # make a1 the return value

done:

## **5.4. Вопросы для контроля**

1.Что такое регистровый файл?

2.В чем отличие регистров и основной памяти?

3. Объясните методы адресации, используемые в RISC-V?

4.Какие другие методы адресации Вы знаете? Почему в RISC-V они не используются

5.В чем отличие команд перехода jal и jalr?

# **Лабораторная работа № 6** АДРЕСАЦИЯ ЭЛЕМЕНТОВ МАССИВА, ОРГАНИЗАЦИЯ ЦИКЛА

## **6.1 Цель и задачи**

Целью работы является разработка программы преобразования данных для приобретения практических навыков программирования на языке ассемблера и закрепления знаний по работе с массивами и циклами. Задачи:

1. Изучить применение методов адресации для манипуляции массивами.
2. Освоить подходы к организации циклов средствами условных и безусловных переходов.

## **6.2. Основные теоретические сведения**

### 6.2.1 Хранение массивов в памяти

Архитектура RISC-V обычно использует **прямой порядок байтов** (от младшего к старшему; little-endian). RISC-V применяет память с побайтовой адресацией. Это значит, что каждый байт памяти имеет уникальный адрес. Поскольку 32-битное слово состоит из четырех 8-битных байтов, то адрес каждого слова (word address) кратен 4. Старший байт (most significant byte, MSB) находится слева, а самый младший байт (least significant byte, LSB) – справа.

Для обращения к элементу одномерного массива в команде используется формула для вычисления его адреса в памяти.

Адрес эл-та = (адрес нач. массива) + (№эл-та) × (размер эл-та в байтах)

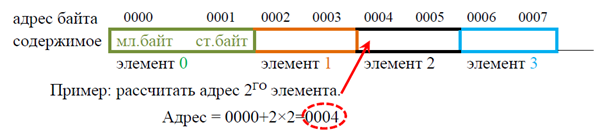


Рис. 6.1 Расчет адресов элементов массива.

Имя массива однозначно определяет адрес младшего байта его нулевого элемента. Начальный адрес массива и его размер (число элементов) заносятся в регистры (можно использовать любой свободный), и на каждой итерации указатель изменяется на длину элемента в байтах.

Обращение к элементу массива, к оперативной памяти в RISC-V организовано единственно возможным способом – с помощью базовой адресации со смещением:

la rs1, mas # Загрузить адрес массива

lw rd, 0(rs1)#Адрес элемента = Смещение (0) + База (rs1)

Пояснение к примеру:

* Смещение — адрес элемента внутри массива
* База — базовый адрес массива в сегменте данных

Другая форма:

li rs1,0

lw rd, mas (rs1) # Адрес элемента = База (mas) +Смещение (rs1)

### 6.2.2 Организация циклов

В языках высокого уровня циклы предназначены для многократного выполнения участка кода в зависимости от условия. В Ассемблере RISC-V нет специальных инструкций для организации цикла. Для перебора элементов массива в цикле требуется продумать самостоятельно вид цикла, регистр-счетчик количества повторений цикла, условие окончания, регистр для хранения флага окончания и временного индикатора для сравнения с индикатором. Проверка условий выполняется операторами «если-то-иначе», организованными командами условного перехода (beq, bne, blt, bge,bgeu, bltu) или безусловного перехода (jal, jalr и псевдокоманды j, jr ).

Команды условного перехода имеют формат типа B.



Рис. 6.2 Структура команд типа B.

В таблице 6.1 представлены все команды условного перехода

| **Instruction** | **name** | **opcode** | **funct3** | **description** |
| --- | --- | --- | --- | --- |
| beq | Branch == | 1100011 | 0x0 | if (rs1 == rs2) pc+= imm |
| bne | Branch != | 1100011 | 0x1 | if (rs1 != rs2) pc+= imm |
| blt | Branch < | 1100011 | 0x4 | if (rs1 < rs2) pc+= imm |
| bge | Branch >= | 1100011 | 0x5 | if (rs1 >= rs2) pc+= imm |
| bltu | Branch < Un | 1100011 | 0x6 | if (rs1 < rs2) pc+= imm |
| bgeu | Branch >= Un | 1100011 | 0x7 | if (rs1 >= rs2) pc+= imm |

Таблица 6.1 Команды условного перехода.

Команда условного перехода сравнивает значения регистров rs1 и rs2. Если результат сравнения (условие) является истинным, программный счетчик увеличивается на **imm**. Например, команда **blt rs1, rs2, imm** проверяет, меньше ли содержимое rs1, чем содержимое rs2. Если это так, то pc = pc + imm. В противном случае работа продолжается со следующей инструкцией в памяти (pc = pc +4).

Инструкции blt и bge используют значения со знаком; а bgeu этого не делает. Использование условного перехода bge и beq

bge s2, s3, else # if (a < b): c = a + 1

addi s4, s2, 1

beq zero, zero, end

else: # else: c = b + 2

addi s4, s3, 2

end: # s2=a; s3=b; s4=c;

Использование условного перехода **blt**

| Адрес | Машинный код | Инструкция | Комментарий |
| --- | --- | --- | --- |
| 0x00 | 0xffb00093 | addi s2, zero, -5 | s2 = -5 |
| 0x04 | 0x00500113 | addi s3, zero, 5 | s3 = 5 |
| 0x08 | 0x0020c463 | blt s2, s3, 0x8 | if (s2 < s3) pc = pc+ 8 |
| 0x0c | 0x00100193 | addi s4, zero, 1 | skipped if (s2 <s3): s4 = 1 |
| 0x10 | 0x00200193 | addi s4, zero, 2 | s4 = 2 |

Таблица 6.2 Пример использования условного перехода blt.

Инструкция по адресу 0x08 проверяет, является ли s2 < s3, то есть -5 < 5. Поскольку это верно, программный счетчик устанавливается на адрес 0x10, а инструкция по адресу 0x0c пропускается.

Если бы команда перехода по адресу 0x08 была **bltu**, отрицательное значение регистра s2 рассматривалось бы как положительное значение (0xfffffffb как положительное шестнадцатеричное значение для 32-разрядной версии), и результат сравнения был бы ложным.

Ниже приведен более сложный пример программы подсчета количества нулевых элементов в одномерном массиве.

.data

mas: .word -1 -2 -3 -4 1 2 0 4 2000 4096 65535 0

size: .word 12 # длина массива

head: .asciiz "\n Null element amount: "

.text

la t0, mas # загрузить адрес массива — начальный индекс [i]

lw t5, size # загрузить длину массива в регистр

loop:

lw t3, 0(t0) # поместить значение mas[i] в регистр $t3

bne 0, t3, met1 # перейти к met1 если $t3 не ‘0’ и элемент массива не ‘zero’

addi t4, t4, 1 # увеличить счетчик нулевых элементов

met1: addi t5, t5, -1 # уменьшить счетчик цикла

addi t0, t0, 4 # увеличить индекс текущего элемента

bge t5, loop # повторить, пока не закончили

la a0, head # загрузить адрес константы заголовка для печати

li v0, 4

ecall # напечатать заголовок

add a0, t4, zero # загрузить значение счетчика нулевых элементов для печати

li v0, 1

ecall # напечатать значение счетчика

Согласно вышеприведенной программе в памяти последовательно расположены 12 четырёхбайтовых элемента массива mas (среди которых есть два нулевых), затем переменная size, содержащая размер массива (12), затем ASCII-коды символов выводимого сообщения head. Далее следуют три инструкции загрузки:

1) адреса массива,

2) адреса переменной-размера массива,

3) значения переменной-размера массива.

Затем начинается цикл просмотра элементов (метка loop), в котором

• загружается очередной элемент массива mas[i],

• сравнивается с нулём и при необходимости увеличивается счетчик нулей,

• уменьшается счетчик цикла – количество оставшихся итераций цикла,

• увеличивается адрес элемента на его длину (4 Байта),

• проверяется условие окончания цикла (счетчик цикла равен нулю).

Далее выполняются два системных вызова

• для вывода строки-сообщения,

• для вывода переменной – счетчика нулей.

В ряде случаев полезными выступают команды безусловного перехода, являющиеся аналогами инструкции GOTO. Команды безусловного перехода имеют формат типа J или I. Для RISC-V есть две инструкции безусловного перехода (Таблица 6.3).

| **Instruction** | **name** | **format** | **opcode** | **funct3** | **description** |
| --- | --- | --- | --- | --- | --- |
| jal | Jump and Link | J | 1101111 | - | rd=PC+4; PC+=imm |
| jalr | Jump and Link Register | I | 1100111 | 0x0 | rd=PC+4; PC=rs1+imm |

Таблица 6.3 Команды безусловного перехода.

Команда **jal rd, imm** добавляет непосредственное значение к программному счетчику pc +=imm и записывает в регистр rd адрес после команды **jal**. Команда **jalr rd, rs, imm** устанавливает программный счетчик как результат pc = rs + imm, и записывает в rd обратный адрес. Обратный адрес — это инструкция по адресу, указанному после инструкции перехода.

| **Адрес** | **Машинный код** | **Инструкция** | **Комментарий** |
| --- | --- | --- | --- |
| 0x00 | 0x00c000ef | jal s2,0xc | pc = 0x0 + 0xc= 0xc, s2 = pc + 4 = 0x4 |
| 0x04 | 0x00000013 | addi zero, zero, 0 | no operation |
| 0x08 | 0x00010093 | addi s2, s3, 0 | s2= s3 |
| 0x0c | 0x00008167 | jalr s3, s2, 0 | pc = s2, s3= pc + 4 = 0x10 |
| 0x10 | 0x00100093 | addi s2, zero, 1 | s2= 1 |

Таблица 6.4 Пример использования безусловного перехода.

Последовательность выполнения программы выглядит следующим образом. Программа начинается с команды перехода **jal s2, 0xc** по адресу 0х00. Эта команда записывает обратный адрес в регистр2 и переходит к 0х0c. В 0х0c команда перехода **jalr s3, s2, 0** записывает свой соответствующий обратный адрес (0х10) в регистр s3 и выполняет переход (возвращает) к адресу, сохраненному в регистре s2, который равен 0х04. Программа продолжает выполнение инструкции в 0х04, которая ничего не делает. Следующая команда в 0х08 перемещает значение 0х10 (обратный адрес **jalr**) в регистр s2. После этого второй проход **jalr** в 0xc переходит к 0х10, значение регистра s2. Последовательность счетчика команд 0х00, 0х0c, 0х04, 0х08, 0х0c, и 0х10.

## **6.3. Задание к лабораторной работе**

1. Создать программу согласно варианту, которая

• выполняет заданные действия,

• выводит на экран сообщения о результатах/ошибках.

2. Откомпилировать программу и запустить на исполнение.

3. Отладить программу, проследить изменения в регистрах и оперативной памяти.

### **6.3.1 Описание последовательности выполнения работы**

Скомпилируйте исходную программу и убедитесь в ее работоспособности в RISC-V ОС, либо через qemu-riscv64.

### **6.3.2 Пример выполнения задания на защиту**

Напишите программу сложения целых чисел от 0 до 100:

* t0 используйте для текущего индекса "i" для цикла
* t1 хранит величину для сравнения с текущим индексом (в данном примере 100)
* t2 хранит текущую сумму (по окончанию программы — результат)

Код решения:

andi t0, t0, 0 # Clearing contents of register t0

andi t1, t1, 0 # Clearing contents of register t1

andi t2, t2, 0 # Clearing contents of register t2

# Functions as variable "sum"

li t1, 100 # Load t1 with value 100

loop: add t2, t2, t0 # Sum = Sum+i

addi t0, t0, 1 # Increment index "i"

blt t0, t1, loop # Iterate if t0<t1

End: j End # End of WHILE loop

**ВАРИАНТЫ ЗАДАНИЙ**

1.Найти минимальный и максимальный элемент массива А и запомнить их номера и адреса в переменных в памяти A\_min, A\_max. Вывести переменные, и предусмотреть случай (вывести сообщение), когда имеются все/несколько минимальных или максимальных элементов.

2. Для заданного массива А посчитать и запомнить количество нулевых/отрицательных/положительных элементов. Вывести сообщения с этими значениями и предусмотреть случай (вывести сообщение), когда какие-то типы элементов отсутствуют.

3. Найти элементы массива А, содержащие 2 единицы в двоичном представлении и запомнить их номера и адреса в новых одномерных массивах B\_nom и B\_adr, предусмотреть случай, когда таких элементов нет. Вывести сообщение о количестве таких элементов или об их отсутствии.

4. Для заданного массива А составить новый массив В, каждый элемент которого В[i] соответствует № первого слева (старшего) нулевого разряда элемента А[i]. Предусмотреть случай отсутствия нулевых разрядов элемента. Вывести сообщение с числом таких элементов или штатном режиме – весь массив В заполнен.

5. Для заданного массива А составить новый массив В, каждый элемент которого В[i] соответствует № первого справа (младшего) единичного разряда элемента А[i]. Предусмотреть случай отсутствия единичных разрядов элемента. Вывести сообщение с числом таких элементов или штатном режиме – весь массив В заполнен.

6. Для заданного массива А составить новый массив В, каждый элемент которого В[i] равен сумме единиц в двоичном представлении элемента А[i]. Посчитать общее количество единиц. Предусмотреть случай отсутствия единичных разрядов элементов. Вывести сообщение с общим количеством единиц или об их отсутствии.

7. Вычислить сумму всех отрицательных, положительных элементов массива А, число нулевых элементов, число четных и нечетных элементов и запомнить эти 5 значений в указанном порядке в одномерном массиве В. Вывести сообщения о полученных результатах, предусмотреть случай, когда некоторые виды элементов отсутствуют.

8. Вычислить сумму всех нулевых и единичных битов в двоичном представлении элементов массива А, запомнить эти значения в памяти и вывести эти значения или сообщение об отсутствии некоторых видов битов.

9. Для заданного массива А рассчитать разницу между максимальным и минимальным элементом и вывести это число. Предусмотреть случай их отсутствия (когда все элементы массива равны) и вывести об этом сообщение.

10. Вычислить сумму всех элементов массива А, больше заданной переменной. Вывести сообщения о полученных результатах, предусмотреть случай, когда такие элементы отсутствуют.

11. Для заданного массива А составить новый одномерный массив символьных строк В. Первым символом строки В[i] будет символ «Р» (positive), если элемент А[i] положительный и «N», если отрицательный. Вторым символом строки В[i] будет «–», третьим символом строки В[i] будет «Z», если элемент А[i] нулевой. Вывести сообщения об отсутствии одного из типов элементов и числе положительных элементов.

## **6.4. Вопросы для контроля**

1. У нас есть массив int (32 бита) с именем arr 0. Указатель на первый элемент arr0, хранится в регистре a1. Ответьте на следующие вопросы.

a. Как поместить четвертый элемент arr 0 в регистр t1?

b. Как рассчитать t1 + 16? Сохраните результат в регистре t2

c. Найдите эффективный способ вычисления t2 /16 и t2 %16. Сохраните результаты в t3 и t4 соответственно. Обратите внимание, что / — это целочисленное деление, а % — операция по модулю. (подсказка: использование сдвига и логических операций)

2. У нас есть массив int (32 бита) с именем arr1. Указатель на первый элемент arr1 хранится в регистре a2. У нас также есть регистры

t1 = 0xAAAAAAAA, t2 = 0xFEDCBA98

Ответьте на следующие вопросы:

a. Каково значение t3 для следующей последовательности инструкций?

slli t3, t1, 4

srli t3, t3, 4

b. Каково значение t3 для следующей последовательности инструкций?

slli t3, t2, 3

srai t3, t3, 3

c. Напишите фрагмент ассемблерной программы:

• Сохранить результат t1 & t2 в регистре t4;

• Сохранить t4 в первом элементе arr1;

• Сохраните младшие 8 бит из t4 во втором элементе arr1.

3. Рассмотрите следующие инструкции RISC-V

li t1, 0

li t2, 1

li t3, 1

li t4, 10

LOOP:

beq t1, t4, DONE

add t5, t2, t3

addi t2, t3, 0

addi t3, t5, 0

addi t1, t1, 1

jal x0, LOOP

DONE:

# end of the program

a. Сколько раз выполняется цикл (между LOOP и DONE)?

b. Перечислите значение t2 на каждой итерации цикла.

c. Перечислите значение t3 на каждой итерации цикла.

d. Что делает эта программа?

# Лабораторная работа №7 ВЫЗОВ ПОДПРОГРАММЫ, РАБОТА СО СТЕКОМ

## **7.1. Цель и задачи**

Целью работы является разработка простой программы преобразования данных для приобретения практических навыков программирования на языке ассемблера и закрепления знаний по работе с подпрограммами и стеком. Задачи:

1. Изучение соглашений о вызовах подпрограмм.
2. Освоение принципов работы со стеком.

## **7.2. Основные теоретические сведения**

### **7.2.1 Директивы размещения данных в памяти**

**Определения**

o TEXT — область для инструкций программы. Теоретически никто не мешает иметь несколько директив .text *адрес*, размещающих код по различным адресам

o DATA — область для всевозможных данных программы (глобальных переменных, статических локальных переменных, кучи и стека)

o extern base address — область для внешних данных (нужна для взаимодействия с ОС). В этой области размещает данные директива .extern

o .data base address — начало области, в которую обычно раскладываются данные директивами .data. Именно там лежат переменные, объявленные массивы и прочее. Традиционно имеется *зазор* между началом области данных и непосредственно статическими данными. Обычно в процессе работы программы *нельзя переходить по адресам из области данных и декодировать их как инструкции*.

В программе на языке ассемблера возникает необходимость описать содержимое сегмента памяти. Для этого код программы помечается .text, а данные — .data .

В секции .data помещают *директивы* (указания ассемблеру) по размещению данных в памяти.

* .word *число* — одно или несколько 4-байтовых чисел
* .dword *число* — одно или несколько 8-байтовых чисел
* .half *число* — одно или несколько 2-байтовых чисел
* .byte *число* — одно или несколько однобайтовых чисел
* .ascii "*строка*" — последовательность символов в кодировке ASCII
* .asciz "*строка*" — то же, только после последнего символа обязательно записывается нулевой байт (конец строки, договорённость, например, для языка Си). Пример размещения данных различного размера:

.data

.word 0xdeadbeef

.dword 0xacebad0feeded

.half 0x1234, 0x5678

.byte 12, 13, 14, 15

.half 0x3344

.byte 0x66, 0x77

Результат трансляции пословно (допустим секция .data начинается с адреса 0x10010000). Используется little endian: младший байт в слове имеет меньший адрес:

10010000: deafbeef d0feeded 000aceba 56781234 0f0e0d0c 77663344

### **7.2.2** Инструкция ecall

Ранее мы рассматривали инструкцию **ecall.** Остановимся подробнее на специфике ее работы. В общем случае ecall (сокращение от Environment Call) это механизм вызова **внешних** процедур от некоторой вышестоящей среды исполнения. Когда мы ведем разработку приложений для Linux-подобных ОС то сама ОС и является такой средой. В таком случае, ecall выступает механизмом выполнения вызовов ОС — специальных внешних процедур, реализуемых в ядре ОС [28].

Важно отметить, что в случае bare metal программирования на RISC-V (то есть работа с устройствами напрямую, без ОС) процедуры из ecall могут быть не поддержаны, так как будет отсутствовать необходимый уровень абстракции для его работы (ядро ОС).

Рассмотрим соглашение о вызовах **ecall (ABI)** на примере RISC-V ОС:

1. В регистр **a7** помещается номер системной функции (service number).

2. Если есть параметры системного вызова, то они помещаются в регистры **a0-a6**.

3. Инструкция **ecall** передаёт управление операционной системе (в данном случае — ядру RISC-V ОС).

4. Возврат из системного вызова происходит по аналогии с возвратом из подпрограммы, на следующую после **ecall** инструкцию

5. Возвращаемое значение (если есть) помещаются в **a0**.

**Пример**: вывести на консоль число, лежащее в регистре **t0**

li a7 64 # Функция 64 — вывод числа

mv a0 t0

ecall # Выведённое число помещается в a0

Для некоторых вызовов вам могут потребоваться строковые аргументы. С точки зрения ассемблера, строки — это последовательности ненулевых байтов, заканчивающихся символом с кодом 0 (однобайтным нулём) согласно формату ASCIZ. Ввод заканчивается символом перевода строки (ASCII-код 10, соответствует нажатию Enter). Этот символ тоже попадает в строку! Так что если требуется ввести слово не более, чем из 10 букв, размер буфера должен быть на 2 больше — 12. 11-м символом в нём будет '\n', а 12-м — 0.

Полный список поддерживаемых системных вызовов для RISC-V ОС можно найти в таблице [29] (необходимо найти колонку для архитектуры riscv64).

### **7.2.3 Подпрограммы**

Для разработки комплексных приложений может потребоваться не только применение условных переходов, но и разбиение исходного кода на повторно используемые фрагменты (подпрограммы). Подпрограмма — часть программного кода, оформленная таким образом, что

* возможно выполнение этого участка кода более, чем один раз
* переход на этот участок кода (*вызов подпрограммы*) возможен из произвольных мест кода
* после выполнения подпрограммы происходит переход «обратно» в место вызова (*выход из подпрограммы*)

В рамках ассемблера подпрограммы могут быть реализованы в качестве меток. Ниже представлен **упрощенный** пример программы, показывающий общую логику вызова подпрограмм:

.text

main:

# Подготовка к вызову функции - сохранение аргументов и

# сохранение адреса возврата

# Аргументы (первые 8) - в a0-a7

# Адрес возврата в ra (x1)

..

# Вызов функции - выполнение перехода к метке func

j func # Для упрощения мы использовали j, в реальности

# используется lalr или jar

# Продолжение работы программы

..

func:

# Код функции стартует с метки func

..

# Получение аргументов и действия с ними (если они подразумевались)

..

# Формирование возвращаемого значения в a0

..

# выход из функции - переход к адресу возврата

jalr zero, ra,0

Показанный выше пример является крайне упрощенным и не иллюстрирует всех сценариев разработки подпрограмм и организации кода. Для того, чтобы такой ценный и ограниченный ресурс как регистры использовался разумно и безопасно для исполняемого кода, в application binary interface (ABI) есть явный набор соглашений о том, как необходимо поступать со значениями регистров при вызовах и возврате из подпрограмм. Одно из соглашений заключается в сохранении значений регистра, когда команда перехода вызывает подпрограмму. Это означает, что обратный адрес инструкции после команды перехода в памяти сохраняется в регистре, а программный счетчик является началом подпрограммы. Подпрограмма состоит из инструкций, которые обрабатываются и возвращаются обратно к инструкции, следующей за ее вызовом. Для возврата подпрограмма использует сохраненный обратный адрес. В таблице 7.1 обобщены псевдонимы и соглашение о хранении, например, регистр x1 имеет псевдоним ra, который обозначает адрес возврата, и вызывающий функцию должен сохранить значение регистра x1 перед вызовом функции и восстановить его после возврата функции.

| **Номер** | **Название** | **Описание** | **Сохраняется?** |
| --- | --- | --- | --- |
| x0 | zero | Константа нуля (zero register) | *n/a* |
| x1 | ra | Адрес возврата (return address) | нет |
| x2 | sp | Указатель стека (stack pointer) | да |
| x3 | gp | Глобальный указатель (global pointer) | *n/a* |
| x4 | tp | Указатель потока (thread pointer) | *n/a* |
| x5-x7 | t0-t2 | Временные переменные (temporary registers) | нет |
| x8 | s0 / fp | Сохраняемая переменная /Указатель фрейма стека (saved register / frame pointer) | да |
| x9 | s1 | Сохраняемая переменная (saved register) | да |
| x10-x11 | a0-a1 | Аргументы функций/Возвращаемые значения (function arguments / return values) | нет |
| x12-x17 | a2-a7 | Аргументы функций (function arguments) | нет |
| x18-x27 | s2-s11 | Сохраняемые переменные (saved registers) | да |
| x28-x31 | t3-t6 | Временные переменные (temporary registers) | нет |

Таблица 7.1 Соглашение об использовании регистров

Как правило, сохраненные регистры s0-s11 сохраняются при вызовах функций, в то время как регистры аргументов a0-a7 и временные регистры t0-t6 — нет.

В RISC-V для передачи аргументов в подпрограмме используются 8 регистров аргументов, а именно a0-a7. Перед выполнением вызова подпрограммы аргументы подпрограммы копируются в регистры аргументов а0-а7. Стек используется в том случае, если количество аргументов превышает 8.

Для вызова подпрограммы используются команды jal (типа U) илиjalr (типа S). Эти команды работают позиционно-независимо — в одной (jal) чётко формируется смещение относительно адреса текущей инструкции (но этот переход ограничен 20-битным адресом), в другой (jalr) — полный адрес перехода формируется в s0 с помощью команды auipc (адрес 32-битный). auipc выполняет формирование адреса при помощи адреса текущей инструкции и корректирующего сложения. Таким образом, в зависимости от величины необходимого смещения относительно счетчика команд (PC) для вызова подпрограмм применяется или jal или комбинация auipc и jalr.

Пример, в котором подпрограмма сначала вызывается с помощью jal, а затем — с помощью jalr:

jal ra,0x00000018 # jal subr

auipc s1,0 # la s1 subr

addi s1, s1,20

jalr ra, s1,0 # jalr s1

addi x17, zero,10 # li a7 10

ecall # ecall

auipc a0,0x0000fc10 # subr: la a0 ping

addi a0, a0,0xffffffe8

addi a7, zero,4 # li a7 4

ecall # ecall

jalr zero, ra,0 # ret

Примечание: справа в комментариях приведен текст с использованием псевдоинструкций.

В качестве команды возврата используется jalr**.** Как это работает:

* в качестве регистра перехода используется ra
* в качестве регистра возврата используется zero (это позволяет разработчику сразу определять, что перед нами инструкция не перехода, а именно возврата из подпрограммы)

### 7.2.4 Стек

Для динамического хранения локальных переменных и адресов возврата нужен стек. Стек регулируется соответствующим соглашением ABI:

* выделенный регистр **sp** (**x2**)
* определено дно стека и его начальное значение
* начальное значение отделено от дна буфером
* стек растёт *вниз* по одному слову (4 байта)
* операции добавления и снятия:
  + при добавлении сначала уменьшается указатель, затем записывается значение
  + при снятии сначала считывается значение, затем увеличивается указатель
  + при такой организации не используется исходная ячейка стека (начальное значение)

Пример

li t1 123 # какое-то значение

addi sp sp -4 # положить на стек - 1

sw t1 (sp) # положить на стек - 2

addi t2 t1 100 # какое-то ещё значение

addi sp sp -4 # положить на стек - 1

sw t2 (sp) # положить на стек - 2

lw t3 (sp) # доступ к вершине стека

lw t4 4(sp) # доступ ко 2-му элементу стека

lw t0 (sp) # снятие со стека - 1

addi sp sp 4 # снятие со стека - 2

addi sp sp -4 # положить на стек - 1

sw zero (sp) # положить на стек – 2

Особенности хранения данных в стеке

* Несколько более эффективно, чем в произвольном месте памяти (lw/sw не превращаются в псевдоинструкции).
* Использует адресацию относительно постоянно меняющегося sp. Как следствие, требует аккуратного подсчёта текущей глубины стека.
* Не требует явного указания адреса и заведения метки в программе на языке ассемблера.
* Может привести к сбоям в работе при переполнении/исчерпании/неаккуратном использовании стека.

## **7.3 Задание к лабораторной работе**

1. Создать программу согласно варианту, которая

• выполняет заданные действия,

• выводит на экран сообщения о результатах/ошибках.

2. Откомпилировать программу и запустить на исполнение.

3. Отладить программу, проследить изменения в регистрах и ОП.

### **7.3.1 Описание последовательности выполнения работы**

Скомпилируйте исходную программу и убедитесь в ее работоспособности в RISC-V ОС, либо через qemu-riscv64.

### **7.3.2** **Пример выполнения задания на защиту**

Напишите программу, которая вычисляет наибольший общий делитель двух значений с помощью алгоритма Евклида:

function gcd(a, b)

while a ≠ b

if a > b

a := a − b

else

b := b − a

return a

Фрагмент решения:

main:

# read\_int (t0)

# read\_int (t1)

mv a0, t0

mv a1, t1

jal euclid

li a7, 1

ecall

li a7, 10

ecall

euclid:

beq a0, a1, finish

blt a0, a1, if\_less

sub a0, a0, a1

j euclid

if\_less:

sub a1, a1, a0

j euclid

finish:

jr ra

**ВАРИАНТЫ ЗАДАНИЙ**

1.Напишите программу, которая вводит целые числа M и N и выводит сетку M x N, составленную с помощью + и -. Вы должны написать подпрограмму, который принимает два параметра: M и N, и выводит строку, подобную этой: +-+-+-+-+

Input:

3

4

Output:

+-+-+-+

| | | |

+-+-+-+

| | | |

+-+-+-+

| | | |

+-+-+-+

| | | |

+-+-+-+

2. Написать функцию, которая меняет местами два значения, переданные в стек.

3. Написать функцию, которая находит минимальное и максимальное значение в заданном массиве целых чисел.

## **7.4** **Вопросы для контроля**

1.В чем состоит основная идея использования подпрограмм?

2. Какими командами можно реализовать вызов подпрограммы в RISC-V?

3.Для чего используется стек?

# Лабораторная работа №8 ОПТИМИЗАЦИИ НА ЭТАПЕ КОМПИЛЯЦИИ

## **8.1 Цель и задачи**

Целью работы является освоение способов оптимизации программ на этапе компиляции для RISC-V.

Для достижения поставленной цели требуется решить следующие задачи:

1) Изучить флаги компиляции для оптимизации программ на С.

2) Ознакомиться с основными командами для декомпиляции программ.

3) На примере HelloWorld приложений изучить разницу при использовании разных типов оптимизаций, а также ограничений, которые они могут потенциально накладывать.

## **8.2. Основные теоретические сведения**

Работа программиста, использующего языки высокого уровня, часто сопряжена с определенным уровнем противоречий. С одной стороны, разработчик должен писать легко читаемые и легко поддерживаемые программы, что выражается в использовании высокоуровневых синтаксических конструкций. С другой стороны, программа должна эффективно работать с учетом конкретного аппаратного обеспечения пользовательского компьютера, что требует учета низкоуровневых особенностей исполнения. Данное противоречие частично разрешается компиляторами — они не просто выполняют переход от языков высокого уровня к машинным инструкциям, но и стараются выбрать наиболее эффективный способ данного перехода. Однако, в ряде случаев, им требуется дополнительная настройка для достижения максимального эффекта. В таком случае применяются методы оптимизации на уровне компилятора и техники инструментирования. Данные инструменты регулируются флагами компилятора. Для определенности, далее мы будем рассматривать тему применительно к компилятору GCC, однако подобные инструменты также есть и в компиляторе LLVM.

### **8.2.1 Оптимизации на уровне компилятора**

**Оптимизации на уровне компилятора** представляют собой дополнительные процедуры по выбору оптимального преобразования исходного кода программы в машинные инструкции так, чтобы достигнуть максимального значения параметра производительности (время работы и/или объем исполняемого файла). При этом, в зависимости от выбранной интенсивности, компилятор может применять достаточно агрессивные эвристики для оптимизации, что может привести к ошибкам и сбоям в работе программы (например, многопоточные ошибки и ошибки обращения с памятью). Наиболее важные флаги оптимизации для GCC [10]:

* Время работы программы
  + **-O0** — тривиальный случай, когда оптимизация компилятора не выполняется. Тем не менее оптимизация для конкретного языка в соответствии с требованиями стандарта по-прежнему выполняется. Например, вычисления во время компиляции, требуемые стандартом C++, по-прежнему выполняются.
  + **-O1** — на этом уровне включается множество оптимизаций, повышающих производительность программы. Например, развертывание циклов (loop unrolling), встраивание функций (inline functions), планирование инструкций (instruction planning) и т. д.
  + **-O2** позволяет использовать все оптимизации -O1, а также более агрессивные оптимизации в распределении регистров, планировании инструкций, частичном устранении избыточности и т. д. Этот уровень используется при построении кода с преобладанием ветвлений, например, операционных систем.
  + **-O3**. Этот уровень включает в себя все возможности -O2, а также некоторые современные оптимизации, такие как векторизация. -O3 является уровнем оптимизации для максимизации производительности большинства приложений.
  + **-Ofast** или -O3 с -ffast-math. Флаг -ffast-math указывает компилятору ослабить некоторые требования арифметики с плавающей запятой, такие как ассоциативность и коммутативность. Во многих приложениях ошибки, возникающие после ослабления этих требований, вполне допустимы за счёт более высокой производительности. Без -ffast-math многие циклы с операциями с плавающей точкой не могут быть векторизованы.
* Размер бинарного файла
  + **-Os** оптимизирует размер кода. Таким образом, большинство оптимизаций, увеличивающих размер кода, будут менее агрессивными на этом уровне. Это популярная оптимизация среди встраиваемых систем и мобильных приложений, поскольку размер кода там является большой проблемой.
  + **-Wl,--strip-all** (или не передавать флаг -g): этот флаг указывает компоновщику удалить раздел отладки.
  + **-fno-unroll-loops**: отключает развертывание цикла, которое является одной из популярных оптимизаций производительности компилятора, увеличивающей размер кода.
  + **-fno-exceptions**: удаляет код обработки исключений из двоичного файла. Обратите внимание, что это не всегда возможно, если есть код, который их «бросает».
  + **-lto (-flto)**: включение оптимизации времени компоновки с параметром -flto приводит к агрессивной оптимизации компилятора. Оптимизируются многие функции и глобальные переменные, девиртуализируются многие вызовы. Полученный двоичный файл быстрее и меньше одновременно. Могут быть значительные накладные расходы во время компиляции.

### 8.2.2 Инструментирование и профилирование

Описанные выше флаги для оптимизации не являются панацеей — они действуют на всю программу целиком и не учитывают особенностей ее использования, продиктованных действиями пользователя и структурой взаимодействия с ним. Более эффективные оптимизации могут быть достигнуты применением **инструментирования** — методик, внедряющих дополнительный код в бинарный файл для сбора технической информации о производительности (профиля работы) программы в ходе ее работы. Данная информация может быть передана компилятору чтобы повысить эффективность оптимизаций. Для инструментирования применяются следующие флаги [10]:

* **-g** . Чтобы иметь возможность отлаживать приложение с аннотациями исходного кода, компилятор должен предоставить дополнительную информацию в двоичном файле. Флаг -g указывает компилятору сделать это. Без этого флага отладчик будет показывать только имена глобальных символов и дизассемблер, поскольку он не может связать строку исходного кода со сборкой.
* **-finstrument-functions** . Этот флаг используется для инструментирования входа и выхода функций. Инструментирование позволяет получить представление о поведении программ. При использовании этого флага также необходимо определить две функции \_\_cyg\_profile\_func\_enter и \_\_cyg\_profile\_func\_exit, которые вызываются соответственно при входе и выходе из каждой вызываемой функции. Если есть функции, которые не должны быть инструментированы, к ним можно добавить \_\_attribute\_\_ ((no\_instrument\_function)).
* **-fprofile-generate, -fprofile-arcs, -pg** .Эти флаги используются для инструментирования программ с целью сбора профилей времени выполнения различных точек программы. Это позволяет компилятору проводить оптимизацию с учетом профиля в последующих компиляциях. В зависимости от того, какие флаги вы используете, могут быть достигнуты различные типы инструментирования. Подробный обзор различных флагов приведен на странице руководства [11].
* **-fstack-protector, -fstack-protector-all, -fstack-protector-strong** . Эти опции инструментируют уязвимые функции путем вставки защитных переменных в кадр стека. Перед возвратом функции проверяется, что защитная переменная не была перезаписана, что позволяет убедиться в том, что стек не был поврежден. Это тривиальный способ улучшить защиту от атаки на переполнение буфера. Однако это может увеличить размер кода приложения. В случае, если это создает накладные расходы, с этим флагом можно компилировать только критически важные для безопасности части приложения.

Рассмотрим пример использования инструментирования для сбора профиля работы программы. Пример приведен для RISC-V ОС, однако его можно повторить и в гостевой ОС, используя команды для кросс-компиляции. Первое действие это компиляция с флагом -fprofile-generate:

$ gcc -O2 **-fprofile-generate**=/path/to/outputfile test.c -o a.out

Далее необходимо запустить программу (a.out) несколько раз для сбора профиля:

$ ./a.out

Профиль работы программы в виде файла с расширением .gcda будет сохранен в каталог /path/to/outputfile. После этого можно запустить компиляцию исходного кода с флагом -fprofile-use для дополнительной оптимизации:

$ gcc -O2 **-fprofile-use**=/path/to/outputfile test.c -o a.out

Важно понимать, что с одной стороны, инструментирование позволяет собрать больше информации о работе ПО, но с другой — за счет внедрения инструментального кода сам процесс измерения может повлиять на измеряемые величины или даже привести к сбоям в работе ПО. Для получения более детальной картины возможно применение внешних инструментов измерения производительности — профилировщиков, например perf [12], valgrind [13], callgrind [14], cachegrind [15]. Такие профилировщики позволяют собрать информацию о самых низкоуровневых особенностях работы программы — эффективность работы кэш-памяти, качество предсказаний переходов, статистика работы с динамической памятью.

### 8.2.4 Измерение ускорения от оптимизаций

Для того, чтобы убедится в эффективности проводимых оптимизаций, необходимо использовать инструменты для измерения скорости работы программы, а также объема бинарных файлов. Выбор конкретных утилит, а также технологий измерения зависит от множества различных факторов. Например, для измерения времени работы программы:

* структура сценария использования (как пользователь взаимодействует с программой — интерактивно или нет),
* наличие и структура входных данных (что и как пользователь вводит в программу),
* целевая ОС и аппаратное обеспечение (в каких условиях предполагается работа программы),
* тип программы (встраиваемое ПО, веб-приложение, утилита командной строки и тд),
* факторы, влияющие на измерения (наличие фоновых процессов, сетевой трафик),
* требуемая точность измерений.

Данные вопросы подробно изучаются в дисциплине “Метрология ПО”. В данной работе мы предлагаем грубый и быстрый способ оценки эффекта ускорения работы ПО от оптимизаций.

Для **неответственных** измерений времени работы программы в ОС на базе Linux можно использовать утилиту time [16]. Она измеряет время работы программы с момента запуска и до завершения, а также выводит детализацию. Для измерения времени работы скомпилированной ранее программы используйте команду:

$ time ./a.out

Вывод имеет вид:

# Вывод программы a.out

…

real 0m0,020s

user 0m0,005s

sys 0m0,015s

где real — общее время работы программы, user — время работы программы в режиме пользователя [17], sys — время работы программы в режиме ядра [17] (использование системных вызовов и низкоуровневых операций ОС).

Важно отметить, что единичное измерение не обладает ценностью — для оценки производительности важны статистические (множественные наблюдения), проводимые в воспроизводимых условиях. Поэтому, необходимо сделать множество измерений в одинаковых условиях. Также, для оценки эффекта необходимо проводить измерения базового уровня (baseline) — показателей программы до оптимизации. Таким образом, примерный алгоритм измерений будет иметь вид:

* Сохраните исходный бинарный файл программы (baseline).
* Проведите оптимизацию, сохраните отдельно модифицированный бинарный файл программы (optimized).
* Подготовка среды для измерений, исключение помех (закрытие ресурсоемких приложений, подготовка тестовых данных).
* Проведение N запусков для baseline.
* Проведение N запусков для optimized.
* проведите статистическую обработку (подсчет среднего, медианы, минимума, максимума, стандартного отклонения).

Для автоматизации процесса измерений вы можете использовать уже готовую утилиту multitime — она дублирует функции time, а также умеет подводить статистику нескольких последовательных запусков. Например, команда для проведения десяти запусков и вычисления статистики:

$ multitime -n 10 ./a.out

Результат выполнения:

# Вывод программы a.out

…

===> multitime results

1: ./a.out

Mean Std.Dev. Min Median Max

real 0.016 0.002 0.012 0.016 0.021

user 0.007 0.005 0.000 0.007 0.013

sys 0.008 0.004 0.004 0.007 0.015

По умолчанию, multitime может быть не установлена в популярных Linux дистрибутивах (в том числе и в ОС Syntacore Kit), вы можете установить ее командами

$ apt update

$ apt install multitime

### 8.2.5 Измерение уменьшения объема бинарного файла от оптимизаций

Измерение эффекта от соответствующих оптимизаций аналогично описанному ранее алгоритму, за исключением того, что достаточно провести одно измерение объема файла. Для измерения самого объема бинарного файла можно использовать различные утилиты. Утилита командной строки ls позволяет понять общий размер файла в байтах без детализации о распределении объема:

$ ls -l a.out

-rw-r--r-- 1 root root **13423** сен 26 2023 a.out

Значение в пятой колонке соответствует размеру файла в байтах. О значениях в прочих колонках можно подробнее узнать в справке [18].

Для более точной оценки можно использовать утилиту size [19]. Она отображает размер каждой секции в рамках бинарного файла:

$ size a.out

text data bss dec hex filename

1524 600 8 2132 854 a.out

## **8.3. Задание к лабораторной работе**

Подготовьте исходный код программы на языке С, которая будет выполнять заданное вариантом действие со случайной целочисленной матрицей А размера 20 на 20. Необходимо самостоятельно скомпилировать программу с различными опциями для архитектуры RISC-V, а также убедится в работоспособности. Опции для компиляции:

1. Сборка без оптимизации.
2. Оптимизация размера бинарного файла.
3. Оптимизация скорости работы.

С помощью флагов компилятора, постарайтесь добится наилучших характеристик для программ пунктов №1 и №2. Для полученных бинарных файлов с наилучшими результатами измерьте объем файла, а также время работы программы (количество измерений — 10).

Варианты заданий (действия для матрицы) :

1. Произведение всех элементов.
2. Сумма всех элементов.
3. Поиск минимального элемента в матрице.
4. Поиск элемента с наименьшим остатком от деления на 5.
5. Поиск строки с наименьшим количеством четных чисел.
6. a[i][j] = a[i][j] \* a[i][j]
7. a[i][j] = a[i][j] % 9
8. a[i][j] = a[i][j+1] + a[i][j-1]
9. a[i][j] = a[i][j] \* a[j][i]
10. Транспонирование матрицы.

**Примечание для преподавателей**: данную лабораторную работу можно проводить не только в стандартном режиме, описанном выше, но и на соревновательной основе по аналогии с олимпиадами. Для этого предлагается следующий упрощенный алгоритм:

1. Выбрать один из вариантов заданий, выдать его в качестве общего на группу студентов.
2. Зафиксировать тривиальное решение для выбранного задания (исходный код и команду для сборки), которое будет выступать в качестве базового уровня с точки зрения оптимизируемых характеристик (объем файла и скорость).
3. Зафиксировать параметры среды и способа измерения скорости (версии компиляторов и линковщика, ОС, инструментов измерения, параметров виртуальной машины).
4. В качестве задания указать оптимизацию одного или обоих параметров приложения, не меняя его сути и не нарушая корректность работы основного алгоритма.

### **8.3.1. Описание последовательности выполнения работы**

В качестве первого шага, необходимо реализовать программу согласно выбранному варианту. При реализации программы не забудьте реализовать вывод исходных данных и ответа в консоль. Выполним сборку программы без флагов оптимизации с помощью. компилятора gcc (или путем кросс компиляции, или запуская компилятор непосредственно в рамках RISC-V ОС). Программу необходимо отладить перед оптимизацией — с помощью данных меньшего объема и многократных запусков убедитесь, что она корректно справляется с поставленной задачей и успешно выполняется без сбоев и ошибок.

Далее, измерим исходные значения времени работы программы и объем бинарного файла с помощью утилит time / multitime и size. При измерении времени работы программы, убедитесь, что вы измеряете время работы программы, а не время ее запуска в рамках процесса qemu-riscv64:

$ qemu-riscv64 -L "$RISCV/sysroot" time ./hello # Корректное измерение

$ time qemu-riscv64 -L "$RISCV/sysroot" ./hello # Некорректное измерение

Далее необходимо выполнить оптимизацию с помощью флагов компиляции. Мы рекомендуем увеличивать степень оптимизации постепенно, начиная с флага -O1, контролируя корректность работы самой программы по вводимым значениям и измеряя время работы. Аналогичные действия необходимо предпринять и для оптимизации объема бинарного файла. После окончания оптимизации необходимо повторить измерения и внести полученные данные в отчет.

### **8.3.2. Пример выполнения задания на защиту**

Оптимизируйте время работы бинарного файла для программы суммирования остатков от деления на 3 для целочисленного случайного массива из 10 элементов. Предположим, что исходный код программы уже написан. Выполним сборку и измерим время работы на 10 запусках:

$ gcc main.c -o a.out

$ multitime -n 10 ./a.out

…

===> multitime results

1: ./a.out

Mean Std.Dev. Min Median Max

real 0.016 0.002 0.012 0.016 0.021

user 0.007 0.005 0.000 0.007 0.013

sys 0.008 0.004 0.004 0.007 0.015

Для начала оптимизации по времени рекомендуется выполнить компиляцию c флагом -O1 и оценить достигнутый эффект.

$ gcc -O1 main.c -o a.out

$ multitime -n 10 ./a.out

…

===> multitime results

1: ./a.out

Mean Std.Dev. Min Median Max

real 0.012 0.002 0.010 0.016 0.020

user 0.005 0.005 0.000 0.007 0.013

sys 0.008 0.004 0.004 0.007 0.015

## **8.4. Вопросы для контроля**

1. Для каких задач требуется оптимизация размера бинарного файла?
2. Какие негативные последствия может нести оптимизация времени выполнения программы?
3. Какие флаги отвечают за оптимизацию времени выполнения программы?

# Лабораторная работа №9 МЕТОДЫ ОТЛАДКИ ПРОГРАММ

## **9.1 Цель и задачи**

Целью работы является освоение способов отладки программ на примере RISC-V.

Для достижения поставленной цели требуется решить следующие задачи:

1) Изучить флаги компиляции для отладки программ на С.

2) Ознакомиться с основными командами отладчика gdb.

3) Рассмотреть типовые приемы отладки.

## **9.2. Основные теоретические сведения**

Отладка программ является неотъемлемой частью работы программиста. Существует несколько подходов к отладке [20]:

* инструментирование (отладочный вывод, санитайзеры, assertion),
* использование отладчика (в нативном и удаленном варианте),
* применение JTAG.

В идеальных условиях, все три подхода должны применяться на различных стадиях жизненного цикла ПО, так как каждый из них обладает своими сильными сторонами: инструментирование позволяет в автоматическом режиме на ранних этапах обнаружить большое количество ошибок, отладчики обеспечивают эффективное воспроизведение и отладку ошибок времени выполнения, а JTAG является способом отладить ошибки для уже развернутых встраиваемых систем.

Наиболее важным и востребованным инструментом поиска ошибок в работе программ выступает отладчик. Как правило, графические среды разработки содержат специальные интерфейсы для его запуска, позволяющие визуализировать работу программы. Однако при этом, чаще всего, в основе подобных наглядных инструментов лежит открытый отладчик gdb [21], реализующий типовые операции отладки бинарных файлов программ с помощью интерактивной оболочки командной строки:

* точки останова,
* просмотр и изменение значений переменных,
* просмотр отладочной информации (исходный код программы),
* просмотр ассемблерного кода,
* трассировка выполнения программы (пошаговое выполнение инструкций).

Рассмотрим пример сеанса отладки программы a.out в gdb. Для примера будем считать, что наша программа ожидает ввода целого числа и после ввода выполнит деление константы на данное число, а также что она собрана с флагом -g. Для начала запустим gdb с указанием бинарного файла для отладки:

$ gdb a.out

GNU gdb (Ubuntu 8.2-0ubuntu1~16.04.1) 8.2

Copyright (C) 2018 Free Software Foundation, Inc.

License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

This is free software: you are free to change and redistribute it.

There is NO WARRANTY, to the extent permitted by law.

Type "show copying" and "show warranty" for details.

This GDB was configured as "x86\_64-linux-gnu".

Type "show configuration" for configuration details.

For bug reporting instructions, please see:

<http://www.gnu.org/software/gdb/bugs/>.

Find the GDB manual and other documentation resources online at:

<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".

Type "apropos word" to search for commands related to "word"...

Reading symbols from a.out...(no debugging symbols found)...done.

(gdb)

После этого открылась оболочка (Интерфейс командной строки) отладчика gdb. Строка (gdb) является приглашением ввода команд. Обратите внимание, что сразу после запуска отладчика программа не начинает свою работу — для ее запуска необходимо вызвать команду run.

(gdb) run

Starting program: /home/user/a.out

В данной точке начинается выполнение бинарного файла программы — a.out ожидает пользовательского ввода. Введем число 0, чтобы спровоцировать ошибку:

Starting program: /home/user/ws/a.out

0

Program received signal SIGFPE, Arithmetic exception.

0x0000555555554717 in main () at example.c:9

9 a = a / b;

(gdb)

Мы увидим сообщение отладчика о типе ошибки (Arithmetic exception), указание на стек вызова и проблемную строку из исходного кода. В данный момент исполнение программы (в данном случае — аварийное завершение работы) приостановлено и мы можем изучить обстоятельства, приведшие к ошибке. Для печати более широкого фрагмента исходного кода используем команду l (list):

(gdb) l

4 int a=0;

5 int b;

6

7 scanf("%d", &b);

8

9 a = a / b;

10

11 return 0;

12 }

(gdb)

Для печати текущих (на момент сбоя) значений переменных используем команду p (print):

(gdb) p a

$1 = 0

(gdb) p b

$2 = 0

Для изменения значений переменных можно использовать команду set:

(gdb) set variable a=20

(gdb) p a

$3 = 20

Для продолжения выполнения программы используем c (continue):

(gdb) c

Continuing.

Program terminated with signal SIGFPE, Arithmetic exception.

The program no longer exists.

Мы можем также искусственно остановить работу программы в определенной точке, не дожидаясь сбоя, используя команду break (создание точки останова):

(gdb) break 8

Breakpoint 1 at 0x555555554710: file example.c, line 9.

Запустим программу заново командой run и снова введем 0:

(gdb) run

Starting program: /home/vood/ws/a.out

0

Breakpoint 1, main () at example.c:9

9 a = a / b;

(gdb)

Выполнение программы снова остановлено, однако это произошло ДО ошибки деления на ноль, поэтому в данной точке у нас есть возможность подробнее изучить причины сбоя или вмешаться в выполнение программы для проверки гипотез. После данных манипуляций можно возобновить работу программы командой continue.

## **9.3. Задание к курсовой работе**

Вам дан бинарный исполняемый файл, скомпилированный с отладочной информацией (флаг -g). Данный файл реализует логику программы расчета контрольных сумм для строк по оригинальному (но не практичному) алгоритму. Исходная строка вводится в программу через поток stdin. Программа аварийно завершает работу при вводе строки “sdasdasz”.

Вам необходимо провести отладку программы с помощью gdb и определить:

* номер строки в исходном коде для строки, где происходит ошибка;
* вмешаться в поведение программы и, преодолев участок с ошибкой, определить значение контрольной суммы (для этого необходимо зафиксировать команды GDB, которые нужно для этого выполнить).

Пример программы:

#include <stdio.h>

int main(void) {

char data[5] ;

scanf("%s", data);

int sum = 0;

int i;

int \*pointer = NULL;

for(i = 0; data[i] != '\0' && data[i + 1] != '\0'; i += 2) {

int value = ((unsigned char)data[i] << 8) | (unsigned char)data[i + 1];

if (data[i] == 'z' ) pointer = 4 ;

sum = (sum + value ) & 0xffff;

}

printf("Sum is %04x\n", sum);

return 0;

}

**Примечание для преподавателей:** Выше приведен пример исходного кода программы, расчитаный на начинающих студентов. Мы рекомендуем дополнительно модифицировать этот файл в тайне от студентов так, чтобы логика и требуемые значения были изменены — в противном случае, студенты могут скопировать часть ответа из кода выше. При необходимости, данное задание можно усложнить с помощью нескольких способов:

1. Обфусцировать исходный код программы [30].
2. Пересобрать бинарный файл без флага -g. Тогда студентам потребуется анализировать поведение ПО с опорой только на ассемблерный код.
3. Использовать другие виды ошибок времени выполнения.
4. Изменить требования задания в сторону большего уклона в реверс-инжиниринг (выявление промежуточных значений переменных или изменение работы программы).

### **9.3.1. Описание последовательности выполнения работы**

Первым шагом для выполнения задания необходимо воспроизвести описанную неисправность вне отладчика путем запуска программы в RISC-V ОС и передачи ей строки, вызывающей ошибку. Это необходимо для того, чтобы в дальнейшем вести отладку с учетом максимально достоверной картины — в ряде случаев использование отладчиков может само по себе изменять поведение программы.

Далее, программу необходимо запустить в gdb и зафиксировать момент ее аварийной остановки. После этого необходимо визуализировать исходный код программы через команды gdb и путем изучения значений переменных, определить — какая именно операция привела к ошибке. Затем завершить выполнение программы.

Далее, необходимо установить точку останова до момента сбоя так, чтобы на момент остановки у вас была возможность вмешаться в ее работу путем изменения значений переменных. Далее запустите программу через gdb и введите данные, вызывающие ошибку. После остановки на точке останова, измените значения нужных переменных в программе и продолжите ее выполнение. Убедитесь, что после данного вмешательства не происходит ошибок и программа корректно завершает работу. Зафиксируйте ее вывод в отчете.

### **9.3.2. Пример выполнения задания на защиту**

Для примера рассмотрим измененный пример программы из теоретической части:

#include <stdio.h>

int main(){

int a=0;

int b;

int c= 13;

scanf("%d", &b);

a = a / b;

printf("%d", b + c);

return 0;

}

Мы знаем, что программа должна выводить некоторую контрольную сумму, но при вводе значения 0 происходит ошибка. Воспроизводим ошибку в gdb:

$ gdb a.out

Starting program: /home/user/ws/a.out

0

Program received signal SIGFPE, Arithmetic exception.

0x0000555555554717 in main () at example.c:9

9 a = a / b;

Последнее сообщение содержит номер строки, в которой произошла ошибка — строка №9. Выведем часть исходного кода через отладчик

(gdb) l

4 int a=0;

5 int b;

6 int c= 13;

7

8 scanf("%d", &b);

9 a = a / b;

10 printf("%d", b + c);

11

12 return 0;

13 }

Таким образом мы видим проблему в исходном коде — результат ввода пользователя в переменной b используется для деления. В случае если b=0 происходит ошибка деления на ноль. При этом результат деления никак не используется. Поэтому, при использовании точек останова и редактирования значений переменных мы можем вмешаться в поведение программы и преодолеть сбой:

(gdb) continue

Continuing.

Program terminated with signal SIGFPE, Arithmetic exception.

The program no longer exists.

(gdb) break 9

Breakpoint 1 at 0x555555554757: file example.c, line 9.

(gdb) break 10

Breakpoint 2 at 0x555555554763: file example.c, line 10.

(gdb) run

Starting program: /home/vood/ws/a.out

0

Breakpoint 1, main () at example.c:9

9 a = a / b;

(gdb) set variable b=1

(gdb) continue

Continuing.

Breakpoint 2, main () at example.c:10

10 printf("%d", b + c);

(gdb) set variable b=0

(gdb) continue

Continuing.

13[Inferior 1 (process 26853) exited normally]

## **9.4. Вопросы для контроля**

1. Приведите примеры кода на языке С, который может вызвать ошибку времени выполнения.
2. Что необходимо сделать в процессе сборки программы компилятором gcc, чтобы ее бинарный файл включал отладочную информацию?
3. Объясните, как с точки зрения программиста выглядит работа механизма точек останова.

# Лабораторная работа №10 НАСТРОЙКА И ЗАПУСК ЗАДАЧ FreeRTOS

## 10.1 Цели и задачи

Целью работы является освоение процесса запуска многопоточного резидентного программного обеспечения для микросхем на основе RISC-V под управлением FreeRTOS.

Для достижения поставленной цели требуется решить следующие задачи:

1) Изучить принцип настройки FreeRTOS.

2) Ознакомиться с процессом запуска ядра FreeRTOS.

3) Разработать задачи, выполняемые во FreeRTOS.

## 10.2 Основные теоретические сведения

В настоящее время к оборудованию, для которого разрабатывается встроенное ПО, предъявляются новые требования по надежности, времени реакции на событие и объему выполняемых функций. Это приводит к усложнению логики работы программ. Другими словами, эра тривиальных задач подходит к концу или может быть уже завершена, а для решения нетривиальных задач разработчику необходим своего рода помощник. Операционная система реального времени (ОСРВ) прекрасно выполняет роль помощника разработчика. ОСРВ с легкостью берет на себя часть рутинной работы, а главное — реализует механизм псевдопараллельного исполнения кода задач.

На сегодняшний день на рынке ОСРВ появляется все больше решений с открытым исходным кодом, которые могут составить конкуренцию коммерческим аналогам. Примером является FreeRTOS [9], которая портирована более чем на 20 платформ (микроконтроллеров) и потребует от аппаратного обеспечения от 32 Кбайт флэш-памяти и от 16 Кбайт ОЗУ. FreeRTOS предоставляет­ся с открытым исходным кодом программ и лицензирована в соответствии с GNU General Public License (GPL) [22].

Использование FreeRTOS позволит нам организовать сразу несколько потоков (задач, нитей — threads) и в любой момент уничтожить, приостановить или запустить любой из них.

Любая задача может иметь несколько состояний на конкретный момент (Рис. 10.1):

1) Выполняется (Running);

2) Готова к выполнению (Ready);

3) Блокирована (Blocked);

4) Приостановлена (Suspended).

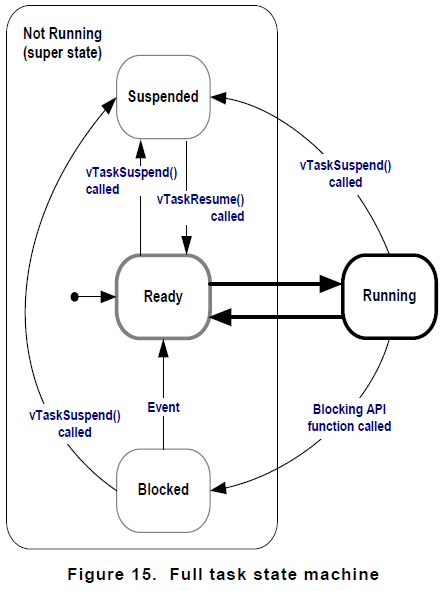


Рис. 10.1 Возможные состояния задач во FreeRTOS.

Отличием FreeRTOS от ОС общего назначения (Windows, Linux, MacOS) заключается в том, что программы под управлением ОСРВ организуют ограниченный набор действий, но эти действие требуют максимально быстрой обработки. ОС общего назначения могут реализовывать больший набор функций, но время отклика для них не является критичным параметром.

Если рассмотреть типичное приложение с использованием FreeRTOS, то можно выделить три слоя поверх аппаратного обеспечения: пользовательский код, платформонезависимый код и платформозависимый код [23]:

* **Платформонезависимый код**
  + **Задачи.** Основное назначение ядра — работа с задачами.
  + **Связь.** Задачам необходимо обмениваться данными, за это отвечает механизм очередей в ядре ОС.
* **Аппаратное сопряжение —** содержит код, абстрагирующий пользовательские программы от особенностей конкретного аппаратного обеспечения.

Для запуска созданных задач требуется запустить планировщик операционной системы. Для этого требуется вызвать функцию vTaskStartScheduler(). Место вызова этой функции в коде не принципиально, однако обычно её используют в функции main.

В общем случае структура программы выглядит следующим образом:

#include “clock\_config.h”

#include “bsp.h”

#include “FreeRTOS.h”

int main()

{

// config CPU/MPU clock

init\_clock();

// init CPU/MPU periphery

init\_bsp();

// create tasks, mutexes, etc

create\_freertos\_elements();

// start OS

vTaskStartScheduler();

// This line will never be reached

while(1){}

}

Точкой входа в программу является функция main, однако задачи запускаются планировщиком ОС. Допускается выносить создание задач и запуск планировщика в отдельную функцию.

Задачи создаются с помощью функции xTaskCreate.

BaseType\_t xTaskCreate( TaskFunction\_t pvTaskCode,

const char \* const pcName,

[configSTACK\_DEPTH\_TYPE](https://www.freertos.org/a00110.html#configSTACK_DEPTH_TYPE) usStackDepth,

void \*pvParameters,

UBaseType\_t uxPriority,

TaskHandle\_t \*pxCreatedTask

);

Функция принимает в качестве параметров указатель на исполняемую функцию, название задачи, размер выделяемой для задачи памяти на стеке, указатель на структуру параметров задачи, приоритет и обработчик (Handler) задачи для доступа из других задач. В качестве параметра для задач может передаваться (void\*)NULL, если параметра нет.

Требуется проверять успешность выполнения запуска задачи, так как из-за нехватки памяти этого может не произойти. В таком случае предлагается уменьшать размер задач и размер стека, выделяемый для задачи.

Удаление задачи выполняется с помощью функции void vTaskDelete( TaskHandle\_t xTask ), которая принимает в качестве параметра обработчик задачи.

Для управления задачами существует набор функций:

* vTaskDelay(const TickType\_t xTicksToDelay)
* vTaskDelayUntil(TickType\_t \*pxPreviousWakeTime, const TickType\_t xTimeIncrement)
* [xTaskDelayUntil(TickType\_t \*pxPreviousWakeTime, const TickType\_t xTimeIncrement)](https://www.freertos.org/xtaskdelayuntiltask-control.html)
* uxTaskPriorityGet(const TaskHandle\_t xTask)
* uxTaskPriorityGetFromISR(const TaskHandle\_t xTask)
* [uxTaskBasePriorityGet(const TaskHandle\_t xTask)](https://www.freertos.org/uxtaskbasepriorityget.html)
* uxTaskBasePriorityGetFromISR(const TaskHandle\_t xTask)
* [vTaskPrioritySet(TaskHandle\_t xTask, UBaseType\_t uxNewPriority)](https://www.freertos.org/a00129.html)
* vTaskSuspend(TaskHandle\_t xTask)
* vTaskResume(TaskHandle\_t xTask)
* xTaskResumeFromISR(TaskHandle\_t xTask)
* xTaskAbortDelay(TaskHandle\_t xTask)

Эти функции осуществляют задержку, смену приоритета, приостановку и возобновление задач. Они вызываются в коде задач. Это требуется для того, чтобы управлять выполнением задач: прервать одну задачу для упрощения выполнения других, повышение приоритета для выполнения более важных функций, итд.

Основной функцией задержки является vTaskDelay( const TickType\_t xTicksToDelay ). Она приостанавливает задачу на указанное число системных тиков. Длительность тика определяется в FreeRTOSConfig.h. Для их использования в этом файле нужно определить макрос INCLUDE\_vTaskDelay, равный 1.

Для получения приоритета задачи используется функция uxTaskPriorityGet(), а для установки — vTaskPrioritySet(). Для использования функции получения приоритета задачи нужно создать макросы INCLUDE\_uxTaskPriorityGet и configUSE\_MUTEXES, равные 1. Для использования функции изменения приоритета нужно создать макрос INCLUDE\_vTaskPrioritySet, равный 1.

Для приостановки задачи используется функция vTaskSuspend(TaskHandle\_t xTaskToSuspend), а для возобновления — vTaskResume(TaskHandle\_t xTaskToResume). Для их использования в этом файле нужно определить макрос INCLUDE\_vTaskDelay, равный 1.

Во многих функциях в названии присутствует FromISR. Это означает, что данные функции предназначены для использования в аппаратных прерываниях.

Пример запуска задачи с параметром:

/\* Task to be created. \*/

void vTaskCode( void \* pvParameters )

{

/\* The parameter value is expected to be 1 as 1 is passed in the

pvParameters value in the call to xTaskCreate() below.

[configASSERT](https://www.freertos.org/a00110.html#configASSERT)( ( ( uint32\_t ) pvParameters ) == 1 );

for( ;; )

{

/\* Task code goes here. \*/

}

}

/\* Function that creates a task. \*/

void vOtherFunction( void )

{

BaseType\_t xReturned;

TaskHandle\_t xHandle = NULL;

/\* Create the task, storing the handle. \*/

xReturned = xTaskCreate(

vTaskCode,/\* Function that implements the task. \*/

"NAME", /\* Text name for the task. \*/

STACK\_SIZE, /\* Stack size in words, not bytes. \*/

( void \* ) 1,/\* Parameter passed into the task. \*/

tskIDLE\_PRIORITY,/\* Priority of created task. \*/

&xHandle ); /\* Pass out the created handle.\*/

if( xReturned == pdPASS )

{

/\* The task was created. \*/

/\* Use the task's handle to delete the task. \*/

[vTaskDelete](https://www.freertos.org/a00126.html)( xHandle );

}

}

При отсутствии параметров для задачи в соответствующем параметре передается указатель на NULL, не используемый в коде. В общем случае разработка кода для одинаковых задач с большим количеством параметров может потребовать больше кода, чем разработка специализированных задач.

Помимо штатных инструментов FreeRTOS одним из важнейших приемов работы для разработки встраиваемого ПО выступит условная компиляция.Для того, чтобы в рамках одного проекта сформировать разный набор задач предлагается использовать условную компиляцию. Для её использования применяются директивы препроцессора для создания макросов. Напомним, как работает данный принцип. Основной директивой является #define, а выбор нужных действий осуществляется с помощью следующих директив:

* #ifdef — выполнить блок команд, если макрос определен,
* #ifndef — выполнить блок команд если макрос не определен,
* #if — выполнить блок команд если макрос соответствует условию,
* #else — завершить блок команд и выполнить другой блок, противоположный по назначению одной из команд выше,
* #endif — завершить блок команд.

Пример использования условной компиляции:

#define TASKS\_SET\_1

//#define TASKS\_SET\_2

void create\_tasks()

{

#ifndef TASKS\_SET\_1

create\_task\_1();

create\_task\_2();

#endif

#ifdef TASKS SET\_2

create\_task\_3();

create\_task 4();

#else

create\_task\_5();

create\_task\_6();

#endif

}

В ходе выполнения этой функции будут созданы задачи 5 и 6.

Использование условной компиляции позволяет достигнуть следующих преимуществ для встраиваемых систем:

* Возможность выбора конфигурации задач на основе созданных наборов.
* Возможность добавления и быстрого отключения отладочных функций.
* Создание числовых или строковых констант без увеличения объема исполняемого файла (использование для этих целей ключевого слова const приведет к размещению созданного значения в постоянной памяти микросхемы).

### 10.2.1 Сборка и запуск программ FreeRTOS

Основным назначением FreeRTOS является разработка многозадачных программ для встраиваемых систем на основе микроконтроллеров и микропроцессоров. Для программирования этих устройств используются отладочные платы. Однако, работа с многозадачностью на реальных аппаратных устройствах чревата дополнительными сложностями, которые выходят за рамки задач данного практикума. Поэтому, для упрощения выполнения лабораторных работ вместо отладочных плат будет использоваться эмулятор на базе QEMU. Так, вместо мигания светодиодом требуется выводить сообщение об изменении его состояния.

Для разработки своих программ на базе FreeRTOS требуется использовать файлы исходного кода ОС. Для этого нужно загрузить архив по ссылке [24]. В архиве есть исходный код и демонстрационные проекты для разных архитектур. Для выполнения лабораторных работ №13-15 предлагается использовать демонстрационный проект /FreeRTOS/Demo/RISC-V-Qemu-virt\_GCC в качестве базового шаблона. Данный проект уже включает в себя необходимые файлы для сборки приложений под архитектуру RISC-V.

Выполним сборку и запуск проекта с помощью инструментов Syntacore Kit. Для этого необходимо открыть в терминале гостевой ОС каталог проекта и выполнить команду

$ make

При ее выполнении возможно возникновение два вида ошибок:

1. Если пример не скомпилируется из-за отсутствующих инструкций csrc и csrv (ошибки вида “**Error: unrecognized opcode …**”), то необходимо в файле Makefile изменить все вхождения -march=rv32imac на -march=rv32imac\_zicsr\_zifencei для поддержки необходимого расширения.
2. В случае невозможности компиляции из-за не определенной константы configCLINT\_BASE\_ADDRESS, необходимо добавить её ручное определение в FreeRTOSConfig.h. В этом файле используется константа CLINT\_ADDR. В таком случае необходимо добавить в начало файла FreeRTOSConfig.h после директив #include строку

#define configCLINT\_BASE\_ADDRESS CLINT\_ADDR

После успешной компиляции, в каталоге проекта появится бинарный файл ./build/RTOSDemo.axf . Запустим его выполнение в QEMU с помощью команды

$ qemu-system-riscv32 -nographic -machine virt -net none \

-chardev stdio,id=con,mux=on -serial chardev:con \

-mon chardev=con,mode=readline -bios none \

-smp 4 -kernel ./build/RTOSDemo.axf

Если всё выполнено корректно, то будет запущен набор задач main\_blinky (Рис. 10.2), который выводит строку приветствия и периодически выводит данные, передаваемые между двумя задачами с помощью очереди.

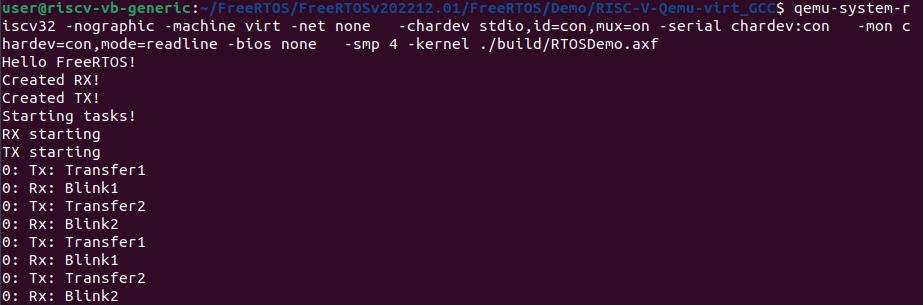


Рис. 10.2 Запущенный проект hello\_blinky

## 10.3 Задание к лабораторной работе

Для выполнения работы требуется:

1. Подготовить проект с настроенной FreeRTOS для QEMU.
2. Реализовать задачи, согласно требованиям.
3. Запустить программу в эмуляторе и убедиться в работоспособности задач.

В лабораторной работе необходимо разработать два набора задач для программы. Каждая задача осуществляет мигание светодиодом. В первом наборе задач светодиод и период мигания задаются в теле задачи. Во втором наборе требуется реализовать одну задачу, которая вызывается необходимое количество раз с передачей параметров. Оба набора задач должны выполнять одинаковый функционал. Для выбора реализуемого набора задач предлагается использовать условную компиляцию. Порядок работы со светодиодами:

* Количество светодиодов n = max(k, 2 + (a % 4));
* Период зажигания первого светодиода T[0] = a \* t;
* Период зажигания последующих светодиодов: T[i+1] = T[i] + b \* t;
* Скважность формируемых сигналов 50%.

где а = номер варианта, b = количество букв в фамилии студента, t — базовый период, принимаемый как 200 тиков процессора, k — количество доступных для работы светодиодов. Для эмуляции в эмуляторе предлагается выводить строку, обозначающую мигание определенного светодиода. Максимальное количество светодиодов k предлагается принять равным 5 при эмуляции. Для остановки выполнения программы используется стандартное в ОС Linux сочетание клавиш для остановки выполнения программы Ctrl+C.

### 10.3.1 Описание последовательности выполнения работы

Для выполнения задания необходимо:

1. Настроить среду для работы согласно инструкции из п. 10.2.1.
2. Вычислить параметры вашего варианта задания.
3. С помощью функций управления задачами FreeRTOS модифицировать поведение светодиодов.
4. Выполнить сборку и запуск модифицированного примера, зафиксировать результат работы в отчете.

### 10.3.2 Пример выполнения задания на защиту

//#define USE\_PARAMETERS\_IN\_TASKS

#ifndef USE\_PARAMETERS\_IN\_TASKS

void parameter\_task\_function\_1(void \*pvParameters)

{

while(1)

{

vSendString(“Toggle 1”);

vTaskDelay(25000);

}

}

void parameter\_task\_function\_2(void \*pvParameters)

{

while(1)

{

vSendString(“Toggle 2”);

vTaskDelay(50000);

}

}

void parameter\_task\_function\_3(void \*pvParameters)

{

while(1)

{

vSendString(“Toggle 3”);

vTaskDelay(100000);

}

}

int main\_tasks(){

xTaskCreate((TaskFunction\_t )parameter\_task\_function\_1,

(const char\* )”LED1\_Task”,

(uint16\_t )512,

(void\* )NULL,

(UBaseType\_t )2,

(TaskHandle\_t\* )NULL);

xTaskCreate((TaskFunction\_t )parameter\_task\_function\_2,

(const char\* )”LED2\_Task”,

(uint16\_t )512,

(void\* )NULL,

(UBaseType\_t )2,

(TaskHandle\_t\* )NULL);

xTaskCreate((TaskFunction\_t )parameter\_task\_function\_3,

(const char\* )”LED3\_Task”,

(uint16\_t )512,

(void\* )NULL,

(UBaseType\_t )2,

(TaskHandle\_t\* )NULL);

vTaskStartScheduler();

return 0;

}

#else

typedef struct {

int num;

char\* text;

} lab\_task\_data;

char\* names[] = {"Toggle 1", "Toggle 2", "Toggle 3"};

lab\_task\_data data[3];

void parameter\_task\_function(void \*pvParameters)

{

lab\_task\_data\* config = (lab\_task\_data\*)pvParameters;

while(1)

{

vSendString(config->text);

vTaskDelay(config->num);

}

}

int main\_tasks(){

for(int i=0;i<3;i++){

data[i].num = 100\*(250<<i);

data[i].text = names[i];

xTaskCreate((TaskFunction\_t )parameter\_task\_function,

(const char\* )names[i],

(uint16\_t )512,

(void\* )&data[i],

(UBaseType\_t )2,

(TaskHandle\_t\* )NULL);

}

vTaskStartScheduler();

return 0;

}

#endif

int main(){

main\_tasks();

while(1);

}

Каждая из этих программ выполняет мигание светодиодами. Каждый светодиод мигает с определенной частотой, заданной параметром задержки.

## 10.4 Вопросы для контроля

1. В чем отличие ОС общего назначения от ОСРВ?
2. Что необходимо определить для создания задачи во FreeRTOS?
3. Какие состояния задач есть во FreeRTOS?

# Лабораторная работа №11 МЬЮТЕКСЫ FreeRTOS

## 11.1 Цели и задачи

Целью работы является освоение механизма блокировок во FreeRTOS на основе мьютексов и семафоров.

Для достижения поставленной цели требуется решить следующие задачи:

1) Изучить назначение мьютексов и семафоров.

2) Ознакомиться с процессом использования мьютексов во FreeRTOS.

3) Разработать задачи с блокировкой, выполняемые во FreeRTOS.

## 11.2 Основные теоретические сведения

Мьютекс (Mutex: MUTual EXception) является разновидностью семафора. Семафор — примитив синхронизации работы процессов и потоков, в основе которого лежит счётчик, над которым можно производить две операции: увеличение и уменьшение значения на единицу, при этом операция уменьшения для нулевого значения счетчика является блокирующейся. Служит для построения более сложных механизмов синхронизации и используется для синхронизации параллельно работающих задач, для защиты передачи данных через разделяемую память, для защиты критических секций, а также для управления доступом к аппаратному обеспечению. Параметром семафора является максимальное значение его счетчика. Мьютекс — семафор, максимальное значение которого равно единице [25].

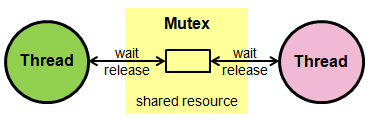


Рис. 11.1 Назначение мьютекса

FreeRTOS разделяет понятия мьютекса и бинарного семафора. Отличие заключается в том, что у бинарных семафорах отсутствует механизм модификации приоритетов, который есть в мьютексах. Это проявляется тем, что у задачи, повышается приоритет на время владения мьютексом. Таким образом двоичные семафоры лучше подходят для синхронизации задач, а мьютексы — для блокировки ресурсов. В данной работе рассматриваются только двоичные семафоры и мьютексы. Существуют счетные семафоры и рекурсивные мьютексы, у которых максимальное значение счетчика больше единицы.

Для использования мьютексов и семафоров нужно создать макрос configSUPPORT\_DYNAMIC\_ALLOCATION, равный 1. Для использования мьютексов нужно также создать макрос configUSE\_MUTEXES, равный 1. Мьютексы создаются функцией SemaphoreHandle\_t xSemaphoreCreateMutex( void ), а семафоры — SemaphoreHandle\_t xSemaphoreCreateBinary( void ).

Взятие семафора выполняется функцией xSemaphoreTake( SemaphoreHandle\_t xSemaphore, TickType\_t xTicksToWait ), а освобождение — xSemaphoreGive( SemaphoreHandle\_t xSemaphore ). В функции взятия используется параметр времени ожидания xTicksToWait. Он определяет количество системных тиков, в течение которых задача ждет получение семафора.

В более старых версиях FreeRTOS для определения мьютекса используется объект xSemaphoreHandle. Функционально он ничем не отличается от SemaphoreHandle\_t.

Пример кода работы с мьютексами:

SemaphoreHandle\_t xSemaphore = NULL;

void vATask( void \* pvParameters )

{

xSemaphore = xSemaphoreCreateMutex();

if( xSemaphore != NULL )

{

if( xSemaphoreGive( xSemaphore ) != pdTRUE ) // will fail!

{

…

}

if( xSemaphoreTake( xSemaphore, ( TickType\_t ) 0 ) )

{

…

if( xSemaphoreGive( xSemaphore ) != pdTRUE )

{

…

}

}

}

}

## 11.3 Задание к лабораторной работе

Для выполнения работы требуется:

1. Подготовить проект с настроенной FreeRTOS для QEMU.
2. Реализовать задачи с блокировкой, согласно требованиям.
3. Запустить программу в эмуляторе и убедиться в работоспособности задач.

В лабораторной работе требуется разработать набор задач для доступа к общему ресурсу. В данной работе общим ресурсом является группа светодиодов на плате. В ходе работы нужно ограничить максимальное количество включенных светодиодов. Для этой цели предлагается использовать мьютексы в качестве механизма блокировки задач.

* Количество светодиодов n = max(k, 2 + ((a+1) % 4));
* Период зажигания светодиода T[i] = (a \* t + b \* t \* (i));
* Скважность 50%.
* Количество одновременно горящих светодиодов: 1 + ((b)%(n/2))

где а = номер варианта, b = количество букв в фамилии, t — базовый период, принимаемый как 200 тиков процессора, k — количество доступных для работы светодиодов.

Если светодиод не зажигается из-за указанных ограничений, то программа пропускает период горения и ждет полупериод для повторной попытки.

### 11.3.1 Описание последовательности выполнения работы

Для выполнения задания необходимо:

1. Настроить среду для работы согласно инструкции из п. 11.2.1.
2. Вычислить параметры вашего варианта задания.
3. С помощью функций управления задачами FreeRTOS модифицировать поведение светодиодов.
4. Реализовать блокировки доступа к общему ресурсу с помощью мьютекса
5. Выполнить сборку и запуск модифицированного примера, зафиксировать результат работы в отчете.

### 11.3.2 Пример выполнения работы

typedef struct {

int num;

char\* text;

char\* text\_fail

} lab\_task\_data\_w\_fail;

char\* names[] = {"Toggle 1", "Toggle 2", "Toggle 3"};

char\* names\_fail[] = {"Toggle 1 failed", "Toggle 2 failed", "Toggle 3 failed"};

lab\_task\_data\_w\_fail data\_fail[3];

void mutex\_task\_function(void \*pvParameters)

{

lab\_task\_data\_w\_fail\* config = (lab\_task\_data\_w\_fail\*)pvParameters;

while(1)

{

if(xSemaphoreTake(mutex,0)){

vSendString(config->text);

vTaskDelay(100\*(250<<config->num));

vSendString(config->text);

xSemaphoreGive(mutex);

} else {

vSendString(config->text\_fail);

}

vTaskDelay(100\*(250<<config->num));

}

}

int main\_mutex(){

mutex = xSemaphoreCreateMutex();

for(int i=0;i<3;i++){

data\_fail[i].num = i;

data\_fail[i].text = names[i];

data\_fail[i].text\_fail = names\_fail[i];

xTaskCreate((TaskFunction\_t )mutex\_task\_function,

(const char\* )names[i],

(uint16\_t )512,

(void\* )&data\_fail[i],

(UBaseType\_t )2,

(TaskHandle\_t\* )NULL);

}

vTaskStartScheduler();

return 0;

}

int main(){

main\_mutex();

while(1);

}

В ходе работы программы максимальное количество включенных светодиодов равно 1, так как созданный мьютекс блокирует включение других светодиодов. По этой причине при зажигании светодиода с самым большим периодом, остальные не могли зажечься.

## 11.5 Вопросы для контроля:

1. Какое основное назначение мьютексов и семафоров?
2. В чем отличие семафоров и мьютексов во FreeRTOS?
3. Какие существуют основные параметры семафоров и мьютексов?
4. Что означает параметр времени ожидания при взятии мьютекса?

# Лабораторная работа №12 ОЧЕРЕДИ FreeRTOS

## 12.1 Цели и задачи

Целью работы является освоение механизма передачи данных во FreeRTOS на основе очередей.

Для достижения поставленной цели требуется решить следующие задачи:

1) Изучить назначение очередей и вариантов их применения.

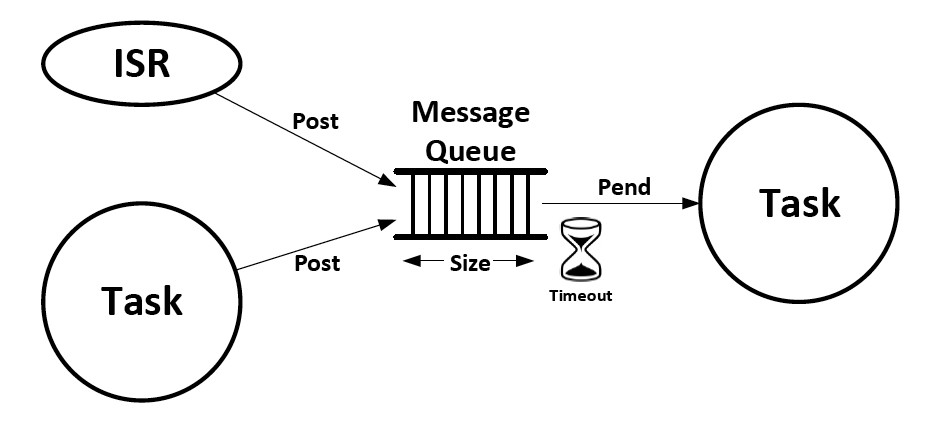
2) Ознакомиться с процессом использования очередей во FreeRTOS.

3) Разработать задачи в FreeRTOS с передачей данных между ними.

## 12.2 Основные теоретические сведения

При разработке ПО возникают ситуации, в которых необходимо передавать данные между задачами. Использование глобальных переменных для этих целей не рекомендуется, так как при переключении между задачами есть риск испортить данные. По этой причине предлагается использовать механизм очередей.

Очередь (Рис. 12.1) создается с помощью функции QueueHandle\_t xQueueCreate( UBaseType\_t uxQueueLength, UBaseType\_t uxItemSize ). Она использует 2 параметра: количество элементов в очереди и их размер.

  
Рис 12.1 Применение очереди

Для отправки данных в очередь есть группа методов: xQueueSendToBack( QueueHandle\_t xQueue, const void \* pvItemToQueue, TickType\_t xTicksToWait ) и xQueueSendToFront( QueueHandle\_t xQueue, const void \* pvItemToQueue, TickType\_t xTicksToWait ). Они отправляют новый элемент в конец или в начало очереди. Параметрами функций являются указатель на используемую очередь, указатель на копируемый параметр и время ожидание на помещение элемента в очередь, если она полностью заполнена.

Следует отметить, что существует функция xQueueSend( QueueHandle\_t xQueue, const void \* pvItemToQueue, TickType\_t xTicksToWait ). Эта функция аналогична xQueueSendToBack и используется для совместимости со старыми версиями FreeRTOS.

Для получения элемента из очереди используется функция BaseType\_t xQueueReceive( QueueHandle\_t xQueue, void \*pvBuffer, TickType\_t xTicksToWait ). Она извлекает элемент из очереди и помещает его в буфер, из которого с ним будет вестись работа. В числе параметров функции есть время ожидания, которая указывает сколько времени ждать элемент, если очередь пуста. Существует аналогичная функция xQueuePeek( QueueHandle\_t xQueue, void \*pvBuffer, TickType\_t xTicksToWait ), которая копирует элемент из очереди без его извлечения. То есть эта функция используется, когда первый в очереди элемент не должен извлекаться.

Функция извлечения получает и удаляет элемент в начале очереди. Таким образом с помощью функций в конец и в начало очереди реализуются примитивы очереди и стека соответственно.

Пример кода работы с очередями:

struct AMessage

{

char ucMessageID;

char ucData[ 20 ];

} xMessage;

QueueHandle\_t xStructQueue = NULL;

QueueHandle\_t xPointerQueue = NULL;

void vCreateQueues( void )

{

xMessage.ucMessageID = 0xab;

memset( &( xMessage.ucData ), 0x12, 20 );

xStructQueue = [xQueueCreate](https://www.freertos.org/a00116.html)(

/\* The number of items the queue can hold. \*/

10,

/\* Size of each item is big enough to hold the

whole structure. \*/

sizeof( xMessage ) );

/\* Create the queue used to send pointers to struct AMessage structures. \*/

xPointerQueue = xQueueCreate(

/\* The number of items the queue can hold. \*/

10,

sizeof( &xMessage ) );

if( ( xStructQueue == NULL ) || ( xPointerQueue == NULL ) )

{

}

}

/\* Task that writes to the queues. \*/

void vATask( void \*pvParameters )

{

struct AMessage \*pxPointerToxMessage;

[xQueueSend](https://www.freertos.org/a00117.html)( xStructQueue,

( void \* ) &xMessage,

( TickType\_t ) 0 );

pxPointerToxMessage = &xMessage;

xQueueSend( xPointerQueue,

( void \* ) &pxPointerToxMessage,

( TickType\_t ) 0 );

/\* ... Rest of task code goes here. \*/

}

/\* Task that reads from the queues. \*/

void vADifferentTask( void \*pvParameters )

{

struct AMessage xRxedStructure, \*pxRxedPointer;

if( xStructQueue != NULL )

{

if( xQueueReceive( xStructQueue,

&( xRxedStructure ),

( TickType\_t ) 10 ) == pdPASS )

{

/\* xRxedStructure now contains a copy of xMessage. \*/

}

}

if( xPointerQueue != NULL )

{

if( xQueueReceive( xPointerQueue,

&( pxRxedPointer ),

( TickType\_t ) 10 ) == pdPASS )

{

/\* \*pxRxedPointer now points to xMessage. \*/

}

}

/\* ... Rest of task code goes here. \*/

}

## 12.3 Задание к лабораторной работе

1. Подготовить проект с настроенной FreeRTOS для QEMU.
2. Реализовать задачи, согласно требованиям.
3. Запустить программу в эмуляторе и убедиться в работоспособности задач.

В лабораторной работе требуется разработать набор задач, которые взаимодействуют между собой с помощью очередей. В данной работе требуется создать числовую последовательность и вычислить итоговое значение на её основе.

Требуется создать набор модифицированных последовательностей Фибоначчи со следующими характеристиками:

Максимальная длина последовательности l = 5 + (a % 6);

f[0] = a;

f[1] = a+b+s;

f[i] = (f[i-1] + f[i-2]) & 0xFF (младший байт)

где где а = номер варианта, b = количество букв в фамилии, s = номер последовательности от 0 до 10.

Требуется реализовать следующий набор задач:

* Задача формирования начальных значений последовательности
* Задачи для вычисления последующих значений.
* Вычисление контрольной суммы КС8 сформированной последовательности (младший байт суммы всех значений в последовательности XOR 0xFF).

Очереди должны обеспечивать передачу всех элементов последовательности в следующую задачу. Использование массивов при формировании последовательности не допускается. Требуется выводить в консоль исходные и генерируемые значения последовательности, а также результат вычислений.

### 12.3.1 Описание последовательности выполнения работы

Для выполнения задания необходимо:

1. Настроить среду для работы согласно инструкции из п. 10.2.1.
2. Вычислить параметры вашего варианта задания.
3. С помощью функций управления задачами FreeRTOS модифицировать поведение светодиодов.
4. Реализовать взаимодействие задач через очереди.
5. Выполнить сборку и запуск модифицированного примера, зафиксировать результат работы в отчете.

### 12.3.2 Пример выполнения

#define SEQUENCE\_LEN 5

QueueHandle\_t queues[SEQUENCE\_LEN-1];

xSemaphoreHandle mutex\_print;

char str[40];

void queue\_init\_task\_function(void \*pvParameters){

int a=1, b=1;

while(1){

for(int a = 1; a < 5; a++){

for (int b = 1; b < 5; b++) {

xQueueSend( queues[0], &a, ( TickType\_t ) 0 );

xQueueSend( queues[0], &b, ( TickType\_t ) 0 );

if(xSemaphoreTake(mutex\_print,2000)){

sprintf( str, "Init: f[0]=%d; f[1]=%d;", a, b);

vSendString(str);

xSemaphoreGive(mutex\_print);

}

vTaskDelay(25000);

}

}

}

}

void queue\_forward\_function(void \*pvParameters){

int cnt = 0, val=0;

int i = (int)pvParameters;

int nb = 0;

while(1){

if(xQueueReceive( queues[i],&val,( TickType\_t ) 10 ) == pdPASS){

xQueueSend( queues[i+1], &val, ( TickType\_t ) 0 );

cnt++;

if(cnt > i){

nb += val;

}

if(cnt == (i + 2)){

if(xSemaphoreTake(mutex\_print,2000)){

sprintf( str, "New: f[%d]=%d;",i+2, nb);

vSendString(str);

xSemaphoreGive(mutex\_print);

}

xQueueSend( queues[i+1], &nb, ( TickType\_t ) 0 );

nb = 0;

cnt = 0;

}

}

}

}

void queue\_process\_function(void \*pvParameters){

int cnt = 0;

int val = 0;

int ks8 = 0;

while(1){

if(xQueueReceive( queues[SEQUENCE\_LEN-2],&val,( TickType\_t ) 10 ) == pdPASS){

ks8+=val;

cnt++;

if(cnt==(SEQUENCE\_LEN)){

cnt = 0;

if(xSemaphoreTake(mutex\_print,2000)){

sprintf( str, "Result: KS8=%d;", ((unsigned char)(~ks8)));

vSendString(str);

xSemaphoreGive(mutex\_print);

}

ks8=0;

}

}

}

}

int main\_queue(){

mutex\_print = xSemaphoreCreateMutex();

for(int i=0; i<SEQUENCE\_LEN-1;i++){

queues[i] = xQueueCreate( 10, sizeof( int ) );

}

xTaskCreate((TaskFunction\_t )queue\_init\_task\_function,

(const char\* )"Init\_Task",

(uint16\_t )512,

(void\* )NULL,

(UBaseType\_t )2,

(TaskHandle\_t\* )NULL);

for(int i=0; i<SEQUENCE\_LEN-2;i++){

xTaskCreate((TaskFunction\_t )queue\_forward\_function,

(const char\* )"Forward\_Task",

(uint16\_t )512,

(void\* )i,

(UBaseType\_t )2,

(TaskHandle\_t\* )NULL);

}

xTaskCreate((TaskFunction\_t )queue\_process\_function,

(const char\* )"Calc\_Task",

(uint16\_t )512,

(void\* )NULL,

(UBaseType\_t )2,

(TaskHandle\_t\* )NULL);

vTaskStartScheduler();

return 0;

}

int main(){

main\_queue();

while(1);

}

Представлен код основных задач: определения начальных элементов, формирования новых байтов, формирования результата. Продемонстрированы основные результаты их работы: формирование начальных данных, генерация новых элементов, вычисление контрольной суммы. Длину последовательности можно регулировать изменением SEQUENCE\_LEN.

## 12.4 Вопросы для контроля

1. Какие основные параметры нужно определить при создании очереди?
2. Можно ли использовать очередь во FreeRTOS для реализации стека?
3. Что происходит при попытке записи элемента в заполненную очередь?

# Лабораторная работа № 13 **Знакомство со средой проектирования. Управление контактами GPIO**

## 13.1 Цели и задачи

Цель работы — изучить порядок работы со средой создания программного обеспечения, освоить правила настройки портов ввода-вывода микроконтроллера ESP32-C3. Задачи:

1. Изучить архитектуру микроконтроллера ESP32-C3.
2. Освоить работу с среде VSCode для разработки микроконтроллерных приложений.
3. Разработать программу для управления контактами GPIO.

## 13.2 Основные теоретические сведения о работе с микроконтроллерами для RISC-V

В настоящее время на рынке представлены микросхемы микроконтроллеров с архитектурой RISC-V различных производителей, например, 32 разрядные контроллеры SweRV Core (Western Digital); FE310 (SiFive); RV32M1 (NXP); MHS001 Huangshan (HUAMI); GD32VF103 (GigaDevice); CH32V103 (WCH); К1986ВК025 (Миландр); ESP32-C3 (Espressif); MIK32 (Микрон).

Перечисленные ядра, исполняющее инструкции RISC-V, разных производителей отличаются: глубиной конвейера, размерами и организацией кэша данных и команд, реализацией схем предсказаний переходов. При этом все ядра используют общую ISA (Instruction Set Architecture).

**Описание использованного ядра RISC-V**

Предлагаемый практикум ориентирован на работу с микроконтроллером ESP-RISC-V (ESP32-C3) компании Espressif Systems. Архитектура микроконтроллера показана на Рис. 13.1.

CPU and Memory. Процессорное ядро ESP-RISC-V представляет собой 32-разрядное ядро, основанное на RISC-V ISA, называемой «I» с префиксом RV32 или RV64 в зависимости от ширины целочисленного регистра. Эта базовая ISA реализует целочисленные вычислительные инструкции, целочисленные инструкции загрузки, сохранения и управления и является обязательной для всех реализаций RISC-V. Кроме того в ядре реализованы стандартные расширения: инструкций целочисленного умножения и деления («M») и расширение набора сжатых инструкций стандарта RISC-V («C»). Для реализации операций умножения и деления предусмотрены соответствующие 32-битные модули. Ядро имеет 4-х ступенчатый скалярный конвейер, оптимизированный по площади, мощности и производительности и поддерживающий тактовую частоту до 160 МГц. В состав процессорного ядра входят контроллер прерываний (INTC), обеспечивающий обработку 32 векторизованных прерываний с семью уровнями приоритета; модуль отладки (JTAG); и системная шина (SYS BUS) для доступа к памяти и периферийным устройствам.

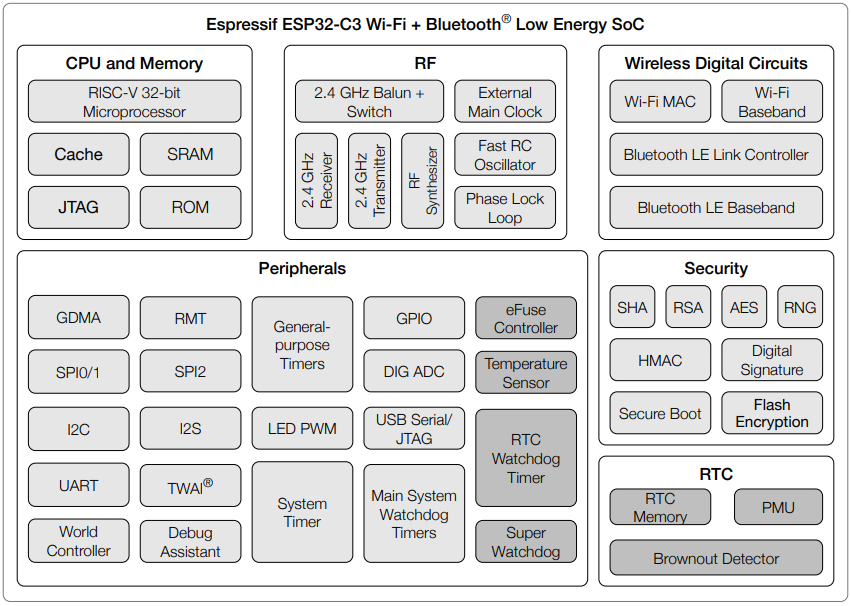


Рис. 13.1. Архитектура микроконтроллера ESP-RISC-V (ESP32-C3)

Внутренняя память ESP32-C3 разделена на три сегмента: внутреннее ПЗУ (384 КБ), внутренняя SRAM (400 КБ) и Real Time Clock (RTC) FAST Memory (8 КБ).

– Внутреннее ПЗУ (384 КБ). ПЗУ содержит программные инструкции и некоторые данные низкоуровневых системных программ.

– Внутренняя память SRAM (400 КБ). Внутреннее статическое ОЗУ– это энергозависимая память, к которой процессор может быстро получить доступ в течение одного такта работы. Часть SRAM может быть настроена на работу в качестве кэша для доступа к внешней памяти. Доступ к некоторым частям SRAM возможен только через шину инструкций процессор, а к некоторым можно обращаться как через шину команд процессора, так и через шину данных процессора.

– Память RTC (Real Time Clock). RTC реализована в виде статического ОЗУ (SRAM), поэтому является волатильной, однако устойчивой в глубоком сне. RTC FAST Memory (8 КБ): к памяти RTC FAST может обращаться только центральный процессор, и она обычно используется для хранения инструкций и данных, которые должны сохраняться во время глубокого сна.

Внутри каждого сегмента могут существовать различные ограничения доступа к шине (например, некоторые части сегмента могут быть доступны только по шине данных процессора), поэтому каждый сегмент также дополнительно делится на части. Части внутренней памяти и диапазоны их адресов на шине данных и/или шине команд приведены в Таблице 13.1.

| Тип шины | Границы адресов | | | Размер | | Применение |
| --- | --- | --- | --- | --- | --- | --- |
| Младший адрес | Старший адрес |  |
| Data bus | 0x3FF0\_0000 | 0x3FF1\_FFFF | 128 | | Internal ROM 1 | |
| 0x3FC8\_0000 | 0x3FCD\_FFFF | 384 | | Internal SRAM 1 | |
| Instruction bus | 0x4000\_0000 | 0x4003\_FFFF | 256 | | Internal ROM 0 | |
| 0x4004\_0000 | 0x4005\_FFFF | 128 | | Internal ROM 1 | |
| 0x4037\_C000 | 0x4037\_FFFF | 16 | | Internal SRAM 0 | |
| 0x4038\_0000 | 0x403D\_FFFF | 384 | | Internal SRAM 1 | |
| Data/Instruction bus | 0x5000\_0000 | 0x5000\_1FFF | 8 | | RTC FAST Memory | |

Таблица 13.1 Части внутренней памяти и диапазоны их адресов на шине данных и/или шине команд

Общая структура и разделение адресов памяти показаны на Рис. 13.2.

Кэш-память (Cache), доступная только для чтения, представляет собой восьмистороннюю ассоциативную память, ее размер составляет 16 КБ, размер блока – 32 байта. Когда кэш активен, он занимает часть пространства внутренней памяти. Кэш единый для инструкций и данных, но одновременно может реализовывать кеширование либо инструкций, либо данных. При возникновении промаха кэш-памяти контроллер кэша инициирует запрос к внешней памяти по отношению к ядру памяти.

Обращение процессора к внешней памяти через кэш показано на Рис. 13.2. В соответствии с настройками MMU (блок управления памятью) кэш сопоставляет адрес процессора с физическим адресом внешней памяти. Благодаря такому сопоставлению адресов ESP32-C3 может адресовать до 16 МБ внешней флэш-памяти.

Блок Peripherals содержит следующие модули:

* GDMA (General Direct Memory Access) – общий контроллер прямого доступа к памяти, он обеспечивает передачу данных от периферийных устройств в память без участия центрального процессора, что позволяет снизить нагрузку на ЦП.
* ESP32-C3 имеет 22 вывода GPIO, которым можно назначить различные функции путем конфигурирования соответствующих регистров. Некоторые GPIO могут использоваться для аналоговых функций, например, для подключения входного сигнала АЦП. Все GPIO имеют возможность выбора внутреннего подтягивания или отвода, или могут быть установлены в высокоимпедансное состояние. Когда GPIO сконфигурированы как входные, входное значение может быть считано программно через регистр. Входные GPIO также могут быть настроены для генерации прерываний процессора по фронту или по уровню. Все цифровые выводы ввода-вывода являются двунаправленными, неинвертирующими и трехпозиционными, включая входные и выходные буферы с трехпозиционным управлением. Эти выводы могут быть мультиплексированы с другими функциями, такими как UART, SPI и т.д. Для работы с низким энергопотреблением GPIO могут быть переведены в состояние удержания. Для маршрутизации сигналов от периферийных устройств к выводам GPIO используются IO MUX и матрица GPIO, обеспечивая высококонфигурируемый ввод/вывод.

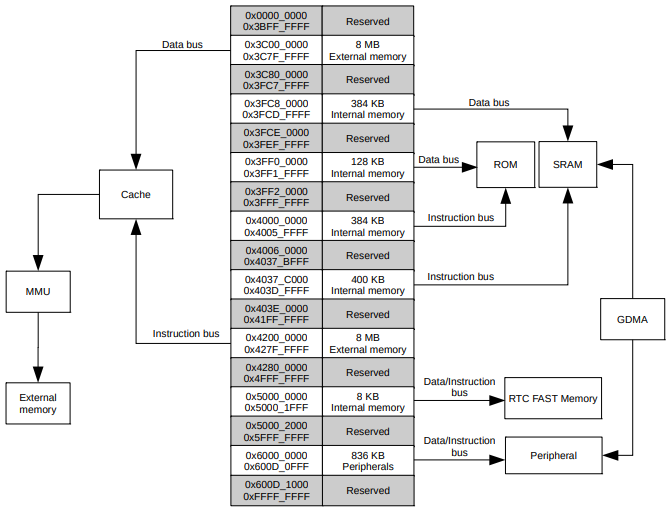
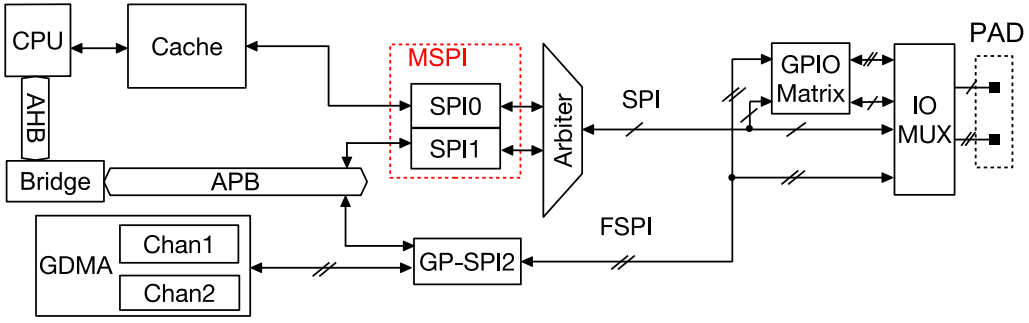


Рис. 13.2. Распределение адресного пространства ESP32-C3

В кристалле реализованы контроллеры последовательных интерфейсов.

SPI (Serial Peripheral Interface) – синхронный последовательный интерфейс, используемый для связи с внешними периферийными устройствами. В ESP32-C3 интегрированы три контроллера SPI: SPI0, SPI1, предназначенные в основном для внутреннего использования и SPI2 (GP-SPI2 или General Purpose SPI2), обладающий широкими возможностями коммуникации. На рис. 13.3 показаны варианты организации обмена данными с устройствами SPI.

  
Рис. 13.3. Подключение SPI контроллеров ESP32-C3

ESP32-C3 имеет два контроллера универсального асинхронного приемника/передатчика UART: UART0 и UART1, контроллер интерфейса I2S и шины I2C (Inter-Integrated Circuit). Для автомобильных и промышленных приложений в кристалл ESP32-C3 встроен двухпроводной автомобильный контроллер TWAI®. Для программирования флэш-памяти системы, для чтения выходных данных программы, а также для подключения отладчика в кристалле ESP32-C3 встроен контроллер USB Serial/JTAG, объединяющий функциональность USB-последовательного преобразователя и адаптера USB-to-JTAG, что исключает необходимость внешних адаптеров. В кристалле имеется устройство дистанционного управления RMT (Remote Control Peripheral), поддерживающее два канала инфракрасной дистанционной передачи и два канала инфракрасного дистанционного приема.

Для обработки аналоговых сигналов в микроконтроллере предусмотрены два 12-битных аналого-цифровых преобразователя ADC (Analog-to-Digital Converter) последовательного приближения (SAR ADC), которые могут обрабатывать аналоговые сигналы с шести выводов.

ШИМ-контроллер LED LED PWM может генерировать независимые цифровые сигналы по шести каналам. ШИМ контроллер используется для управления свечением светодиодов, при котором при котором длительность единичного широтно-импульсно-модулированного сигнала определяет яркость свечения за счет мерцания. Контроллер может генерировать цифровые сигналы с настраиваемыми периодами и рабочим циклом с точностью до 18 бит, поддерживая плавное увеличение или уменьшение рабочего цикла. При этом можно использовать несколько источников синхронизации, включая внешний кристалл. Контроллер может работать даже тогда, когда процессор находится в спящем режиме.

Температурный датчик (Temperature Sensor) предназначен для определения температуры внутри микросхемы. Температура определяется такими факторами, как тактовая частота микроконтроллера или нагрузка на входы/выходы. Выходной сигнал датчика представляет собой напряжение, которое преобразуется АЦП в цифровое значение.

Для реализации различных вариантов контроля времени в микросхеме предусмотрен широкий набор таймеров.

Блок Security содержит модули, реализующие защиту: контроль разрешений на доступ к внутренней и внешней памяти; модули шифрования и дешифрования памяти; средства криптографического аппаратного ускорения, реализованные на алгоритме симметричного шифрования AES (Advanced Encryption Standard), алгоритме безопасного хеширования SHA (Secure Hash Algorithm ) и криптографическом алгоритме с открытым ключом RSA (Rivest, Shamir и Adleman), генератор случайных чисел RNG (Random Number Generator) и средства проверки подлинности сообщений, использующий односторонние хеш-функции HMAC (Hash-based Message Authentication Code).

Блок Wireless digital circuit содержит следующие модули, обеспечивающие передачу данных по радиоканалам. Приемник 2,4 ГГц, демодулирующий радиочастотный сигнал 2,4 ГГц в квадратурные сигналы базовой полосы и преобразующий их в цифровую форму с помощью двух высокоскоростных АЦП высокого разрешения. В приемник интегрированы ВЧ-фильтры, средства автоматической регулировки усиления сигнала, схемы подавления смещения по постоянному току и фильтры базовой полосы. Передатчик 2,4 ГГц модулирующий квадратурные сигналы базовой полосы в радиочастотный сигнал 2,4 ГГц и управляющий антенной с помощью мощных фильтров. Встроенные средства цифровой калибровки дополнительно улучшают линейность усилителя мощности. Тактовый генератор, вырабатывающий квадратурные тактовые сигналы 2,4 ГГц как для приемника, так и для передатчика.

Модуль Wi-Fi MAC реализует полный MAC-протокол Wi-Fi 802.11 b/g/n, поддерживая базовый набор услуг (BSS) STA и SoftAP под управлением функции распределенного управления.

Модуль Bluetooth поддерживает оба существующих протокола: классический с базовой скоростью и повышенной скоростью передачи данных Bluetooth 4.2 BR/EDR (Basic Rate / Enhanced Data Rate) и Bluetooth LE (Low Energy) с низким энергопотреблением. Для подключения и настройки радиомодулей используются функции API Wi-Fi, предоставляемые производителем микроконтроллера.

### 13.2.1 Описание платы прототипирования CORE ESP32

Для выполнения макетных экспериментов используется плата CORE ESP32, разработанная на основе Espressif Systems ESP32-C3. Основная плата поддерживает UART, GPIO, SPI, I2C, ADC, PWM и другие интерфейсы, которые можно выбрать и настраивать в соответствии с решаемой задачей. Описание внешних контактов платы приведено на рис. 13.4.

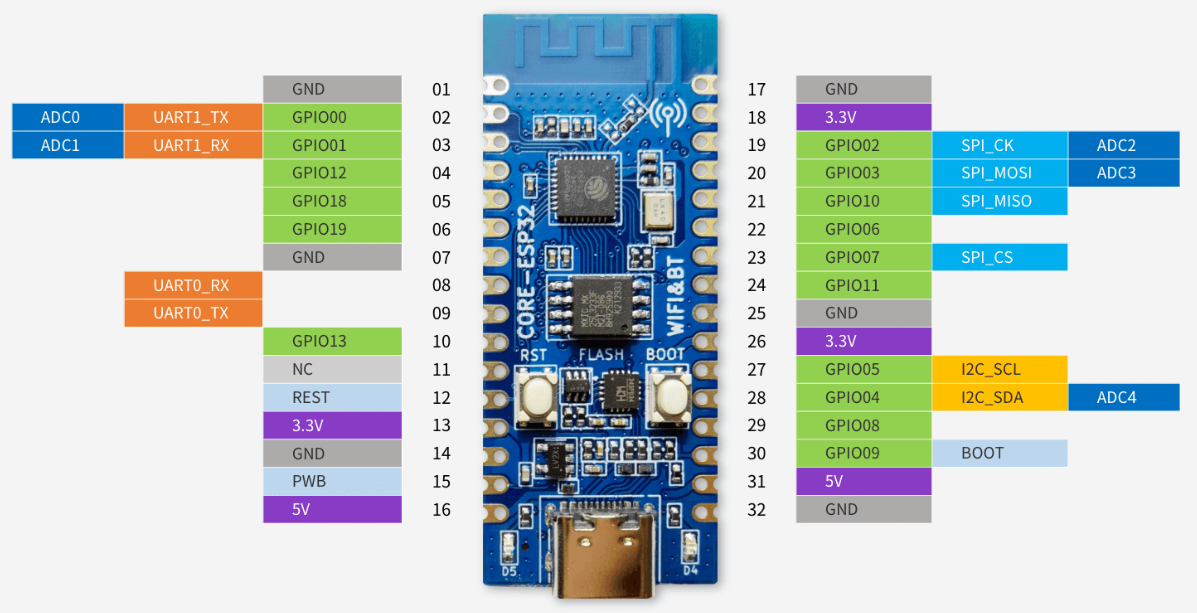


Рис. 13.4. Отладочная плата Core-ESP32

Питание платы CORE-ESP32-C3 можно обеспечить следующими способами:

– источник питания интерфейса USB Type-C, этот способ используется по умолчанию и предпочтителен во время отладки;

– блок питания с контактами 5 В и GND;

– блок питания с контактами 3V3 и GND.

На плате CORE ESP32 установлены:

– 2 светодиода, контакты для подключения к ним показаны в таблице 13.2.

– две кнопки: кнопка BOOT может реализовать функцию загрузки BOOT, кнопка RST может реализовать функцию сброса, контакты для подключения к кнопкам показаны в таблице 13.3.

| **Cветодиод** | **GPIO** | **Описание** |
| --- | --- | --- |
| D4 | IO12 | Активный высокий уровень |
| D5 | IO13 | Активный высокий уровень |

Таблица 13.2 Индикация платы

| **Кнопка** | **Функция контакта** | **Описание** |
| --- | --- | --- |
| BOOT /GPIO9 | При нажатии кнопки микроконтроллер переходит в режим загрузки | Активный низкий уровень |
| RST | При нажатии кнопки происходит сброс микроконтроллера | Активный низкий уровень |

Таблица 13.3 Кнопки платы.

При работе с платой необходимо учитывать следующие особенности:

1. BOOT (IO09) Нельзя допустить подачу на контакт IO09 нуля до включения питания, микроконтроллер перейдет в режим загрузки.
2. Для проектирования используйте вывод IO08, а прямое внешнее подключение не рекомендуется, поскольку при загрузке и записи на выводе IO08 низкий уровень, а последовательный порт нельзя использовать для загрузки.
3. При написании собственного программного обеспечения следует обратить внимание на режим флэш-памяти. Существуют следующие режимы:
   1. DIO – хост SPI использует технологию «быстрого чтения двойного ввода-вывода». Два контакта SPI используются для записи части адреса флэш-памяти и для чтения данных флэш-памяти: требуется вдвое меньше тактов по сравнению со стандартным SPI.
   2. DOUT – хост SPI использует «быстрое чтение с двумя выходами». Два контакта SPI используются для чтения данных флэш-памяти. Немного медленнее, чем DIO, поскольку адрес записывается через один вывод данных MOSI (Master Out Slave In).
   3. QIO – хост SPI использует технологию «Quad I/O Fast Read». Четыре контакта SPI используются для записи части адреса флэш-памяти и для чтения данных флэш-памяти: этим фазам требуется четверть тактового цикла по сравнению со стандартным SPI.
   4. QOUT – хост SPI использует функцию быстрого чтения с четырьмя выходами. Четыре контакта SPI используются для считывания данных флэш-памяти. Немного медленнее, чем QIO, поскольку адрес записывается через один вывод данных MOSI.
   5. При настройке флэш-памяти QIO сигналы SPI SPIHD (IO12) и SPIWP (IO13) мультиплексируются. Чтобы увеличить количество доступных GPIO, плата разработки выбирает режим DIO с использованием 2-проводного SPI и в этом случае контакты IO12 и IO13 не подключены к флэш-памяти.
4. Сигнал VDD внешней флэш-памяти SPI подключен к системе питания 3,3 В. Во время использования не требуется настраивать другие источники питания. Применяется общий 2-проводной режим связи SPI.
5. По умолчанию контакт VDD флэш-памяти SPI подключен к GPIO11, и для его использования в качестве GPIO необходимо выполнить его перенастройку.

### 13.2.2 Подготовка среды проектирования

Для проектирования под выбранный микроконтроллер необходимо иметь следующие инструменты:

* инструментарий для компиляции кода для ESP32;
* инструменты сборки проектов (CMake и Ninja) для создания полноценного приложения для ESP32;
* среду проектирования ESP-IDF, которая содержит API (библиотеки программного обеспечения и исходный код) для ESP32 и сценарии для работы с набором инструментов разработки.

В качестве сред разработки можно использовать среду на основе Eclipse или VSCode (Visual Studio Code) с соответствующими расширениями. В текущей работе будет использован VSCode, для подготовки среду необходимо выполнить его установку, воспользовавшись ссылкой [26]. После установки VSCode необходимо активизировать расширение ESP-IDF [27] для работы с микроконтроллером. По указанной ссылке находится подробная инструкция по установке расширения Espressif IDF.

После завершения установки рекомендуется открыть шаблон проекта командой Open Folder. В левой части окна VS Code (рис. 13.5) размещена панель управления, содержащая следующие основные элементы:

* Проводник, позволяющий переходить между файлами проекта.
* Поиск, реализующий поиск конкретных слов внутри документа.
* Система управления версиями, предназначенная для взаимодействия с Git, просмотра изменений в коде и разных его версий.
* Запуск кода, обеспечивающий запуск и проверку кода.
* Расширения, показывающий список расширений, которые можно дополнительно установить в VS Code.

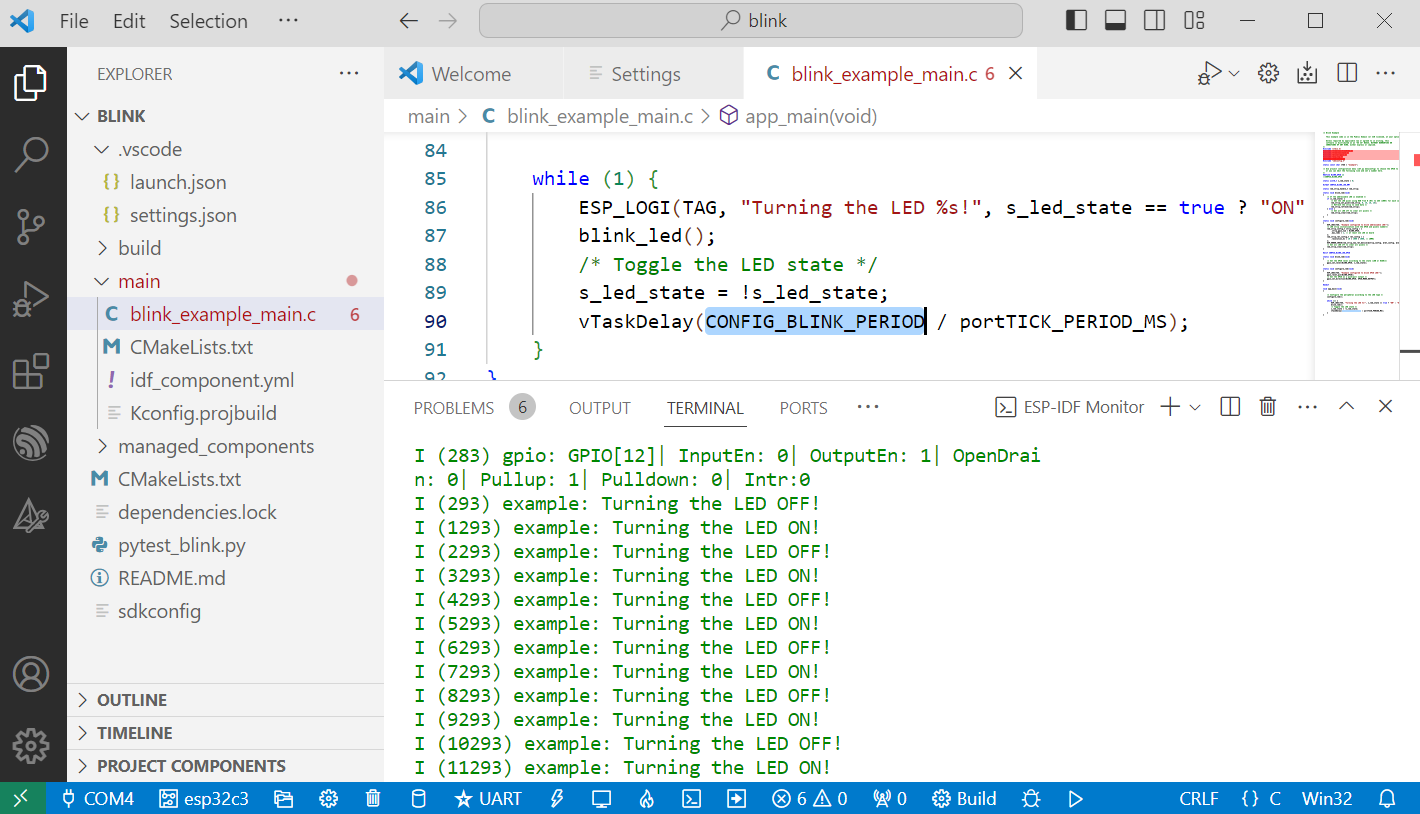
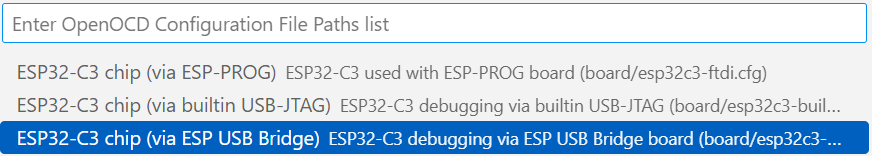


Рис. 13.5. Рабочее окно VS Code

В строке, расположенной внизу окна необходимо выполнить следующие настройки:

* Указать COM порт, к которому подключается плата. Определение номера порта происходит автоматически после подключения платы к компьютеру, которое осуществляется через USB порт. После активизации команды выбора COM порта появится выпадающий список, в котором будут показаны активные устройства с последовательным подключением.
* Указать микросхему, в которую осуществляется загрузка проекта, выбрав в выпадающем списке ESP32C3. Также указать способ загрузки, как показано на рис. 13.6.
* Выполнить сборку и загрузку проекта и запустить монитор для наблюдения результатов работы. Эти операции можно запускать соответствующими командами, закрепленными в строке за отдельными пиктограммами, а можно запустить весь процесс одной пиктограммой, интегрирующей три команды вместе.

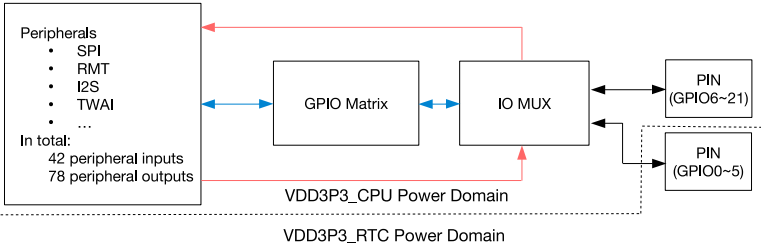


*Рис*. 13.6. Меню выбора способа загрузки

После загрузки проекта в плату в окне монитора можно отслеживать статус процессов сборки и загрузки, а также контролировать процесс работы программы.

### 13.2.3 GPIO

На рис. 13.7 показана общая схема работы IO MUX и матрицы GPIO.



*Рис*. 13.7. IO MUX и матрица GPIO

Матрица GPIO обеспечивает переключение между входными/выходными сигналами периферийных устройств и контактами. 42 входных сигнала периферийных устройств могут быть подключены к любым выводам GPIO, также, как и любой из 78 периферийных выходных сигналов могут быть скоммутированы на любой контакт GPIO. Поддерживается синхронизация сигналов периферийных входов на основе тактовой шины APB.

IO MUX конфигурируется через регистр IO\_MUX\_GPIOn\_REG для каждого вывода GPIO. Контакт может быть настроен на функцию, маршрутизируемую матрицей GPIO, или подключен, минуя матрицу GPIO. Обход матрицы актуален для улучшения частоты работы высокоскоростных цифровых сигналов (SPI, JTAG, UART), поскольку в этом случае IO MUX используется для прямого подключения выводов к периферийным устройствам.

Электрический интерфейс между логикой микросхемы и выводом GPIO показан на рис. 13.8. Схема управляется с помощью сигналов разрешения входа (IE), разрешение выхода (OE), WPU и WPD подключения подтягивающих резисторов (pull-up и pull-down). Bonding pad – точка, используемая для физического подключения от корпуса микросхемы к выводу GPIO.

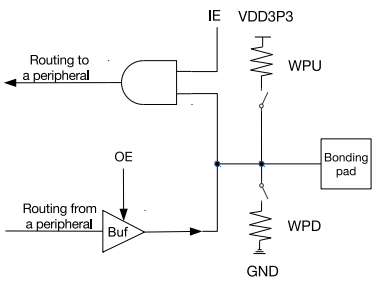


Рис. 13.8. Внутренняя структура подключения к контакту микросхемы

### 13.2.4 Ввод данных в периферийные устройства через GPIO матрицу

Для считывания данных с контакта GPIO X в периферийный сигнал Y, необходимо выполнить следующие действия:

1. Настроить регистр GPIO\_FUNCY\_IN\_SEL\_CFG\_REG, соответствующий периферийному сигналу Y в матрице GPIO:

– установить GPIO\_SIGY\_IN\_SEL, чтобы разрешить ввод периферийного сигнала через матрицу GPIO;

– установить GPIO\_FUNCX\_IN\_SEL на нужный вывод GPIO.

Некоторые периферийные сигналы не имеют действительного бита GPIO\_SIGy\_IN\_SEL, а именно, эти периферийные устройства могут только получать входные сигналы через матрицу GPIO.

2. Дополнительно можно включить фильтр для входных сигналов, установив регистр IO\_MUX\_GPIOX\_FILTER\_EN. Фильтровать можно только сигналы длительностью более двух тактов.

3. Синхронизировать вход GPIO, установив GPIO\_PINX\_REG, соответствующий выводу GPIO X, следующим образом:

– установить GPIO\_PINX\_SYNC1\_BYPASS, чтобы разрешить входной сигнал, синхронизированный по нарастающему или по спадающему фронту;

– установить GPIO\_PINX\_SYNC2\_BYPASS, чтобы разрешить входной сигнал, синхронизированный по нарастающему или спадающему фронту во втором такте.

4. Настроить регистр IO MUX для включения входного сигнала внешнего контакта. Для этого установить IO\_MUX\_GPIOX\_REG соответствующий выводу GPIO X, следующим образом:

– установить IO\_MUX\_GPIOX\_FUN\_IE для разрешения входа;

– установить или снять IO\_MUX\_GPIOX\_FUN\_WPU и IO\_MUX\_GPIOX\_FUN\_WPD, чтобы включить или отключить подтягивающие и отводящие резисторы.

Пример: подключение контакта I2S\_MCLK\_in, индекс сигнала 12 к GPIO7 (MTDO), будет выглядеть следующим образом:

– установить значение GPIO\_SIG12\_IN\_SEL в регистре GPIO\_FUNC12\_IN\_SEL\_CFG\_REG, чтобы разрешить ввод периферийного сигнала через матрицу GPIO.

– установить значение GPIO\_FUNC12\_IN\_SEL в регистре GPIO\_FUNC12\_IN\_SEL\_CFG\_REG равным 7.

– установить IO\_MUX\_GPIO7\_FUN\_IE в регистре IO\_MUX\_GPIO7\_REG, чтобы разрешить ввод пинов.

### 13.2.5 Вывод данных из периферийных устройств через GPIO матрицу

Для вывода сигнала Y из периферийного устройства в порт X GPIO, а затем в IO MUX и далее на внешний контакт нужно выполнить следующие действия.

1. Сконфигурировать регистры GPIO\_FUNCX\_OUT\_SEL\_CFG\_REG и GPIO\_ENABLE\_REG[X], соответствующие конкретному выводу GPIO в матрице GPIO (рекомендуется использовать регистры W1TS (запись 1 для установки) и W1TC (запись 1 для сброса) для установки или сброса GPIO\_ENABLE\_REG.

– установить поле GPIO\_FUNCX\_OUT\_SEL в регистре GPIO\_FUNCX\_OUT\_SEL\_CFG\_REG на индекс нужного периферийного выходного сигнала Y.

– если сигнал должен быть всегда включен как выходной, нужно установить бит GPIO\_FUNCX\_OEN\_SEL в регистре GPIO\_FUNCX\_OUT\_SEL\_CFG\_REG и бит в регистре GPIO\_ENABLE\_W1TS\_REG, соответствующий выводу GPIO X. Чтобы сигнал разрешения выхода определялся внутренней логикой.

– установить соответствующий бит в регистре GPIO\_ENABLE\_W1TC\_REG, чтобы отключить выходной сигнал с контакта GPIO вывод.

2. Для выхода с открытым стоком установить бит GPIO\_PINX\_PAD\_DRIVER в регистре GPIO\_PINX\_REG, соответствующем выводу GPIO X.

3. Настроить регистр IO MUX для включения выхода через матрицу GPIO. Установить IO\_MUX\_GPIOX\_REG, соответствующий выводу GPIO X, следующим образом:

– Установите в поле IO\_MUX\_GPIOX\_MCU\_SEL желаемую функцию IO MUX, соответствующую выводу GPIO X. Эта функция GPIO должна иметь числовое значение 1, для всех выводов.

– Установите в поле IO\_MUX\_GPIOX\_FUN\_DRV требуемое значение силы тока выходного сигнала (0 ~ 3). Чем выше числовое значение, тем больший ток может быть получен от внешнего контакта. При настройках следует ориентироваться на следующие значения

– 0: ~5 мА

– 1: ~10 мА

– 2: ~20 мА (значение по умолчанию)

– 3: ~40 мА

Если используется режим открытого стока, следует установить/очистить биты IO\_MUX\_GPIOX\_FUN\_WPU и IO\_MUX\_GPIOX\_FUN\_WPD для включения/выключения внутренних подтягивающих/отключающих резисторов.

***13.2.6 Прямой ввод и вывод через IO MUX***

Некоторые высокоскоростные сигналы (SPI и JTAG) могут обходить матрицу GPIO для улучшения высокочастотных цифровых характеристик. В этом случае IO MUX используется для прямого подключения этих выводов к периферийным устройствам.

Этот вариант менее гибок, чем маршрутизация сигналов через матрицу GPIO, поскольку регистр IO MUX для каждого вывода GPIO может только выбрать из ограниченного числа функций, но при этом можно улучшить высокочастотные цифровые характеристики.

Для обхода матрицы GPIO для входных периферийных сигналов необходимо настроить два регистра:

1. IO\_MUX\_GPIOn\_MCU\_SEL для пина GPIO должен быть установлен на требуемую функцию пина. Список функций контактов функций см. в разделе 5.12.

2. Снять значение GPIO\_SIGn\_IN\_SEL, чтобы направить входной сигнал непосредственно на периферийное устройство.

Чтобы обойти матрицу GPIO для выходных сигналов периферийного устройства, IO\_MUX\_GPIOn\_MCU\_SEL для пина GPIO должен быть установлен на требуемую функцию пина.

***13.2.7 Аналоговые функции выводов GPIO***

Некоторые выводы GPIO в ESP32-C3 обеспечивают аналоговые функции. Если вывод используется для аналоговых целей, необходимо отключить подтягивающие и понижающие резисторы, выполнив следующие действия:

1. Установить IO\_MUX\_GPIOn\_MCU\_SEL в 1, и очистить IO\_MUX\_GPIOn\_FUN\_IE, IO\_MUX\_GPIOn\_FUN\_WPU, IO\_MUX\_GPIOn\_FUN\_WPD.

2. Записать 1 в GPIO\_ENABLE\_W1TC[n], чтобы снять разрешение на вывод.

Настройка регистров осуществляется с использованием функций API. При выполнении работы будут использованы следующие функции:

1. gpio\_reset\_pin (gpio\_num\_t gpio\_num) сбрасывает настройки пина gpio\_num к стандартным (если на этом пине есть возможность подключения к разным периферийным устройствам, то выбрана функция GPIO, подключено подтягивание, отключен ввод и вывод). Её необходимо использовать перед настройкой.
2. gpio\_set\_direction (gpio\_num\_t gpio\_num, gpio\_mode\_t mode) устанавливает на пин gpio\_num режим работы mode. Доступные режимы:
   1. GPIO\_MODE\_DEF\_DISABLE – ввод и вывод отключены;
   2. GPIO\_MODE\_DEF\_INPUT – только ввод;
   3. GPIO\_MODE\_DEF\_OUTPUT – только вывод;
   4. GPIO\_MODE\_DEF\_OD – выход с открытым коллектором.
   5. Режимы можно комбинировать, например, режим, заданный как (GPIO\_MODE\_DEF\_INPUT) | (GPIO\_MODE\_DEF\_OUTPUT), разрешает чтение и запись.
3. gpio\_set\_level (gpio\_num\_t gpio\_num, uint32\_t level) устанавливает заданный уровень level на выход пина gpio\_num. Уровень может принимать значения 0(low) или 1(high).
4. vTaskDelay (500/portTICK\_PERIOD\_MS) откладывает выполнение программы на заданное количество тиков. Для перевода значения в миллисекунды его нужно разделить на константу portTICK\_PERIOD\_MS.

## 13.3 Задания на лабораторную работу

Подготовить программу, обеспечивающую управление светодиодами в соответствии с заданным вариантом.

1. Обеспечить включение/выключение двух диодов в противофазе с частотой 1 Гц.
2. Обеспечить включение светодиода с изменяющейся частотой от 1 Гц до 50 Гц с равномерным шагом 5 Гц.
3. Обеспечить включение светодиода с постоянной частотой 50 Гц с равномерным шагом и изменяющейся от 20% до 80% скважностью.

### 13.3.1 Описание последовательности выполнения работы

1. Установить среду проектирования в соответствии с рекомендациями раздела «Подготовка среды проектирования».

2. Открыть шаблон проекта командой Open Folder. В качестве шаблона для первого проекта можно использовать проект Sample\_project, находящийся в каталоге esp\ esp-idf\examples\get-started\

3. Подготовить текст основной программы в соответствии с заданием, используя шаблон, реализующий мигание светодиода и приведенный в п. 13.3.2.

### 13.3.2 Пример выполнения задания на защиту

#include <stdio.h>

#include "freertos/FreeRTOS.h"

#include "freertos/task.h"

#include "driver/gpio.h"

#include "led\_strip.h"

#define LED1 12

static uint8\_t s\_led\_state;

static void blink\_led(void)

{

gpio\_set\_level(LED1, s\_led\_state);

}

static void configure\_led(void)

{

gpio\_reset\_pin(LED1);

gpio\_set\_direction(LED1, GPIO\_MODE\_OUTPUT);

}

void app\_main(void)

{

s\_led\_state = 0;

configure\_led();

while (1) {

blink\_led();

s\_led\_state = !s\_led\_state;

vTaskDelay(500/portTICK\_PERIOD\_MS);

}

}

## 13.4 Вопросы для контроля

1. Какие существуют варианты коммутации выходов периферийных устройств с внешними контактами?
2. Как осуществляется настройка внешних контактов для реализации альтернативных функций?
3. Какие режимы доступны для внешних контактов микросхемы, как их можно изменять?

# Лабораторная работа № 14 **Работа с LED PWM контроллером**

## 14.1 Цель и задачи

Цель работы: изучить порядок настройки и использования встроенного контроллера LED PWM. Задачи:

1. Получить навыки управления светодиодами с использованием широтно-импульсно модулированных (ШИМ) сигналов.
2. Разработать программу для управления и индикации через светодиоды платы.

## 14.2 Краткие теоретические сведения

Контроллер LED PWM – это периферийное устройство, предназначенное для генерации ШИМ сигналов для управления светодиодами, однако выходные ШИМ сигналы контроллера могут использоваться и для других целей.

Контроллер LED PWM включает в себя четыре независимых таймера (Time0 – Timer3) с поддержкой деления на доли, мультиплексор и шесть независимых ШИМ-генераторов (PWM0 – PWM5) с разрешением 14 бит (рис. 14.1).

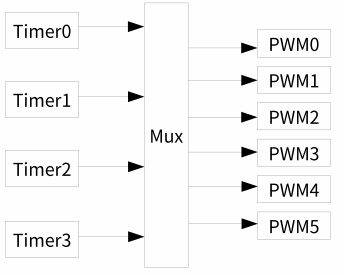
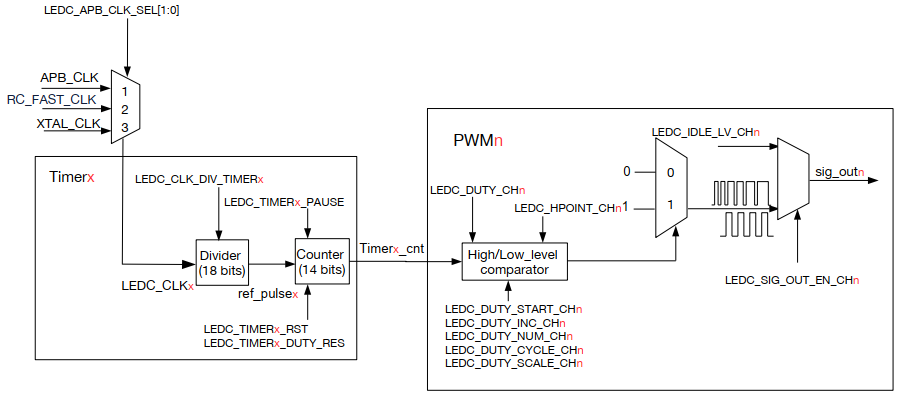


Рис. 14.1. Архитектура LED PWM контроллера

Контроллер может реализовывать затухание рабочего цикла (т.е. постепенное увеличение/уменьшение рабочего цикла ШИМ без вмешательства процессора) с генерацией прерывания по завершении затухания, позволяет регулировать фазу выходного ШИМ-сигнала, допускает перевод вывода ШИМ-сигнала в режим пониженного энергопотребления.

На рисунке 14.2 показаны настраиваемые элементы контроллера.

Каждый таймер в контроллере LED PWM реализует счет времени на основе тактового сигнала ref\_pulseX. Все таймеры используют один и тот же источник тактового сигнала LEDC\_CLKX, который затем пропускается через делитель тактового сигнала, чтобы сформировать ref\_pulseX для счетчика.



*Рис*. 14.2. Настройка элементов PWM контроллера

Настройка канала для генерации ШИМ-сигнала проводится в три последовательных этапа:

1. Инициализация таймера;
2. Инициализация канала (подключение к нему таймера и GPIO);
3. Задание значения на выходе GPIO.

Для выполнения этих действий можно использовать следующие API функции и структуры.

Для настройки таймера в функцию ledc\_timer\_config нужно передать заполненную структуру ledc\_timer\_config\_t, которая состоит из следующих элементов:

struct ledc\_timer\_config\_t {

ledc\_timer\_bit\_t duty\_resolution,

uint32\_t freq\_hz,

ledc\_mode\_t speed\_mode,

ledc\_timer\_t timer\_num,

ledc\_clk\_cfg\_t clk\_cfg

}

* timer\_num – номер таймера, число от 0 до 3.
* clk\_cfg. Для автоматического выбора параметра можно установить его в значение LEDC\_AUTO\_CLK.
* speed\_mode. Для ESP32-C3 этот параметр может быть установлен только в значение LEDC\_LOW\_SPEED\_MODE.
* duty\_resolution – разрешение канала, количество бит. Например, при значении duty\_resolution 10, можно устанавливать интенсивность сигнала (duty) в значения от 0 до (2\*\*10–1).
* freq\_hz – частота таймера. Значения частоты таймера и разрешения канала (duty\_resolution) зависят друг от друга: при увеличении частоты необходимо понизить разрешение. Например, при частоте 40 мегагерц допустимое разрешение составляет только 1 бит, а при 5 килогерц – 13 бит.

Далее нужно настроить канал.

ledc\_channel\_config\_t {

ledc\_channel\_t channel,

uint32\_t duty,

int gpio\_num,

ledc\_mode\_t speed\_mode,

int hpoint,

ledc\_timer\_t timer\_sel

}

* channel – номер канала от 0 до 5.
* duty – интенсивность сигнала, от 0 до (2\*\*duty\_resolution-1).
* gpio\_num – номер пина.
* timer\_sel – номер выбранного таймера, аналогичный номеру timer\_num в структуре ledc\_timer\_config\_t.
* speed\_mode – режим, аналогичный выбранному в структуре ledc\_timer\_config\_t.

Чтобы эти изменения установились, как и в случае с таймером структура передается в специальную функцию ledc\_timer\_config. Сразу после этого на выбранном пине начинает генерироваться заданный сигнал.

Для изменения параметра duty последовательно используются функции ledc\_set\_duty(ledc\_mode\_t speed\_mode, ledc\_channel\_t channel, uint32\_t duty) и ledc\_update\_duty(ledc\_mode\_t speed\_mode, ledc\_channel\_t channel).

## 14.3 Задание к лабораторной работе

Подготовить программу, обеспечивающую управление светодиодами с использованием LED PWM контроллера, в соответствии с заданным вариантом. С использованием беспаечной макетной платы собрать схему, подключив светодиоды к настроенным на связь с выходами LED PWM внешним контактам.

1. Реализовать плавное уменьшение яркости светодиода до полного его выключения в течение 10 секунд после запуска программы.
2. Реализовать плавное увеличение яркости светодиода до его максимальной яркости в течение 10 секунд после запуска программы.
3. Реализовать циклическое плавное уменьшение с последующим увеличением яркости. Длительность цикла – 10 секунд.

### 14.2.1 Описание последовательности выполнения работы

Для выполнения работы необходимо иметь беспаечную макетную плату для многократного макетирования; одножильные провода-перемычки с контактами, допускающими коммутацию с установленными на макетной плате элементами; светодиоды.

1. Открыть шаблон проекта командой Open Folder. В качестве шаблона для первого проекта можно использовать проект Sample\_project, находящийся в каталоге esp\ esp-idf\examples\get-started\

2. Подготовить текст основной программы в соответствии с заданием, используя шаблон, реализующий формирование ШИМ сигнала с использованием контроллера и приведенный в п. 14.2.2.

### 14.2.2 Пример выполнения задания на защиту

#include <stdio.h>

#include "freertos/FreeRTOS.h"

#include "freertos/task.h"

#include "driver/ledc.h"

#include "esp\_err.h"

//Для подключения таймера

#define LEDC\_LS\_TIMER LEDC\_TIMER\_1

#define LEDC\_LS\_MODE LEDC\_LOW\_SPEED\_MODE

//Для подключения первого светодиода

#define LEDC\_LS\_CH0\_GPIO (8)

#define LEDC\_LS\_CH0\_CHANNEL LEDC\_CHANNEL\_0

//Для подключения второго светодиода

#define LEDC\_LS\_CH1\_GPIO (5)

#define LEDC\_LS\_CH1\_CHANNEL LEDC\_CHANNEL\_3

#define LEDC\_TEST\_CH\_NUM (2) //число светодиодов

#define LEDC\_TEST\_DUTY (8191)//максимальная интенсивность

void app\_main(void)

{

int ch;

/\*Конфигурация таймера\*/

ledc\_timer\_config\_t ledc\_timer = {

.duty\_resolution = LEDC\_TIMER\_13\_BIT,

.freq\_hz = 5000,

.speed\_mode = LEDC\_LS\_MODE,

.timer\_num = LEDC\_LS\_TIMER,

.clk\_cfg = LEDC\_AUTO\_CLK,

};

ledc\_timer\_config(&ledc\_timer);

/\*Конфигурация двух ШИМ-каналов\*/

ledc\_channel\_config\_t ledc\_channel[LEDC\_TEST\_CH\_NUM] = {

{

.channel = LEDC\_LS\_CH0\_CHANNEL,

.duty = 0,

.gpio\_num = LEDC\_LS\_CH0\_GPIO,

.speed\_mode = LEDC\_LS\_MODE,

.hpoint = 0,

.timer\_sel = LEDC\_LS\_TIMER,

.flags.output\_invert = 0

},

{

.channel = LEDC\_LS\_CH1\_CHANNEL,

.duty = 0,

.gpio\_num = LEDC\_LS\_CH1\_GPIO,

.speed\_mode = LEDC\_LS\_MODE,

.hpoint = 0,

.timer\_sel = LEDC\_LS\_TIMER,

.flags.output\_invert = 1 /\*1 – сигнал на выходе инвертируется\*/

},

};

for (ch = 0; ch < LEDC\_TEST\_CH\_NUM; ch++) {

ledc\_channel\_config(&ledc\_channel[ch]);

}

while (1) {

/\*Установка и обновление интенсивности для каждого канала\*/

printf(" LEDC set duty from %d to 0 without autofade\n", LEDC\_TEST\_DUTY);

for(int i = 0; i<LEDC\_TEST\_DUTY; i+= 40){

for (ch = 0; ch < LEDC\_TEST\_CH\_NUM; ch++) {

ledc\_set\_duty(ledc\_channel[ch].speed\_mode, ledc\_channel[ch].channel, LEDC\_TEST\_DUTY-i);

ledc\_update\_duty(ledc\_channel[ch].speed\_mode, ledc\_channel[ch].channel);

}

vTaskDelay(10 / portTICK\_PERIOD\_MS);

}

vTaskDelay(100 / portTICK\_PERIOD\_MS);

}

}

## 14.4 Вопросы для контроля

1. Поясните правила управления яркостью светодиода с использованием ШИМ сигнала.
2. Какие действия необходимо выполнить для настройки LED PWM контроллера?
3. Как осуществляется настройка таймера? Какую функцию выполняет предварительный делитель?

# Лабораторная работа № 15 **Настройка GPIO для ввода аналогового сигнала**

## 15.1 Цель и задачи

Цель работы: изучить правила настройки портов ввода-вывода микроконтроллера ESP32-C3 на реализацию альтернативных функций. Задачи:

1. Освоить прием аналогового сигнала и его оцифровку с использованием встроенного аналого-цифрового преобразователя.
2. Получить навыки подключения встроенного температурного сенсора.

## 15.2 Краткие теоретические сведения

ESP32-C3 включает в себя 2 АЦП последовательного приближения, поддерживающие в общей сложности 6 каналов измерения, которые обеспечиваются следующим образом:

1. АЦП1 допускает подключение 5 каналов: GPIO0 – GPIO4
2. АЦП2 допускает подключение 1 канала: GPIO5.

АЦП ESP32 оцифровывают напряжения в диапазоне от 0 В до Vref. Vref – это опорное напряжение, у разных микросхем Vref различается, среднее значение составляет 1,1 В. Чтобы преобразовать напряжения, превышающие Vref, входные напряжения можно ослабить перед подачей на АЦП. Доступны 4 варианта затухания: чем выше затухание, тем выше может быть измеряемое входное напряжение.

Контроллер поддерживает частоту дискретизации до 12 бит, однократное считывание (single read mode), запускаемое с помощью ПО и непрерывное сканирование (continuous read mode), запускаемое по таймеру. Режим однократного чтения подходит для низкочастотных измерений. Режим непрерывного чтения подходит для постоянно повторяющихся с высокой частотой измерений.

Для настройки однократного считывания с помощью ПО, необходимо:

1. Выбрать АЦП1 или АЦП2:

– если установлен APB\_SARADC1\_ONETIME\_SAMPLE, выбран АЦП1;

– если установлен APB\_SARADC2\_ONETIME\_SAMPLE, выбран АЦП2;

2. Настроить APB\_SARADC\_ONETIME\_CHANNEL, чтобы выбрать один канал.

3. Настроить APB\_SARADC\_ONETIME\_ATTEN, чтобы установить затухание.

4. Установить APB\_SARADC\_ONETIME\_START, чтобы начать считывание.

5. По окончанию считывания, генерируется прерывание APB\_SARADC\_ADCX\_DONE\_INT\_RAW, которое может использоваться для считывания данных из APB\_SARADC\_ADCX\_DATA. X может принимать значение 1:(АЦП1), или 2:(АЦП2).

Для настройки непрерывного сканирования, нужно выполнить следующие действия:

1. Установить APB\_SARADC\_TIMER\_TARGET для выбора DIG АЦП в качестве цели для запуска таймера.

2. Установить APB\_SARADC\_TIMER\_EN для запуска таймера.

3. Когда время таймера истекает, он заставляет DIG ADC FSM начать измерения в соответствии с таблицей шаблонов.

4. Значения автоматически помещаются в память с помощью DMA (прямой доступ в память). Прерывание генерируется после завершения считывания.

В контроллер ESP32-C3 встроен датчик температуры, позволяющий отслеживать изменения температуры внутри чипа. К нему можно обратиться программно для получения текущих значений. В зависимости от температуры окружающей среды настраивается диапазон измерений и смещение, которое позволяет повысить точность датчика.

Получить данные с датчика температуры можно следующим образом:

1. Установить APB\_SARADC\_TSENS\_PU, чтобы запустить XPD\_SAR, а затем включить датчик температуры.

2. Установить SYSTEM\_TSENS\_CLK\_EN, чтобы запустить тактовый сигнал датчика.

3. Ждать тактов APB\_SARADC\_TSENS\_XPD\_WAIT, пока не будет выполнен сброс датчика температуры. После этого начнется измерение температуры;

4. Через некоторое время данные можно будет считать из APB\_SARADC\_TSENS\_OUT. Для получения значений в градусах Цельсия, нужно использовать формулу T(°C) = 0.4386 ∗ VALUE–27.88 ∗ offset–20.52, где значение offset определяется по таблице 15.1.

| Диапазон измерений, °C | Смещение, °C |
| --- | --- |
| 50 ~ 125 | –2 |
| 20 ~ 100 | –1 |
| -10 ~ 80 | 0 |
| -30 ~ 50 | 1 |
| -40 ~ 20 | 2 |

Таблица 15.1. Смещения для датчика температуры

Настройка регистров осуществляется с использованием функций API.

Для подключения контакта к АЦП1 последовательно вызываются две функции:

* adc1\_config\_width (adc\_bits\_width\_t width\_bit) – настройка разрешения канала, width\_bit – число бит. Для ESP32-C3 доступно 12-ти битное разрешение.
* adc1\_config\_channel\_atten (adc1\_channel\_t channel, adc\_atten\_t atten) – выбор канала (channel) и уровня ослабления (atten). Благодаря этому можно изменять верхнюю границу измеряемого диапазона для увеличения точности значений.

Доступные значения adc\_atten\_t:

* ADC\_ATTEN\_DB\_0 (или значение 0) 0 mV ~ 750 mV
* ADC\_ATTEN\_DB\_2\_5 (1) 0 mV ~ 1050 mV
* ADC\_ATTEN\_DB\_6 (2) 0 mV ~ 1300 mV
* ADC\_ATTEN\_DB\_11 (3) 0 mV ~ 2500 mV

Для одиночного считывания данных вызывается следующая функция adc1\_get\_raw(adc1\_channel\_t channel), в которой указывается номер канала, выбранный в предыдущей функции. На выходе получаем целое значение val в диапазоне от 0 до (2\*\*width\_bit–1)

При необходимости получить значение напряжения можно использовать формулу mVolts = val\*(Vref/4095), Vref – верхнее значение выбранного диапазона adc\_atten.

Для получения данных со встроенного датчика температуры выполняются следующие действия:

1. После создания пустого объекта типа temperature\_sensor\_handle\_t необходимо определить, в каких условиях датчик будет работать, и задать их верхнюю и нижнюю температурные границы.
2. Структуру temperature\_sensor\_config\_t можно задать функцией TEMPERATURE\_SENSOR\_CONFIG\_DEFAULT (min, max), где min и max – границы температуры. Ссылка на эту структуру вместе с указателем на объект передается в функцию temperature\_sensor\_install(). Далее для начала работы с датчиком вызывается temperature\_sensor\_enable(), в которую также нужно передать указатель на созданный temperature\_sensor\_handle\_t.
3. Для получения значений температуры используется функция temperature\_sensor\_get\_celsius(temperature\_sensor\_handle\_t tsens, float \*out\_celsius), в out\_celsius будет храниться значение в градусах Цельсия.

## 15.3 Задания к лабораторной работе

Подготовить программу, обеспечивающую прием аналогового сигнала, его оцифровку и использование полученного цифрового кода для управления внешним элементом. С использованием беспаечной макетной платы собрать схему, подключив к аналоговому входу переменный резистор, а к цифровому выходу – элемент, управляемый цифровым сигналом.

1. Реализовать плавное изменение яркости светодиода в зависимости от значения сопротивления переменного резистора.
2. Реализовать плавное изменение частоты звучания зуммера в зависимости от значения сопротивления переменного резистора.
3. Реализовать вывод на экран компьютера оцифрованных значений сигнала, получаемого от переменного резистора.

### 15.3.1 Описание последовательности выполнения работы

Для выполнения работы необходимо иметь беспаечную макетную плату для многократного макетирования; одножильные провода-перемычки с контактами, допускающими коммутацию с установленными на макетной плате элементами; светодиоды; зуммер.

1. Изучить настройки портов GPIO для подключения аналогового сигнала.

2. Для выполнения работы рекомендуется воспользоваться шаблоном проекта ADC. Подготовить текст основной программы в соответствии с заданием, используя шаблон, приведенный в п.15.2.3. Доработать программу шаблона таким образом, чтобы входной аналоговый сигнал преобразовывался в цифровой эквивалент, который, в свою очередь, использовался для управления внешним элементом (зуммером, светодиодом).

3. Используя беспаечную макетную плату, подключить к аналоговому входу резистор с переменным напряжением, а к цифровому выходу элемент, управляемый от цифрового выхода.

### 15.3.2 Пример выполнения задания на защиту

#include <stdio.h>

#include "freertos/FreeRTOS.h"

#include "freertos/task.h"

#include "driver/adc.h"

#include "esp\_adc\_cal.h"

#include "driver/temperature\_sensor.h"

void app\_main(void){

//подключаемся к каналу АЦП

adc1\_config\_width(12);

adc1\_config\_channel\_atten(ADC1\_CHANNEL\_2, ADC\_ATTEN\_DB\_11);

//подключаем встроенный датчик температуры

temperature\_sensor\_handle\_t temp\_sensor = NULL;

temperature\_sensor\_config\_t temp\_sensor\_config = TEMPERATURE\_SENSOR\_CONFIG\_DEFAULT(10, 50);

ESP\_ERROR\_CHECK(temperature\_sensor\_install(&temp\_sensor\_config, &temp\_sensor));

ESP\_ERROR\_CHECK(temperature\_sensor\_enable(temp\_sensor));

float tsens\_value;

while(1){

//получение значений температуры с внешнего датчика

int val = adc1\_get\_raw(ADC1\_CHANNEL\_2);

float voltage = val\*(2500.0/4096.0);

float temp = (voltage-500)/10.0; //для датчика TMP36

printf("Temperature in the room is %.02f ℃ \n", temp);

//получение значений температуры со встроенного датчика

ESP\_ERROR\_CHECK(temperature\_sensor\_get\_celsius(temp\_sensor, &tsens\_value));

printf("Temperature value inside the chip is %.02f ℃\n\n", tsens\_value);

vTaskDelay(2000/portTICK\_PERIOD\_MS);

}

}

## 15.4 Вопросы для контроля

1. Поясните порядок настройки аналого-цифрового преобразователя.
2. Поясните режимы работы АЦП, для каких задач они могут быть использованы?
3. С помощью каких настроек можно регулировать разрешающую способность АЦП?

# СПИСОК ОСНОВНОЙ ЛИТЕРАТУРЫ

1. Таненбаум Э. С., Херберт Б. Современные операционные системы. 4-е изд. – " Издательский дом"" Питер""", 2021.
2. VirtualBox. URL: <https://www.virtualbox.org/>
3. ssh(1) — Linux manual page. URL: <https://man7.org/linux/man-pages/man1/ssh.1.html>
4. An Introduction to Assembly Programming with RISC-V. Prof. Edson Borin, Institute of Computing, Unicamp URL: <https://riscv-programming.org/book/riscv-book.html>
5. RISC-V ASM manual. URL: <https://github.com/riscv-non-isa/riscv-asm-manual/blob/master/riscv-asm.md>
6. GOTO statement. URL: <https://en.cppreference.com/w/c/language/goto>
7. Сара Л. Харрис, Дэвид Харрис Цифровая схемотехника и архитектура компьютера: RISC-V / пер. с англ. В. С. Яценкова, А. Ю. Романова; под ред. А. Ю. Романова. – М.: ДМК Пресс, 2021.
8. Harry H. Porter III RISC-V:An Overview of the Instruction Set Architecture. URL: [www.cs.pdx.edu/~harry/riscv](http://www.cs.pdx.edu/~harry/riscv)
9. FreeRTOS: Real-time operating system for microcontrollers. URL: <https://www.freertos.org/>
10. Введение в компиляторные оптимизации. Перевод курса RISC-V Toolchain and Compiler Optimization Techniques (LFD113x). URL: <https://github.com/iakov/riscv-courses/blob/main/LFD113x-RU/Chapters/Chapter2.adoc>
11. gcc(1) — Linux manual page. URL: <https://man7.org/linux/man-pages/man1/gcc.1.html>
12. Perf. URL: <https://perf.wiki.kernel.org/index.php/Main_Page>
13. Valgrind. URL: <https://valgrind.org/>
14. callgrind. URL: <https://valgrind.org/docs/manual/cl-manual.html>
15. cachegrind. URL: <https://valgrind.org/docs/manual/cg-manual.html>
16. time. URL: <https://man7.org/linux/man-pages/man2/time.2.html>
17. Basic operating systems terms and concepts. URL: <https://linux-kernel-labs.github.io/refs/heads/master/lectures/intro.html>
18. ls. URL: <https://man7.org/linux/man-pages/man1/ls.1.html>
19. size. URL: <https://man7.org/linux/man-pages/man1/size.1.html>
20. Методы отладки. Перевод курса RISC-V Toolchain and Compiler Optimization Techniques (LFD113x). URL: <https://github.com/iakov/riscv-courses/blob/main/LFD113x-RU/Chapters/Chapter4.adoc>
21. gdb. URL: <https://man7.org/linux/man-pages/man1/gdb.1.html>
22. GNU General Public License (GPL). URL: <https://www.gnu.org/licenses/gpl-3.0.ru.html>
23. FreeRTOS / «Си для встраиваемых систем», онлайн-версия книги robotclass.ru. URL: <https://robotclass.ru/articles/c_book_freertos/>
24. Download FreeRTOS. URL: <https://www.freertos.org/a00104.html>
25. xSemaphoreCreateMutex. URL: <https://www.freertos.org/CreateMutex.html>
26. Установка среды VSCode. URL: <https://code.visualstudio.com/docs/?dv=win64user>
27. Установка расширения ESP-IDF для Visual Studio Code. URL: <https://github.com/espressif/vscode-esp-idf-extension/blob/master/docs/tutorial/install.md>
28. syscalls. URL: <https://man7.org/linux/man-pages/man2/syscalls.2.html>
29. Linux syscalls numbers for riscv64. URL: <https://gpages.juszkiewicz.com.pl/syscalls-table/syscalls.html>
30. C code obfuscator. URL: <https://github.com/whoward3/C-Code-Obfuscator>

# СПИСОК ДОПОЛНИТЕЛЬНОЙ ЛИТЕРАТУРЫ ДЛЯ САМОСТОЯТЕЛЬНОГО ИЗУЧЕНИЯ

1. [Computer Organization and Design The Hardware Software Interface: RISC-V Edition](http://staff.ustc.edu.cn/~llxx/cod/reference_books/Computer%20Organization%20and%20Design%20RISC-V%20edition.pdf)
2. [The RISC-V Reader: An Open Architecture Atlas](http://riscvbook.com/)
3. [Digital Design and Computer Architecture, RISC-V Edition aka «H&H»](https://www.amazon.com/Digital-Design-Computer-Architecture-RISC-V-ebook/dp/B099WJ2TZV/ref=sr_1_1?keywords=Digital+Design+and+Computer+Architecture%3A+RISC-V+Edition&qid=1637686816&qsid=145-1056019-7140512&s=digital-text&sr=1-1&sres=B099WJ2TZV%2CB08TRLDR2Q%2CB076HPP6R3&srpt=ABIS_BOOK)
4. [Цифровая схемотехника и архитектура компьютера: RISC-V](https://dmkpress.com/catalog/electronics/circuit_design/978-5-97060-961-3/)
5. [RISC-V Assembly Programming — учебник](https://riscv-programming.org/)