А.Ю. Романов, М.Т. Омаров

Программирование на языке ассемблера RISC-V: практикум по MIK32 Амур

Москва, 2026

УДК: 004.43

ББК: 32.973.2-018.1

Авторы:

Романов А. Ю. – докт. техн. наук, доцент, профессор Московского института электроники и математики им. А. Н. Тихонова Национального исследовательского университета «Высшая школа экономики».

Омаров М.Т. – аспирант 1 г.о. Московского института электроники и математики им. А. Н. Тихонова Национального исследовательского университета «Высшая школа экономики».

А.Ю. Романов, М.Т. Омаров

Программирование на языке ассемблера RISC-V: практикум по МІК32 Амур. М.: ДМК Пресс, 2026. – 115 с.: ил.

Перед вами практикум, дополняющий переведенную и адаптированную для русскоязычного читателя книгу Стивена Смита «Программирование на языке ассемблера RISC-V» (пер. с англ. А. В. Логунова; под науч. ред. А. Ю. Романова. – М.: ДМК Пресс, 2025. – 276 с.: ил. – (Книжная полка Истового инженера)). Если основная книга знакомит с архитектурой набора инструкций RISC-V и всеми группами инструкций языка ассемблера, то данный практикум переносит эти знания на реальную отечественную аппаратную платформу — микроконтроллер МІК32 Амур (К1948ВК015/К1948ВК018) производства АО «Микрон».

В книге описан полный маршрут разработчика: от настройки среды и запуска кода в эмуляторе QEMU и среде Docker до кросс-компиляции, линковки и прошивки микроконтроллера через openOCD. Отдельное внимание уделено работе с RARS, подготовке сборочных файлов, подключению библиотек mik32-hal и mik32v2-shared, а также особенностям сборки программ на ассемблере RISC-V под микроконтроллер. Все этапы сопровождаются практическими примерами, командами и пояснениями, готовыми к непосредственному использованию.

Практикум предназначен для программистов, инженеров и студентов, уже освоивших основы языка ассемблера RISC-V и желающих применить их при проектировании встраиваемых систем на отечественном микроконтроллере МІК32 Амур.

© А.Ю. Романов, М.Т. Омаров, 2026

Распространяется по лицензии CC BY 4.0

Об авторах



Романов Александр Юрьевич (https://www.hse.ru/staff/a.romanov) – автор русской адаптации данной книги, д. т. н., доцент, профессор Московского института электроники и математики им. А. Н. Тихонова Национального исследовательского университета «Высшая школа экономики» (МИЭМ НИУ ВШЭ). С 2014 г. работает в МИЭМ НИУ ВШЭ, где возглавляет лабораторию САПР (https://miem.hse.ru/edu/ce/cadsystem), специализирующуюся на проектной деятельности, а также разработке цифровых систем на ПЛИС/микроконтроллерах, робототехнических комплексов, аппаратных реализаций систем искусственного интеллекта, многопроцессорных систем, систем удаленного доступа к лабораторному оборудованию и т. д. Является автором более 200 научных статей.



Омаров Марат Тимурович – аспирант 1 г.о. МИЭМ НИУ ВШЭ. Выпускник и преподаватель МИЭМ НИУ ВШЭ. Работает над диссертацией по теме: «Исследование передачи трафика многопоточных приложений в сетях на кристалле с использованием высокоуровневого моделирования и симуляции процессорных ядер».

Авторы благодарны Зиёеву Рустаму и Мусиной Александре, студентам МИЭМ НИУ ВШЭ, за помощь в подготовке этой книги.

Содержание

ПРЕДИСЛОВИЕ **10**

ГЛАВА 1. НАЧАЛО РАБОТЫ **11**

ГЛАВА 1.1. КОМПИЛЯЦИЯ RISC-V КОДА **11**

Введение 11

Работа в QEMU 11

Подготовка среды в Ubuntu 11

Компиляция RISC-V кода 11

Работа с кросс-компилятором 12

Сборка программы под микроконтроллер MIK32 Амур 12

Использование утилиты RARS 13

ГЛАВА 1.2. РЕКОМЕНДАЦИИ ПО РАБОТЕ С QEMU И DOCKER НА UNIX-СИСТЕМАХ **14**

Введение 14

Установка QEMU 14

Установка Docker 14

Работа с QEMU 14

Создание виртуальной машины 14

Настройка ssh 15

Дополнительно 15

Работа с Docker 16

Настройка Docker 16

Подготовка образа 16

Запуск контейнера 17

ГЛАВА 1.3. КРОСС-КОМПИЛЯЦИЯ ПОД MIK32 АМУР **18**

Установка пакетов 18

Установка зависимых проектов 18

Установка набора утилит для сборки проектов под mik32 Амур 18

Установка библиотек периферийных блоков 18

Установка загрузчика 18

Инструкция для Windows 18

Инструкция для Linux 18

Установка компилятора 18

Установка openOCD 19

Инструкция для macOS 19

Установка компилятора 19

Установка openOCD 19

Сборочный файл проекта 20

Теоретические вопросы 22

ГЛАВА 2. ЗАГРУЗКА И СЛОЖЕНИЕ 24

Цель 24

Теоретическая часть 24

Архитектура и регистры 24

Представления чисел и дополнительный код 24

Форматы инструкций 24

Базовые инструкции 24

Задание для лабораторной работы 25

Теоретические вопросы 26

ГЛАВА 3. ИНСТРУМЕНТЫ РАЗРАБОТЧИКА 27

ГЛАВА 3.1. ИНСТРУМЕНТЫ РАЗРАБОТЧИКА: GNU MAKE 27

Цель 27

Теоретическая часть 27

Процесс сборки программы 27

Роль Make 27

Создание правил в Makefile 27

Возможности Makefile 28

Задание для лабораторной работы 31

Теоретические вопросы 31

ГЛАВА 3.2. ИНСТРУМЕНТЫ РАЗРАБОТЧИКА: GNU DEBUGGER 32

Цель 32

Теоретическая часть 32

Что такое GDB 32

Использование GDB 32

Задание для лабораторной работы 34

Введение 34

Работа в QEMU 37

Работа с MIK32 Амур 39

Теоретические вопросы 39

ГЛАВА 4. УПРАВЛЕНИЕ ПОТОКОМ ПРОГРАММ 41

Цель 41

Теоретическая часть 41

Безусловные переходы 41

Условные переходы 41

Циклы 41

Логические операции 41

Преобразование чисел в ASCII 41

Задание для лабораторной работы 41
Теоретические вопросы 42
ГЛАВА 5. РАБОТА С ПАМЯТЬЮ 43
Цель 43
Теоретическая часть 43
Задание для лабораторной работы 43
Теоретические вопросы 44
ГЛАВА 6. ФУНКЦИИ И СТЕК 45
Цель 45
Теоретическая часть 45
Стек и регистр sp 45
Вызов функций и регистр ra 45
Соглашения о вызовах 45
Стековые кадры 46
Макрофункции 46
Рекурсия 46
Преобразование чисел в строку 46
Задание для лабораторной работы 46
Теоретические вопросы 46
ГЛАВА 7. СЛУЖБЫ ОПЕРАЦИОННОЙ СИСТЕМЫ LINUX 48
Цель 48
Теоретическая часть 48
Задание для лабораторной работы 48
Теоретические вопросы 49
ГЛАВА 8. ПРОГРАММИРОВАНИЕ КОНТАКТОВ GPIO 50
Цель 50
Теоретическая часть 50
Задание для лабораторной работы 50
Теоретические вопросы 51
ГЛАВА 9. ВЗАИМОДЕЙСТВИЕ С ЯЗЫКАМИ C И PYTHON 52
Цель 52
Теоретическая часть 52
Особенности линковки C и ассемблера 52
Вызов функций C из ассемблера 52
Ассемблерные вставки в коде C 52
Разделяемые библиотеки и Python 53
Задание для лабораторной работы 53
Теоретические вопросы 53

ГЛАВА 10. УМНОЖЕНИЕ И ДЕЛЕНИЕ 54
Цель 54
Теоретическая часть 54
Инструкции умножения 54
Инструкции деления 54
Умножение матриц 54
Интеграция с языком C 55
Задание для лабораторной работы 55
Теоретические вопросы 55
ГЛАВА 11. ОПЕРАЦИИ С ЧИСЛАМИ С ПЛАВАЮЩЕЙ ТОЧКОЙ 56
Цель 56
Теоретическая часть 56
Задание для лабораторной работы 56
Теоретические вопросы 57
ГЛАВА 12. ОПТИМИЗАЦИЯ ИСХОДНОГО КОДА 58
Цель 58
Теоретическая часть 58
Задание для лабораторной работы 58
Теоретические вопросы 60
ГЛАВА 13. ЧТЕНИЕ И ПОНИМАНИЕ ИСХОДНОГО КОДА 61
Цель 61
Теоретическая часть 61
Зачем нужно понимать ассемблерный код 61
Компиляция высокоуровневого кода 61
Инструмент Ghidra 61
Задание для лабораторной работы 61
Теоретические вопросы 62
ГЛАВА 14. ВЗЛОМ ИСХОДНОГО КОДА 63
Цель 63
Теоретическая часть 63
Задание для лабораторной работы 64
Теоретические вопросы 64
ПРИЛОЖЕНИЕ 1 65
Лабораторная работа 1 65
Лабораторная работа 2 65
Пример возможной реализации программы 65
Оценивание лабораторной работы 67
Лабораторная работа 3.1 67

Пример возможной реализации программы 67
Оценивание лабораторной работы 67
Лабораторная работа 3.2 67
Пример возможной реализации программы 67
Оценивание лабораторной работы 67
Лабораторная работа 4 68
Пример возможной реализации программы 68
Оценивание лабораторной работы 70
Лабораторная работа 5 71
Пример возможной реализации программы 71
Оценивание лабораторной работы 75
Лабораторная работа 6 75
Пример возможной реализации программы 75
Оценивание лабораторной работы 78
Лабораторная работа 7 78
Пример возможной реализации программы 78
Оценивание лабораторной работы 81
Лабораторная работа 8 81
Пример возможной реализации программы 81
Оценивание лабораторной работы 86
Лабораторная работа 9 87
Пример возможной реализации программы 87
Оценивание лабораторной работы 90
Лабораторная работа 10 90
Пример возможной реализации программы 90
Оценивание лабораторной работы 93
Лабораторная работа 11 93
Пример возможной реализации программы 93
Оценивание лабораторной работы 95
Лабораторная работа 12 95
Пример возможной реализации программы 95
Оценивание лабораторной работы 97
Лабораторная работа 13 97
Пример возможной реализации программы 97
Оценивание лабораторной работы 98
Лабораторная работа 14 98
Пример возможной реализации программы 98
Оценивание лабораторной работы 101

ПРИЛОЖЕНИЕ 2 102

Ответы на вопросы к главе 1 102

Ответы на вопросы к главе 2 102

Ответы на вопросы к главе 3 103

Глава 3.1 103

Глава 3.2 104

Ответы на вопросы к главе 4 104

Ответы на вопросы к главе 5 105

Ответы на вопросы к главе 6 106

Ответы на вопросы к главе 7 107

Ответы на вопросы к главе 8 107

Ответы на вопросы к главе 9 108

Ответы на вопросы к главе 10 108

Ответы на вопросы к главе 11 109

Ответы на вопросы к главе 12 110

Ответы на вопросы к главе 13 112

Ответы на вопросы к главе 14 114

Предисловие

Когда Стивен Смит писал свою книгу «Программирование на языке ассемблера RISC-V», он ставил целью познакомить читателя с новой открытой процессорной архитектурой, которая стремительно набирает популярность во всем мире. Его книга, переведенная на русский язык и изданная в серии «Книжная полка Истового инженера», дает читателю прочный фундамент: архитектура набора инструкций, все группы команд ассемблера, приемы оптимизации кода, взаимодействие с C и Python.

Но любой фундамент требует практики. Настоящий практикум – это попытка перенести базу и теоретические знания, полученные из книги С. Смита, на реальную аппаратную платформу. Причем платформу не абстрактную, а отечественную – микроконтроллер МІК32 Амур производства АО «Микрон».

Почему именно МІК32 Амур? Потому что сегодня как никогда важно уметь работать с собственными аппаратными решениями. RISC-V – это открытая архитектура, и МІК32 Амур – один из первых отечественных микроконтроллеров, построенных на этой архитектуре. Освоив его, вы получаете не просто навык программирования конкретного чипа, а понимание того, как устроены современные встраиваемые системы.

Структура практикума повторяет структуру книги Смита. Глава за главой вы будете идти тем же маршрутом: от настройки среды и первых шагов до работы с периферией и оптимизации кода. Но если в исходной книге примеры были отвлеченными, то здесь каждый пример привязан к конкретному микроконтроллеру.

Вы научитесь:

- запускать код в эмуляторе QEMU и среде Docker, если под рукой нет реального железа;
- использовать RARS для быстрой отладки и проверки идей;
- настраивать кросс-компилятор и собирать программы на ассемблере под МІК32 Амур;
- подключать библиотеки mik32-hal и mik32v2-shared;
- прошивать микроконтроллер через openOCD.

Отдельно отмечу: этот практикум не заменяет книгу С. Смита. Он ее дополняет. Если вы еще не читали исходную книгу, начните с нее – иначе многие примеры могут оказаться вам не по зубам. Если же вы уже освоили базовую книгу, практикум станет вашим следующим шагом.

Книга рассчитана на программистов, инженеров и студентов, которые уже знакомы с языками высокого уровня и хотят освоить ассемблер RISC-V на реальном отечественном микроконтроллере. Для работы с практикумом достаточно базовых знаний C и представления о том, как устроен компьютер.

Все примеры в книге проверены на реальном железе. Авторы надеются, что этот практикум поможет вам не только освоить МІК32 Амур, но и полюбить ассемблер RISC-V так же, как любит его автор исходной книги.

В добрый путь!

Александр Юрьевич Романов,

д. т. н., доцент, профессор ДКИ МИЭМ НИУ ВШЭ,

преподаватель курсов «Проектирование систем на кристалле»,

«Системное проектирование цифровых устройств»

и «Системы искусственного интеллекта»

г. Москва, Россия

Глава 1. Начало работы

Глава 1.1. Компиляция RISC-V кода

Введение

Программы, разработанные на языке ассемблера RISC-V можно запускать разными способами:

- используя кросс-компилятор и компьютер с архитектурой RISC-V (например, микроконтроллер esp32 или mik32 Амур),
- эмулируя ОС на базе RISC-V процессора (QEMU),
- используя сторонние утилиты (RARS).

Ниже приведены рекомендации по работе с кодом RISC-V на разных платформах.

Работа в QEMU

QEMU позволяет эмулировать работу процессора отличного от процессора хост-системы. Благодаря этому можно эмулировать работу компьютера на процессоре RISC-V.

В книге С. Смита «Программирование на языке ассемблера RISC-V» в главе 1 «Начало работы» имеется инструкция по настройке и запуску Ubuntu RISC-V в QEMU в разделе «Программирование Hello World в эмуляторе QEMU». Кроме того, в качестве дополнения к этой информации были разработаны рекомендации по использованию QEMU и Docker для эмуляции работы Ubuntu RISC-V (см. раздел ниже).

Подготовка среды в Ubuntu

Если в качестве эмулированной среды был использован Docker-образ, собранный по Dockerfile из методических указаний, данный раздел можно пропустить.

При использовании QEMU напрямую или Docker для эмуляции компьютера ОС Ubuntu на RISC-V необходимо установить пакет **<build-essential>** для возможности компилировать RISC-V код:

Листинг 1 – Установка пакета build-essential в Ubuntu RISC-V

```
apt-get update && apt-get -y upgrade && \
apt-get -y install build-essential
```

Кроме набора инструментов компиляции для редактирования документов удобно использовать пакет **<vim>** или **<nano>**:

Листинг 2 – Установка пакетов vim и nano

```
apt-get -y install vim nano
```

Компиляция RISC-V кода

Для сборки подготовленного ассемблерного кода RISC-V нужно использовать следующую команду:

Листинг 3 – Сборка программы на ассемблере RISC-V

```
gcc -nostartfiles example.S -o example.elf
```

Флаг **<-nostartfiles>** необходимо использовать, если в программе определяется метка **<_start>**, иначе произойдет ошибка линковки с переопределением системной метки.

Работа с кросс-компилятором

Разработанную на языке ассемблера RISC-V программу можно собрать в исполняемый файл на компьютере, который не использует RISC-V процессор, если для него существует пакет кросс-компилятора RISC-V.

Для кросс-компиляции рекомендуется использовать пакет **<riscv-none-elf-gcc-xpack>**, инструкцию по установке которого можно найти на официальном сайте (**https://xpack-dev-tools.github.io/riscv-none-elf-gcc-xpack/docs/install/**).

При использовании кросс-компилятора программу на RISC-V ассемблере можно собрать следующей командой:

Листинг 4 – Сборка программы на ассемблере RISC-V кросс-компилятором

```
riscv-none-elf-gcc -nostartfiles example.S -o example.elf
```

Программа, собранная командой выше будет успешно запускаться на компьютере с процессором RISC-V, но для кросс-компиляции программ под микроконтроллер могут потребоваться дополнительные действия (например, включение в сборку специальных библиотек).

Сборка программы под микроконтроллер MIK32 Амур

Для сборки программ на языке ассемблера RISC-V и их успешного запуска на микроконтроллере MIK32 Амур кроме самого кросс-компилятора нужны специальные библиотеки, определяющие функционал для работы с периферийными блоками микроконтроллера, а также библиотеки, содержащие заголовочные файлы, стартовые скрипты и скрипты линковки.

Все описанные выше зависимости и их установка подробно описаны в разделе методических указаний по кросс-компиляции программ под MIK32 Амур.

Кроме того, для быстрой сборки программ под микроконтроллер были подготовлены правила сборки, поддерживающие:

- быструю установку пакетов кросс-компилятора и загрузчика **<openOCD>** в зависимости от архитектуры хост-системы;
- автоматическую установку библиотек и проектов со скриптами памяти микроконтроллера;
- корректное включение исходников библиотек, скриптов линковки и определение флагов компиляции;
- правила для быстрого программирования микроконтроллера.

Данные сборочные правила доступны на github: **https://github.com/RomeoMe5/riscv-assembly-programming/tree/master/lab_03/1_make/solution**.

Для использования проекта необходимо:

- установить проект;
- заменить код в папке src/ на свой код;

– выполнить команду **`make`** и следовать инструкциям при возникновении ошибок;
– выполнить команду **`make upload`** для программирования микроконтроллера.

Использование утилиты RARS

Для запуска и отладки кода на ассемблере RISC-V можно использовать утилиту RARS («RISC-V Assembler and Runtime Simulator»). Для этого нужно скачать последний доступный релиз (файл формата «**.jar**») по ссылке **https://github.com/TheThirdOne/rars/releases**.

Для запуска утилиты необходимо установить **<Java Runtime Environment>** для своей операционной системы **https://www.java.com/en/download/manual.jsp**.

После этого нужно запустить файл «**.jar**» двойным кликом и в качестве программы для запуска выбрать «Java (™) Platform SE binary».

Исполняемый код необходимо вставить в окно «Edit», компиляция выполняется с помощью нажатия «Run → Assemble» (рис. 1).

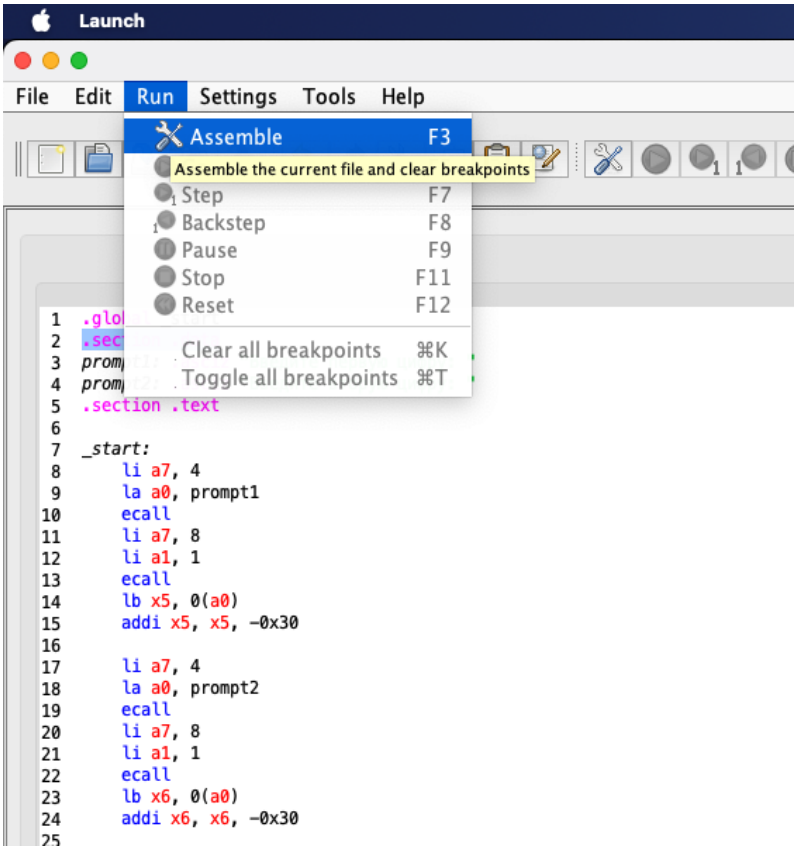


Рисунок 1 – Компиляция кода в RARS

Для запуска программы необходимо в окне «Execute» нажать на «Run» (рис. 2). После запуска программы состояния регистров отображаются в отдельном окне справа.

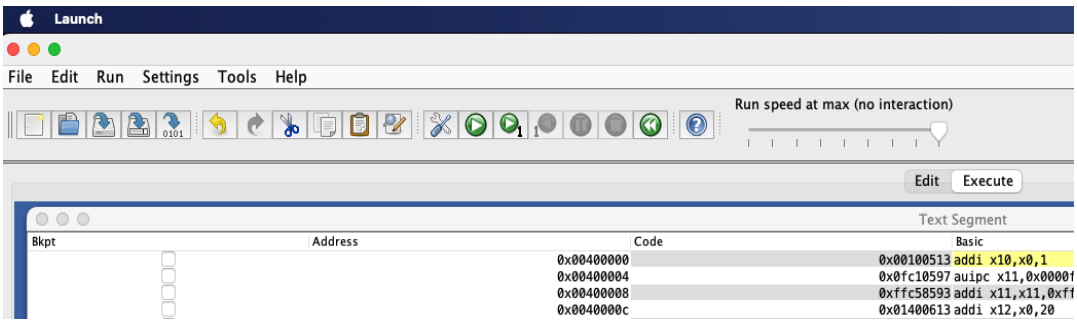


Рисунок 2 – Запуск программы в RARS.

Глава 1.2. Рекомендации по работе с QEMU и Docker на Unix-системах

Введение

Установка QEMU

Для установки QEMU на компьютер достаточно следовать инструкциям с официального сайта: https://www.qemu.org/download/#source.

Для установки пакета, как правило, достаточно воспользоваться пакетным менеджером используемой системы:

– для Debian/Ubuntu – apt:

Листинг 5 – Установка пакета QEMU и необходимых зависимостей на Ubuntu

```
sudo apt-get install -y qemu-system-riscv64 opensbi u-boot-qemu
```

– для macos – brew:

Листинг 6 – Установка пакета QEMU на macos

```
brew install qemu
```

Установка Docker

Для установки Docker необходимо воспользоваться документацией с официального сайта: https://docs.docker.com/engine/install/.

Работа с QEMU

Ниже приведены рекомендации по работе с QEMU, но можно руководствоваться и официальной документацией установки RISC-V Ubuntu в QEMU: https://canonical-ubuntu-boards.readthedocs-hosted.com/en/latest/how-to/qemu-riscv/.

Для создания виртуальной машины на базе RISC-V требуется скачать предустановленный образ Ubuntu Server (LTS): https://ubuntu.com/download/risc-v, а также следующие системные файлы в одну папку с образом сервера:

– fw_jump.elf,

– uboot.elf.

Создание виртуальной машины

Для удобства использования ранее скачанные файлы необходимо разместить в одной директории (избегайте использования кириллицы в пути директории).

Перед запуском QEMU, рекомендуется увеличить доступный для виртуальной машины (ВМ) размер хранилища:

Листинг 7 – Увеличение размера образа для ВМ

```
sudo qemu-img resize ubuntu-24.04.1-server-riscv64.img +10G
```

Далее рекомендуется создать bash-скрипт, который позволит упростить запуск ВМ (листинг 8).

Листинг 8 – Bash-скрипт запуска ВМ QEMU

```
#!/bin/bash

SCRIPT_DIR=$(cd -- "$(dirname -- "${BASH_SOURCE[0]}")" &>/dev/null && pwd)

qemu-system-riscv64 \
  -machine virt -nographic -m 2048 -smp 4 \
  -bios $SCRIPT_DIR/qemu-elfs/fw_jump.elf \
  -kernel $SCRIPT_DIR/qemu-elfs/uboot.elf \
  -drive file=$SCRIPT_DIR/ubuntu-24.04.1-server-riscv64.img,format=raw,if=virtio \
  -device virtio-net-device,netdev=net0 -netdev user,id=net0,hostfwd=tcp::2222-:22
\
  -append "root=LABEL=rootfs console=ttyS0" \
  -full-screen
```

Приведенные инструкции позволяют запустить виртуальную машину, к которой можно подключиться на локальном хосте на порту 2222.

Настройка ssh

Для подключения к запущенной ВМ можно использовать ssh, который необходимо установить в ВМ:

Листинг 9 – Установка и настройка ssh в ВМ

```
sudo apt update && sudo apt install -y openssh-server

sudo systemctl enable --now ssh

sudo systemctl status ssh
```

Последняя команда должна подтвердить успешный запуск ssh-сервера.

Теперь с хост-системы должно успешно выполняться подключение выпет вызова команд:

Листинг 10 – Подключение к ВМ по ssh

```
ssh -p 2222 localhost
```

Дополнительно

Для удобства работы с ВМ рекомендуется воспользоваться пакетом tmux, который позволит удобно создавать и управлять многими рабочими пространствами:

Листинг 11 – Установка tmux

```
sudo apt install -y tmux
```

Полезные материалы по использованию tmux:

- использование tmux,
- команды tmux.

При проблемах с корректным выводом консоли в ВМ, можно восстановить консоль путем выполнения следующих команд:

Листинг 12 – Восстановление работы консоли

```
stty sane && echo -e "\033c"
```

Работа с Docker

Альтернативой использованию QEMU в Unix-системах является Docker, поскольку поддерживает сборку и запуск контейнеров под разные архитектуры, используя QEMU. Для изучения возможностей Docker обратитесь к официальной документации. Ниже приведены основные команды для подготовки образа Ubuntu RISC-V.

Настройка Docker

Настройка Docker для возможности использования QEMU отличается для разных платформ: на macos (arm-чипы) все должно работать по умолчанию из-за особенностей архитектуры, при этом на Linux необходимо заранее выполнить следующую команду:

Листинг 13 – Настройка Docker на Linux

```
docker run --privileged --rm tonistiigi/binfmt --install all
```

Подготовка образа

Для создания готового для работы образа Ubuntu архитектуры riscv64 можно взять за основу образ riscv64/ubuntu версии 24.04 (или любой другой поддерживаемой версии). Ниже приведен Dockerfile, устанавливающий все требуемые пакеты и локальные настройки для поддержки вывода кириллицы:

Листинг 14 – Dockerfile для подготовки образа Ubuntu riscv64

```
FROM riscv64/ubuntu:noble

ENV DEBIAN_FRONTEND=noninteractive

RUN apt-get update && \
    apt-get -y upgrade && \
    apt-get -y install \
    tmux git neofetch \
     locales language-pack-ru \
    build-essential vim nano \
    qemu-user-static
```

```
ARG USR=root
WORKDIR /home/${USR}/

ARG LOCALE=ru_RU.UTF-8
ENV LANG=${LOCALE}
ENV LANGUAGE=${LOCALE}
ENV LCC_LANG=${LOCALE}
ENV LCC_ALL=${LOCALE}
RUN locale-gen ${LOCALE} && dpkg-reconfigure locales && \
    cd /home/${USR} && \
    echo -e "export LANG=${LOCALE}\nexport LANGUAGE=${LOCALE}" > .bash_profile && \
    echo -e "export LC_LANG=${LOCALE}\nexport LC_ALL=${LOCALE}" >> .bash_profile &&
\
    echo "source ~/.bash_profile" >> .bashrc

CMD ["/bin/bash"]
```

Предложенный выше код необходимо сохранить в файле с названием «Dockerfile», после чего, в консоли перейти в директорию с данным файлом и выполнить сборку образа:

Листинг 15 – Сборка образа

```
docker buildx build --platform linux/riscv64 -t riscv-ubuntu .
```

Запуск контейнера

После успешной сборки образа нужно создать контейнер на его основе:

Листинг 16 – Создание контейнера

```
docker run --privileged --name riscv -h riscv-ubuntu \
-v /tmp:/host -p 7777:7777 -w /home -it riscv-ubuntu
```

Эта команда создает контейнер с именем riscv и добавляет связь с временной директорией на хосте (директория /host в контейнере соответствует директории /tmp на хосте).

Для запуска контейнера после его завершения нужно выполнить следующие команды:

Листинг 17 – Запуск контейнера

```
docker start riscv && docker exec -it riscv /bin/bash
```

Для завершения работы контейнера нужно выполнить следующую команду:

Листинг 18 – Выключение контейнера

```
docker stop riscv
```

Глава 1.3. Кросс-компиляция под MIK32 Амур

Установка пакетов

Для компиляции программ под RISC-V архитектуру необходим кросс-компилятор. Также нужны исходные файлы официальных проектов Mikron **<mik32-hal>** (библиотеки для программирования периферийных блоков микроконтроллера), **<mik32v2-shared>** (набор для сборки проектов под микроконтроллер) и **<mik32-uploader>** (для прошивки микроконтроллера mik32 Амур через **openOCD**).

Кросс-компиляцию программы под mik32 Амур можно выполнять на разных платформах – как на macOS, так и на Windows.

Ниже приведены инструкции по установке нужных инструментов и по работе с ними на различных платформах.

Установка зависимых проектов

Установка набора утилит для сборки проектов под mik32 Амур

Листинг 19 – Установка набора утилит для сборки проектов

```
git clone \
https://github.com/MikronMIK32/mik32v2-shared.git
cd mik32v2-shared && git checkout tags/v0.1.3
```

Установка библиотек периферийных блоков

Листинг 20 – Установка библиотек периферийных блоков микроконтроллера

```
git clone --depth 1 https://github.com/MikronMIK32/mik32-hal.git
cd mik32-hal && git checkout tags/v0.1.3
```

Установка загрузчика

Загрузка прошивки на микроконтроллер осуществляется через **openOCD**. Проект **mik32-uploader** содержит вспомогательные скрипты для работы с памятью микроконтроллера.

Листинг 21 – Установка вспомогательных скриптов для загрузчика

```
git clone https://github.com/MikronMIK32/mik32-uploader.git
```

Кроме проекта также нужен сам **openOCD** (см. установку ниже).

Инструкция для Windows

Рекомендуется обратиться к документации на сайте wiki mik32.

Инструкция для Linux

Установка компилятора

Необходимо выполнить следующие инструкции: https://xpack-dev-tools.github.io/riscv-none-elf-gcc-xpack/

Достаточно установить актуальный релиз в виде архива и распаковать его по рекомендуемому пути (подробнее здесь).

Ниже приведены все команды для быстрой установки по путям, которые далее будут использоваться в правилах сборки проектов:

Листинг 22 – Установка компилятора под Linux x64

```
cd ~/Downloads && \
wget https://github.com/xpack-dev-tools/riscv-none-elf-gcc-
xpack/releases/download/v14.2.0-3/xpack-riscv-none-elf-gcc-14.2.0-3-linux-x64.tar.gz

mkdir -p ~/.local/xPacks/riscv-none-elf-gcc
cd ~/.local/xPacks/riscv-none-elf-gcc

tar xvf ~/Downloads/xpack-riscv-none-elf-gcc-14.2.0-3-linux-x64.tar.gz

chmod -R -w xpack-riscv-none-elf-gcc-14.2.0-3
```

Установка openOCD

Листинг 23 – Установка openOCD

```
cd ~/Downloads && \
wget     https://github.com/xpack-dev-tools/openocd-xpack/releases/download/v0.12.0-
6/xpack-openocd-0.12.0-6-linux-x64.tar.gz

mkdir -p ~/.local/xPacks/openocd
cd ~/.local/xPacks/openocd

tar xvf ~/Downloads/xpack-openocd-0.12.0-6-linux-x64.tar.gz
chmod -R -w xpack-openocd-0.12.0-6
```

Инструкция для macOS

Для стабильной работы пакета программ рекомендуется пользоваться xpack-пакетом (с постфиксом darwin), инструкция по установке которого описана в разделе «Инструкция для Linux». Для работы устройств на arm-чипе (m1, m2 и т.д.), необходимо использовать релиз с постфиксом darwin-arm.

Инструкция ниже является альтернативным путем и описывает установку другого пакета с помощью пакетного менеджера brew.

Установка компилятора

Достаточно использовать пакетный менеджер brew (инструкция по установке):

Листинг 24 – Установка компилятора на macos

```
brew tap riscv/riscv && brew install riscv-gnu-toolchain
```

При необходимости можно использовать инструкцию от сообщества RISC-V: https://github.com/riscv-software-src/homebrew-riscv.

Установка openOCD

Листинг 25 – Установка openOCD на macOS

```
brew install riscv-openocd
# возьмется из riscv/riscv/riscv-openocd
```

Сборочный файл проекта

Сборочный файл и пример проекта расположены по ссылке: https://github.com/RomeoMe5/riscv-assembly-programming/tree/master/lab_03/1_make/solution.

Ниже приведен исходный код сборки проекта под mik32 с пояснениями:

Листинг 26 – Исходный код правил сборки

```
# Пути проекта:
#       build – результаты сборки,
#       src – исходники проекта,
#       obj – объектные файлы (промежуточные результаты сборки)
BUILD = build
SRC   = src
OBJ   = obj
# Название проекта и выходного .hex файла
PROJECT_NAME ?= blink

# Пути к компилятору, линковщику и прочим инструментам (из папки
# кросс-компилятора)
CC = $(CROSS_PREFIX)gcc
LD = $(CROSS_PREFIX)ld
STRIP   = $(CROSS_PREFIX)strip
OBJCOPY = $(CROSS_PREFIX)objcopy
OBJDUMP = $(CROSS_PREFIX)objdump

# Режим работы микроконтроллера
# (зависит от положения перемычек CON2 на плате)
BOOT_MODE ?= eeprom
# Целевая архитектура
MARCH = rv32i_zicsr
MABI  = ilp32

# Пути к вспомогательным скриптам (скрипты памяти)
# Выбор LDSCRIPT зависит от режима работы микроконтроллера (BOOT_MODE)
LDSCRIPT = $(SHARED_DIR)/ldscripts/$(BOOT_MODE).ld
RUNTIME  = $(SHARED_DIR)/runtime/crt0.S

# “Подключение” заголовочных файлов (при использовании компилятор будет искать их по
этим путям)
INCLUDE += -I $(SHARED_DIR)/include \
                -I $(SHARED_DIR)/periphery \
                -I $(SHARED_DIR)/runtime \
                -I $(SHARED_DIR)/libs \
                -I $(HAL_DIR)/core/Include \
                -I $(HAL_DIR)/peripherals/Include \
                -I $(HAL_DIR)/utilities/Include

# Подключаемые библиотеки
LIBS    += -lc
```

```
# Флаги компиляции
CFLAGS  += -Os -MD -fstrict-volatile-bitfields -fno-strict-aliasing \
                -march=$(MARCH) -mabi=$(MABI) -fno-common -fno-builtin-printf

# Флаги линковки:
#       -T$(LDSCRIPT) – передача пути скрипта памяти линковщику
#       -L$(SHARED_DIR)/ldscripts – передача пути к папке со скриптами памяти #
(необходимо, поскольку $(BOOT_MODE).ld подключает другие файлы
#    из директории ldscripts)
LDFLAGS += -nostdlib -lgcc -mcmodel=medlow -nostartfiles -ffreestanding \
           -Wl,-Bstatic,-Map,$(OBJ)/$(PROJECT_NAME).map,\
           --print-memory-usage \
           -march=$(MARCH) -mabi=$(MABI) -specs=nano.specs -lnosys \
           -L$(SHARED_DIR)/ldscripts -T$(LDSCRIPT)

# Определение компилируемых файлов проекта:
#       - подключается все, что лежит в папке $(SRC),
#       - подключаются используемые зависимости из $(HAL_DIR)
#       и $(SHARED_DIR): это зависит от самого проекта
CORE_SOURCES := $(HAL_DIR)/peripherals/Source/mik32_hal.c \
                $(HAL_DIR)/peripherals/Source/mik32_hal_pcc.c \
                $(HAL_DIR)/peripherals/Source/mik32_hal_gpio.c \
                $(SHARED_DIR)/libs/uart_lib.c \
                $(RUNTIME)

SOURCES := $(CORE_SOURCES) $(wildcard $(SRC)/*.c $(SRC)/*.S)

OBJECTS := $(filter %.o,$(SOURCES:%.c=%.o) $(SOURCES:%.S=%.o))
OBJECTS += $(addprefix $(OBJ)/,$(OBJECTS))

# Выходной .hex файл
OUT = $(BUILD)/$(PROJECT_NAME).hex

# Ниже определены цели сборки (последовательность правил):
#       - all – главная цель, выполняется по умолчанию,
#       - после символа `:` перечисляются зависимые цели
#       (они будут выполнены в первую очередь),
#       - на следующей строке указываются сами команды сборки,
#       они выполняются тогда, когда все зависимости учтены.
all: $(OUT)
# Вызов all приведет к выполнению цели $(OUT)

# $(OUT) требует создания .elf файла (результата компиляции
# всех исходников проекта) и директории $(BUILD)
$(OUT): $(OBJ)/$(PROJECT_NAME).elf | $(BUILD)
        $(OBJCOPY) -O ihex $^ $@

# При этом elf-файл создается после создания всех объектных
# файлов. Иными словами, сначала нужно скомпилировать
# все исходники и получить все объектные файлы $(OBJECTS)
$(OBJ)/$(PROJECT_NAME).elf: $(OBJECTS)
        $(CC) $(CFLAGS) $(INCLUDE) -o $@ $^ $(LDFLAGS) $(LIBS)
```

```
# Каждый объектный (с расширением .o) будет собираться
# по следующему правилу:
$(OBJ)/%.o: %.c
        @mkdir -p $(dir $@)
        $(CC) -c -g $(CFLAGS) $(INCLUDE) -o $@ $^

# Для объектных файлов, имеющих расширение .S в исходниках
# создано свое отдельное правило
$(OBJ)/%.o: %.S
        @mkdir -p $(dir $@)
        $(CC) -c $(CFLAGS) $(INCLUDE) -o $@ $^

# Следующая цель создает директорию $(BUILD): $@ – короткий
# псевдоним для названия цели
$(BUILD):
        mkdir -p $@

# Цель upload не выполняется по умолчанию:
#       - она запускает прошивку микроконтроллера,
#       - чтобы запустить цель, нужно вызвать ее командой `make upload`
upload: $(OUT)
        python3 $(UPLOADER_DIR)/mik32_upload.py --run-openocd \
             --openocd-exec=$(OPENOCD) \
             --openocd-scripts $(UPLOADER_DIR)/openocd-scripts \
             --boot-mode $(BOOT_MODE) \
             --openocd-interface interface/ftdi/mikron-link.cfg $^

# Цель clean возвращает проект в исходное состояние
clean:
        rm -rf $(OBJ) $(BUILD)

# Директива .PHONY позволяет учесть особенность утилиты make:
#       - каждая цель для make – это файл,
#       - если файл с заданным именем уже существует и при этом
#         дата его создания не позже даты изменения его
#         зависимостей (тех самых, что стоят после `:`),
#         то данная цель будет пропущена.
#
# Это создает следующую проблему – цели all, upload, clean и deps
# не являются файлами, но если создать такие файлы в папке проекта,
# то цели перестанут выполняться и все перестанет работать.
# .PHONY заставляет make игнорировать файлы, совпадающие
# с названиями целей, чтобы решить вышеописанную проблему.
.PHONY: all upload clean deps
```

Теоретические вопросы

1. Зачем при сборке программы на ассемблере RISC-V используется флаг -nostartfiles?
2. Чем отличается кросс-компиляция от обычной компиляции и когда она необходима?
3. Почему для запуска RARS требуется установить Java Runtime Environment?
4. В чем преимущество использования QEMU для запуска RISC-V кода по сравнению с реальным микроконтроллером?

5. Зачем при работе с МІК32 Амур нужны библиотеки mik32-hal и mik32v2-shared, помимо самого кросс-компилятора?
6. Какую команду нужно выполнить для сборки программы example.S в файл example.elf с помощью кросс-компилятора?
7. Как увеличить размер образа виртуальной машины QEMU перед первым запуском?
8. Как подключиться к запущенной в QEMU виртуальной машине по SSH с хост-системы?
9. Какую команду используют для настройки Docker на Linux перед сборкой контейнеров под riscv64?
10. Какой целью в Makefile выполняется прошивка микроконтроллера МІК32 Амур?

Глава 2. Загрузка и сложение

Цель

Закрепить понимание архитектуры загрузки-хранения в RISC-V, представления чисел (включая отрицательные в дополнительном коде) и базовых арифметических операций. Научиться загружать константы в регистры, выполнять вычисления и выводить результаты в консоль Linux.

Теоретическая часть

Архитектура и регистры

Процессор RISC-V основан на архитектуре загрузки-хранения (load-store). Это означает, что арифметические операции (add, sub) могут выполняться только над данными, находящимися в регистрах. Чтобы работать с данными из памяти, их нужно сначала загрузить в регистры (load), а после вычислений – сохранить обратно (store).

Процессор имеет 31 регистр общего назначения (x1-x31) и нулевой регистр (x0), который всегда содержит 0. Также существуют ABI-имена (например, a0-a7 для аргументов, t0-t6 для временных значений), которые делают код читаемым.

Представления чисел и дополнительный код

Компьютеры используют двоичную систему. Для отрицательных чисел применяется дополнительный код (two’s complement). Для того, чтобы получить отрицательное число, необходимо инвертировать все биты положительного числа и прибавить к результату 1. (Например: -1 в 8-битном представлении – это 1111 1111 или 0xFF). Главное преимущество подхода: сложение знаковых и беззнаковых чисел выполняется аппаратно одинаково, переполнение отбрасывается без дополнительных операций.

Форматы инструкций

Каждая инструкция RISC-V занимает ровно 32 бита. Во 2 главе рассматривались три базовых формата:

- R-тип (Register): операции только с регистрами (например, add rd, rs1, rs2);
- I-тип (Immediate): операции с регистром и 12-битной константой (например, addi rd, rs1, imm);
- U-тип (Upper): загрузка 20-битной константы в старшие биты регистра (например, lui rd, imm).

Базовые инструкции

Основные инструкции в ассемблере RISC-V можно разделить по их назначению:

- арифметика: add (сложение), sub (вычитание), addi (сложение с константой); для вычитания используется addi с отрицательным числом);
- загрузка констант:
  - li rd, imm – псевдоинструкция для загрузки любого числа (ассемблер сам подставит нужные lui/addi);

- lui rd, imm – загружает 20-битное число в старшие биты (младшие 12 обнуляются).
- сдвиги: slli (логический влево), srli (логический вправо), srai (арифметический вправо); сдвиг влево на N бит эквивалентен умножению на 2^N;
- перемещение: mv rd, rs – псевдоинструкция, копирующая регистр (транслируется в addi rd, rs, 0).

Задание для лабораторной работы

Разработайте программу, которая выполняет следующие действия:

1. Загрузка и вычисление

В секции .data объявлены две переменные x и y.

1. Загрузите значения x и y в регистры.
2. Вычислите выражение: (x + y) * 2 + 4.
3. Сохраните итоговое значение обратно в память.

Примечание: Подберите значения x и y так, чтобы итоговый результат был двузначным числом (например, x=10, y=20 с результатом 64), чтобы упростить вывод.

2. Вывод результата в консоль

Отладку программы обычно значительно проще выполнять, используя вывод арифметических операций в консоль. Результат работы программы можно проверять непосредственно в регистрах, но необходимо освоить и данный способ вывода информации. Для вывода числа в консоль необходимо разделить его на разряды и перевести их в формат ascii.

Справка по системным вызовам Linux (RISC-V ABI):

- a7 = 64 (sys_write): вывод на экран; a0 – дескриптор (1 для stdout), a1 – адрес строки, a2 – длина;
- a7 = 93 (sys_exit): завершение программы; a0 – код возврата.

После выполнения работы ответьте на теоретические вопросы для закрепления материалов главы.

```
    # Пример: вывод числа в консоль
    .data
msg:    .ascii  "Result: "    # префикс для более аккуратного вывода
num:    .word   64            # пример числа
buf:    .space  2             # буфер под две ASCII-цифры
nl:     .ascii  "\n"          # перевод строки

    .text
    .globl _start
    .align 2
_start:
    # -- загрузка числа
    la    t0, num
    lw    t0, 0(t0)           # t0 ← 64

    # -- разбиение числа на разряды
    li    t1, 10
```

```
    div   t2, t0, t1        # t2 = 64 / 10 = 6
    rem   t3, t0, t1        # t3 = 64 % 10 = 4

    # -- преобразование в ASCII
    addi  t2, t2, '0'       # '6'
    addi  t3, t3, '0'       # '4'

    # -- посимвольная запись в буфер buf[0..1]
    la    t4, buf
    sb    t2, 0(t4)         # buf[0] = '6'
    sb    t3, 1(t4)         # buf[1] = '4'

    # -- вывод "Result: "
    li    a0, 1             # stdout
    la    a1, msg
    li    a2, 8             # длина строки "Result: " составляет 8 символов
    li    a7, 64            # sys_write
    ecall

    # -- вывод всех элементов буфера
    li    a0, 1
    mv    a1, t4            # адрес buf
    li    a2, 2
    li    a7, 64
    ecall

    # -- перевод строки
    li    a0, 1
    la    a1, nl
    li    a2, 1
    li    a7, 64
    ecall

    # -- завершение
    li    a0, 0
    li    a7, 93            # sys_exit
    ecall
```

Теоретические вопросы

11. Почему в RISC-V для доступа к данным из памяти всегда используют пару la + lw, а не одну инструкцию?
12. Как в RISC-V представляются отрицательные числа и почему при загрузке знакового 32-бита важно знать о двоичном дополнительном коде (two’s complement)?
13. Как умножить число на 2 с помощью логического сдвига?
14. В чем заключаются различия add и addi?
15. Объясните разницу между slli, srai и srli.
16. Зачем нужна mv?

Глава 3. Инструменты разработчика

Данная глава логически разделена на две части, предсталвяющие собой две отдельные работы по работе с инструментами GNU Make и GNU Debugger.

Глава 3.1. Инструменты разработчика: GNU Make

Цель

Закрепить навыки автоматизации компоновки программ с использованием GNU Make. Разработать Makefile для сборки проекта под микроконтроллер mik32 Амур.

Теоретическая часть

Процесс сборки программы

Сборка программы – это процесс преобразования исходного кода в исполняемый файл. Он состоит из нескольких этапов:

1. **Препроцессинг** – обработка директив препроцессора (такие команды, как: **#include**, **#pragma**, **#define** и пр.). Часто используемая директива **#include** еще до этапа компиляции просто копирует содержимое файла в целевой документ.
2. **Компиляция** – сложный процесс перевода высокоуровневого кода в машинный код. Каждый исходный файл компилируется независимо от других и представляет собой единицу компиляции. Потому для каждого исходного файла формируется свой объектный файл.
3. **Линковка** – объединение объектных файлов и используемых библиотек в исполняемый файл. Независимо скомпилированные объектные файлы на этом этапе собираются в единую программу.

При ручной сборке программ и проектов необходимо самостоятельно перечислять правила компиляции исходных файлов, передавать каждый из них компилятору, а также указывать множество других настроек (например, где компилятору искать библиотеки и заголовочные файлы).

Роль Make

GNU Make автоматизирует описанный выше процесс сборки благодаря правилам, которые позволяют:

- определить последовательности действий по созданию одних файлов из других (например, как создать объектные файлы из исходных и как получить исполняемый файл из объектных),
- упростить процесс сборки растущего проекта с помощью универсальных символов (**wildcard**-функции), переменных и функций.

Утилита GNU Make также позволяет ускорить процесс пересборки проектов благодаря инкрементной сборке, т.к. make пересоздает объектные файлы только для изменившихся файлов. Это осуществляется через сравнение даты последнего изменения исходных файлов с датой последнего изменения объектного файла.

Для работы с Make полезно изучить официальную документацию. Ниже приведено сжатое описание работы с Make.

Создание правил в Makefile

Правила в Make состоят из:

- цели – что будет получено в результате (обычно это путь к файлу);
- зависимостей – что необходимо для выполнения правила (требуют выполнения других правил);
- команд – что необходимо выполнить для получения результата.

Ниже приведен пример двух простых правил в Makefile:

Листинг 27 – Пример простых правил в Makefile

```
example_project_exe: obj/main.o obj/example.o obj/utils.o
        gcc $^ -o $@
obj/%.o: src/%.c
        gcc -c $< -o $@
```

В листинге описаны два правила: создание исполняемого файла **example_project_exe** и создание каждого объектного файла из исходников.

Первое правило определяет зависимости – объектные файлы с расширением «**.o**» в директории «**obj/**». Утилита определяет, что каждая из указанных зависимостей может быть создана вторым правилом.

Второе правило говорит утилите, что для создания объектного файла в «**obj/**» с расширением «**.o**» необходимо, чтобы документ в «**src/**» был с таким же названием, но с расширением «**.c**».

Возможности Makefile

Символы «$@», «$<», «$^» в правилах

Для того, чтобы не было дублирования кода, предусмотрены специальные переменные:

- **$@** – значение цели правила;
- **$^** – все зависимости правила;
- **$<** – значение первой зависимости в правиле.

Замена суффиксов (Substitution References)

В Make есть специальная конструкция вида «**$(<var>:a=b)**», позволяющая из одних переменных быстро получать другие. Данная конструкция заменяет последний символ **<a>** на **<b>** в каждом слове переданной переменной **<var>**.

Эту конструкцию удобно использовать для простых случаев замены. Например, так можно быстро сформировать из исходных файлов объектные:

Листинг 28 – Пример замены суффиксов в Makefile

```
SRC = main.c utils.c helper.c
OBJ = $(SRC:.c=.o)
# Результат: main.o utils.o helper.o
```

Функция wildcard и специальные символы «*», «?», «%»

Специальные символы позволяют описать правила независимо от имен документов (например, опираясь на их принадлежность к директории и/или на их расширения). Такие символы работаю так же, как и в командной оболочке **shell**.

Для определения переменных с использованием специальных символов необходимо использовать функцию **wildcard**. С помощью **wildcard** можно, например, быстро находить все исходные коды программы:

Листинг 29 – Пример использования wildcard в Makefile

```
SOURCES := $(wildcard $(SRCDIR)/*.c)
```

Важно помнить, что **wildcard** не производит поиск рекурсивно (документы с расширением «**.c**», расположенные в подпапках внутри **$(SRCDIR)/**) не будут обнаружены. Кроме того, при отсутствии совпадений с шаблоном **wildcard** вернет пустую строку.

Функция patsubst

Данная функция позволяет производить более сложные замены с помощью следующей конструкции: **$(patsubst <pattern>,<replacement>,<text>)**. Слова в переданном тексте **<text>**, соответствующие шаблону **<pattern>**, будут заменены на значения **<replacement>**. При несовпадении аргумента с шаблоном значение аргумента (слова в **<text>**) остается без изменений.

В описанных конструкциях часто используют специальные символы «**%**», позволяющие обозначить повторяющиеся подстроки.

Пример:

Листинг 30 – Пример использования patsubst в Makefile

```
# Простая замена расширений
OBJ = $(patsubst %.c,%.o,$(SRC))

# Смена пути директории
OBJ = $(patsubst src/%.c,obj/%.o,$(SRC))
```

В данном примере с помощью символа «**%**» замена корректно обобщается на все слова в переменной **$(SRC)**.

Функции filter и filter-out

Функция **filter** оставляет только те элементы в переданном аргументе, которые соответствуют указанному шаблону, тогда как **filter-out** удаляет элементы, которые соответствуют шаблону. Синтаксис у команд схожий: **$(filter <pattern>,<text>)** и **$(filter-out <pattern>,<text>)**.

Ниже приведен пример использования этих команд:

Листинг 31 – Пример использования filter-out в Makefile

```
# Создание переменной для сборки тестов программы
TEST_SOURCES := $(wildcard $(TESTDIR)/*.cpp)

# Создание объектных файлов
TEST_OBJECTS := $(TEST_SOURCES:$(TESTDIR)/%.cpp=$(OBJDIR)/%.o)

# Добавление объектных файлов стандартной сборки, но с удалением
# main.o
```

```
TEST_OBJECTS += $(filter-out $(OBJDIR)/main.o,$(OBJECTS))
```

Другие полезные функции

- **$(subst <from>,<to>,<text>)** – заменяет все вхождения **<from>** на **<to>** в **<text>**;
- **$(dir <names>)** – извлекает директории из путей к файлам;
- **$(notdir <names>)** – извлекает имена файлов из путей к файлам;
- **$(suffix <names>)** – извлекает расширения файлов;
- **$(basename <names>)** – получает имена файлов без расширений;
- **$(addsuffix <suffix>,<names>)** – добавляет суффиксы к каждому слову в **<names>**;
- **$(addprefix <path>,<names>)** – добавляет префиксы (например, путь) к каждому слову в **<names>**;
- и прочие.

Пример «универсального» Makefile для С-проектов

При подготовке Makefile для больших проектов полезно разделять документ на несколько частей. Например, можно вынести определение всех констант и переменных в отдельный документ «Makefile.conf», а в стандартном «Makefile» описать правила работы с этими переменными.

Ниже приведен пример Makefile для небольшого проекта на языке C++.

Листинг 32 – Определение Makefile

```
CXX = g++
CXXFLAGS = -O2 -Wall -Werror -std=c++17
INCLUDE = -I include
SRCDIR = src
OBJDIR = obj
SOURCES := $(wildcard $(SRCDIR)/*.cpp)
OBJECTS := $(SOURCES:$(SRCDIR)/%.cpp=$(OBJDIR)/%.o)

# pwd from Makefile-script directory and extract directory name
OUT := $(shell basename "$(shell cd -- "$(dirname -- "${BASH_SOURCE[0]}" )" &>
/dev/null && pwd)")

all: $(OUT)

$(OUT): $(OBJECTS)
        $(CXX) $^ $(CXXFLAGS) $(INCLUDE) -o $@

$(OBJDIR)/%.o: $(SRCDIR)/%.cpp | $(OBJDIR)
        $(CXX) -c $< $(CXXFLAGS) $(INCLUDE) -o $@

$(OBJDIR):
        mkdir -p $@
```

```
clean:
	rm -r $(OUT) $(OBJDIR)

.PHONY: all clean
```

Задание для лабораторной работы

Необходимо добавить ключевые элементы Makefile, собирающего простую программу под микроконтроллер MIK32 Амур. Для кросс-компиляции проектов под микроконтроллер потребуются:

- кросс-компилятор xpack;
- openocd xpack;
- набор для сборки приложений под микроконтроллер: https://github.com/MikronMIK32/mik32v2-shared/releases/tag/v0.1.3;
- библиотеки для программирования периферийных блоков микроконтроллера: https://github.com/MikronMIK32/mik32-hal/releases/tag/v0.1.3;
- скрипты для программирования памяти микроконтроллера: https://github.com/MikronMIK32/mik32-uploader/releases/tag/v0.3.3.

Большая часть логики в Makefile подготовлена и содержит предустановку вышеописанных зависимостей. В задании намеренно пропущена часть определений переменных исходных файлов и объектных файлов, а также опущена часть правил.

Для выполнения работы потребуется скачать проект (https://github.com/RomeoMe5/riscv-assembly-programming.git) с помощью git (обратите внимание на необходимость использования git-lfs для корректной работы с проектом) или путем скачивания архива. После установки проекта нужно перейти в директорию riscv-assembly-programming/lab_03/1_make/task, выполнить команду `**make**` и следовать инструкциям для установки набора инструментов и зависимостей.

Корректно составленные Makefile.conf и Makefile должны при вызове `**make**` собирать программу по пути `**build/blink.hex**`, которая должна успешно выполняться на микроконтроллере и демонстрировать мигания встроенных светодиодов.

Теоретические вопросы

1. Что такое Makefile и для чего он используется?
2. Из каких частей состоит правило в Makefile? Приведите пример.
3. Что делает этот код?

```
SOURCES = $(wildcard src/*.c)
OBJECTS = $(patsubst %.c,%.o,$(SOURCES))
```

4. Что означают автоматические переменные «**$@**», «**$<**» и «**$^**»? Приведите пример их использования.
5. Как проверить, существует ли файл в Makefile?
6. Что делает это правило?

```
%.o: %.c
```

```
	$(CC) -c $< -o $@
```

Глава 3.2. Инструменты разработчика: GNU Debugger

Цель

Закрепить навыки отладки программ с GNU Debugger. Найти и исправить ошибки в программе с использованием GDB. Научиться отлаживать программы на микроконтроллере MIK32 Амур.

Теоретическая часть

Что такое GDB

GNU Debugger (GDB) – это специальная программа для Unix-подобных систем с открытым исходным кодом, которая позволяет контролировать и изучать процесс выполнения программ, разработанных как на компилируемых языках высокого уровня (например, C, C++, Rust, Go и т.д.), так и на языке ассемблера.

GDB позволяет разработчикам анализировать работу программ, находить ошибки, а также изучать процесс выполнения разработанного кода на низком уровне. Сценарии использования утилиты можно разделить на несколько основных категорий:

- отладка ошибок;
- анализ программы: пошаговое выполнение функций, строк кода или даже ассемблерных инструкций для изучения работы потока программы;
- анализ памяти и регистров: просмотр состояния памяти, переменных и регистров в процессе выполнения программы;
- поиск уязвимостей.

GDB работает, подключаясь к запущенному процессу программы и контролируя ее выполнение. В частности, утилита позволяет:

- останавливать выполнение программы;
- пошагово проходить по линиям кода или инструкциям;
- просматривать и менять состояния памяти и регистров.

- Кроме того, у утилиты есть несколько режимов использования:

- локальная отладка: использование утилиты на том же компьютере, где запущена программа;
- удаленная отладка: отладка программы, запущенной на другом компьютере;
- отладка программы после ее завершения (Post-Modern Debugging): сохранение пространства памяти процесса и структур данных (снимок процесса) для изучения проблемы в работе после завершения программы.

Использование GDB

1. Запуск GDB с заданной программой

Отладчик можно запустить, передав в команде в качестве аргумента путь к исполняемому файлу. В таком случае утилита запустит программу и начнет контролировать ее исполнение:

Листинг 33 – Запуск GDB с программой

```
gdb <путь к программе>
```

2. Запуск GDB для удаленной отладки

Если нужно отладить программу, запущенную на другом компьютере, потребуется на этом удаленном компьютере запустить сервер GDB (gdbserver) на конкретном порту, после чего на локальной машине запустить GDB и подключиться к порту удаленного компьютера:

Листинг 34 – Удаленная отладка с помощью GDB

```
# На удаленном компьютере
apt -y install gdbserver && gdbserver :<порт> <путь к программе>
# или, при необходимости отлаживать уже работающую программу:
gdbserver --attach :<порт> <PID запущенной программы>


# На локальном компьютере
gdb <путь к программе> # необходим доступ к отладочной
                       # информации кода
target remote <IP-адрес удаленного компьютера>:<порт>
```

3. Запуск GDB для отладки с QEMU

Если нужно запустить программу в эмуляторе QEMU нужно использовать gdb сервер, предоставляемый QEMU. Кроме того, на локальной машине потребуется разместить собранную программу для получения отладочной информации, а также специальный экземпляр gdb, собранный под целевую архитектуру программы:

Листинг 35 – Удаленная отладка с QEMU

```
# В QEMU
qemu-<arch> -g <порт> <путь к программе>


# На локальном компьютере
<путь к кросс-gdb> <путь к программе>
target remote :<порт QEMU>
```

4. Основные команды

В GDB доступно множество команд, основная часть которых приведена в таблице 1. Для корректной работы GDB программу необходимо собирать с сохранением отладочной информации (для gcc должен быть установлен флаг «**-g**»).

Таблица 1 – Основные команды GDB

Команда	Описание
run [аргументы]	Запуск программы
break <функция / номер строки>	Установка точки останова на конкретной

	функции или строке кода
next	Выполнение следующей строки кода (пропустить содержимое функции)
step	Выполнение следующей строки кода (войти внутрь функции)
continue	Продолжить исполнение программы до следующей точки останова
print <переменная>	Вывести значение переменной
backtrace	Вывести стек вызовов
info registers	Вывести значения регистров процессора
x/<формат> <адрес>	Вывести данные с заданного адреса (например, команда **`x/4 $sp`** показывает 4 слова в hex-формате с вершины стека)
disassemble	Показать ассемблерный код текущей функции
list	Показать исходный код
quit	Выйти из GDB

Задание для лабораторной работы

Введение

Частая ошибка в языке C при работе со строками – потерянный терминирующий ноль. В этом языке все строки должны оканчиваться специальным символом, который позволяет определять конец данных. Данный символ записывается как число – «**0**», а в самих строках может быть представлен через экранирующий символ «****»: «**\0**» (например, строка «**Hello World**» в языке C будет представлена как «**Hello World\0**»). При потерянном терминирующем ноле стандартные функции в языке (например, «**strcpy**» или «**strlen**») будут итерироваться по данным некорректно, затрагивая адреса памяти, не принадлежащие строке, до тех пор, пока не найдут ноль или процесс не будет принудительно завершен.

Ниже приведен код на языке ассемблера RISC-V (листинг 36), воссоздающий простой пример с неправильно реализованной функцией копирования строк. Выполнение данной программы демонстрирует последствия простой ошибки, часто приводящей к замусориванию памяти, попытке перейти на недоступную для процесса память и незапланированному прекращению выполнения программы («падению»).

Листинг 36 – Программа с ошибкой на языке ассемблера RISC-V

```
.global main

.section .text

main:
    # Вывести оригинальную строку
    la a0, src_string
    jal ra, print_string

    # Скопировать строку
    la a0, src_string
    la a1, dest_buffer
    jal ra, buggy_strcpy

    # Вывести скопированную строку
    la a0, dest_buffer
    jal ra, print_string

    # Выйти из программы
    li a0, 0
    li a7, 93
    ecall

# Функция копирования строки с ошибкой
buggy_strcpy:
    mv t0, a0               # Указатель на источник (src)
    mv t1, a1               # Указатель на результат (dest)

copy_loop:
    lb t2, 0(t0)            # Загрузить байт из src
    beqz t2, copy_done      # Закончить на терминирующем нуле
    sb t2, 0(t1)            # Сохранить байт в dest
    addi t0, t0, 1          # Увеличить указатель src
    addi t1, t1, 1          # Увеличить указатель dest
    j copy_loop             # Повторить

copy_done:
    ret                     # Вернуться к месту вызова
```

```
# Вывести строку (читает до терминирующего нуля)
print_string:
    # Вычислить длину строки через поиск терминирующего нуля
    mv t0, a0               # Адрес строки
    li t1, 0                # Счетчик длины

length_loop:
    lb t2, 0(t0)
    beqz t2, print_it       # Остановиться на терминирующем нуле
    addi t0, t0, 1
    addi t1, t1, 1
    j length_loop

print_it:
    # Выполнить системный вызов записи
    li a0, 1                # вывод в stdout
    mv a1, t0               # текущая позиция
    sub a1, t0, t1          # вычисление адреса начала
    mv a2, t1               # длина
    li a7, 64               # запись номера системного вызова
    ecall

    # Вывод переноса строки («\n»)
    li a0, 1
    la a1, newline
    li a2, 1
    li a7, 64
    ecall

    ret

.section .data
# Переменные, записанные в памяти последовательно
src_string:   .asciz "Hello RISC-V"      # Оригинальная строка
dest_buffer:  .space 12                  # Память для результата
                                         # (конкретный размер)
new_string:   .asciz "...Copy Failed"    # Данные сразу после
                                         # результата
newline:      .ascii "\n"
```

Работа в QEMU

1. Подготовка среды

Для выполнения работы в эмуляторе QEMU необходимо подготовить среду выполнения. Рекомендуется использовать методические указания для подготовки среды в Docker-контейнере (для стабильности и быстрой работы): https://github.com/RomeoMe5/riscv-assembly-programming/blob/599814b2f515c3a539745f49afe95bd07e2853e7/manuals/manual_book_RISC-V_programming_2025_QEMU_and_Docker.pdf

2. Компиляция программы и запуск GDB

После настройки среды, в Ubuntu RISC-V, запущенном с помощью QEMU и содержащем пакеты «**build-essential**», «**qemu-user-static**», необходимо выполнить следующие действия:

Листинг 37 – Подготовка среды в QEMU

```
# Компиляция программы
gcc -g ch_3-gdb.S -o ch_3-gdb.out
# Запуск программы
./ch_3-gdb.out
# Запуск GDB сервера на порту 7777 (должен быть доступен
# на хост-компьютере)
qemu-riscv64-static -g 7777 ./ch_3-gdb.out
```

Далее нужно подключиться к запущенному в QEMU серверу с хост-компьютера. Перед этим необходимо скопировать собранную программу «**ch_3-out**». После копирования на хост-компьютере нужно выполнить действия, приведенные в листинге 38.

Листинг 38 – Подключение к GDB серверу в QEMU с хост-компьютера

```
# Запуск кросс-GDB в режиме tui (отображение кода)
<путь к cross-gdb «riscv-none-elf-gdb»> --tui <путь к программе ch_3-out>
# Подключение к серверу
target remote localhost:7777
break main
continue
```

После выполнения всех описанных шагов должно открыться следующее окно (рис. 3):

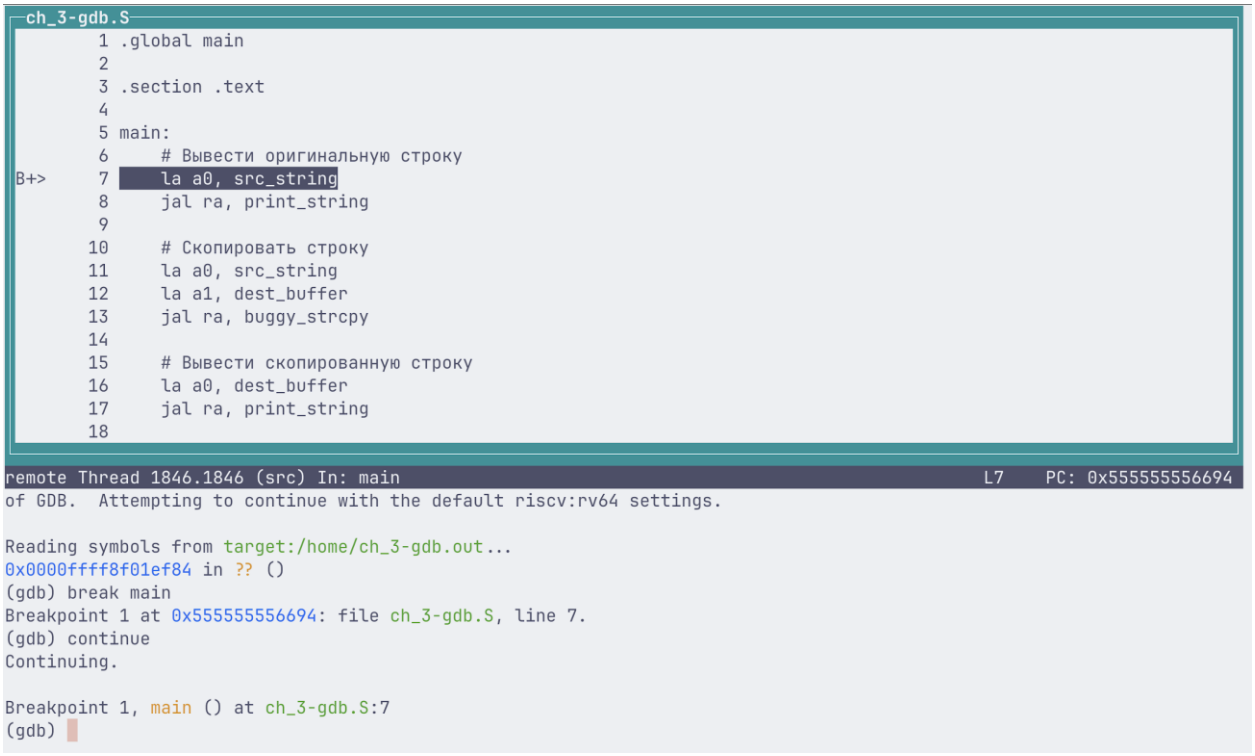


Рисунок 3 – Запуск GDB на исполняемой в QEMU программе

3. Отладка программы

После запуска отладчика выполните следующие действия:

1. Выведите значение переменной src_string: `**x/s &src_string**`.
2. Выведите 12 байт, содержащиеся в dest_buffer и убедитесь, что они равны нулю: `**x/12xb &dest_buffer**`.
3. Установите точку останова на метке buggy_strcpy.
4. Возобновите выполнение программы до остановки на метке buggy_strcpy.
5. Шагайте по инструкциям (stepi) до выполнения команды изменения памяти dest_buffer (`**sb t2, 0(t1)**`): `**stepi**`.
6. Проверьте состояние переменной dest_buffer в виде набора байт и в виде строки.
7. Установите точку останова на инструкцию `**sb t2, 0(t1)**` для ускорения проверки состояния переменной dest_buffer после каждой итерации.
8. После завершения выполнения функции копирования выведите состояние переменной dest_buffer и опишите выводы, сделанные вами.
9. Изучите процесс выполнения функций вывода скопированной переменной print_string и подсчета длины строки length_loop по аналогии с предыдущими шагами.
10. Сделайте выводы по работе length_loop и проанализируйте ее результат: зафиксируйте значение счетчика длины строки (регистр t1) перед выполнением системного вызова записи результата.

Задание

1. Продемонстрируйте выполнение шагов по запуску и отладке кода в QEMU (зафиксируйте в отчете выполненные шаги).
2. Опишите, в чем заключается ошибка программы и как ее исправить.
3. Предоставьте исправленную программу и зафиксируйте в отчете ее выполнение.
4. Ответьте на вопрос: какое значение получилось в счетчике длины скопированной строки? Почему? Какое значение должно было получиться?
5. Ответьте на теоретические вопросы.

Работа с MIK32 Амур

1. Компиляция программы

Выполните установку проекта по ссылке: **https://github.com/RomeoMe5/riscv-assembly-programming/tree/master/lab_03/2_gdb/solution**

Выполните сборку программы с помощью следующих команд:

Листинг 39 – Сборка программы под микроконтроллер

```
cd <путь к папке со скриптами>
make install-toolchain install-openocd install-deps all
make upload
```

2. Запуск GDB

После успешной прошивки микроконтроллера потребуется запустить сервер openOCD, выполняющий подключение к микроконтроллеру и позволяющий подключиться к процессу исполняемой программы. В конце потребуется запустить GDB и подключиться к удаленной цели на порт, который прослушивает запущенный openOCD.

Для выполнения описанных команд в правилах сборки Makefile уже подготовлены специальные команды. Вам потребуется выполнить действия, приведенные в листинге:

Листинг 40 – Запуск openOCD сервера и GDB для подключения к MIK32 Амур

```
make start-server
break main
continue
```

Дополнительное задание

Продемонстрируйте компиляцию кода и выполнение программы на микроконтроллере MIK32 Амур. Зафиксируйте в отчете работу с отладчиком GDB на микроконтроллере.

Теоретические вопросы

1. Для чего может использоваться команда `**break**` в GDB?
   a) Она останавливает программу, когда меняются значения переменных.
   b) Она устанавливает точку останова на функцию, строку кода или адрес.
   c) Она завершает сессию GDB.
   d) Она пропускает следующую инструкцию.

2. Какая из приведенных команд позволяет выполнить следующую инструкцию без вхождения в функцию?
   a) `**step**`.
   b) `**next**`.
   c) `**continue**`.
   d) `**finish**`.
3. Как вывести значение переменной `**x**` в GDB?
   a) `**display x**`.
   b) `**watch x**`.
   c) `**print x**`.
   d) `**info x**`.
4. Что делает команда `**backtrace**`?
5. Что делает следующий набор команд?

```
target remote :1234
break *0x400256
continue
stepi
```

Глава 4. Управление потоком программ

Цель

Изучить и закрепить базовые принципы управления порядком выполнения команд в языке ассемблера RISC-V: безусловные и условные переходы, реализацию циклов и конструкций if/then/else, а также освоить алгоритм преобразования целых чисел в ASCII-строку для вывода в консоль.

Теоретическая часть

В рамках этой темы рассматриваются средства реализации ветвлений (условных и безусловных переходов), построения циклов, применения логических операций, а также основы преобразования чисел для вывода в формате ASCII. Эти инструменты являются фундаментом для построения программ с динамическим поведением, реагирующих на данные, вводимые пользователем.

Безусловные переходы

Безусловный переход – это инструкция, которая передает управление другой части программы независимо от условий. В RISC-V используются инструкции j (переход по метке) и jal (переход с сохранением адреса возврата в регистр для вызова функций rd). Такая конструкция применяется, например, при завершении программы или возврате в начало цикла.

Условные переходы

Для организации логики «если... то...» в RISC-V используются инструкции, которые осуществляют переход на основе сравнения значений в регистрах, – beq (равны), bne (не равны), blt (меньше), bge (больше или равно) и их беззнаковые аналоги bltu/bgeu. Эти команды сравнивают значения двух регистров и, если условие выполнено, осуществляют переход на указанную метку.

Циклы

Циклы реализуются с помощью сочетания условных или безусловных переходов и меток, которые играют роль контрольных точек. Типичный цикл состоит из инициализации переменной, проверки условия, выполнения тела цикла и изменения счетчика. Для этого применяются инструкции addi, blt, bge, j и другие.

Логические операции

Язык ассемблера RISC-V поддерживает базовые побитовые логические операции, такие как and, or, xor, not (с использованием xori), а также операции сдвига (srl, sll, sra). Эти команды позволяют анализировать и модифицировать содержимое регистров на уровне отдельных битов.

Преобразование чисел в ASCII

Для отображения чисел в текстовом виде необходимо преобразовать их в символы кодировки ASCII. В случае с цифрами от 0 до 9 это выполняется путем прибавления значения 0x30 (или десятичного числа 48) к числовому значению.

Задание для лабораторной работы

Разработайте программу, которая выполняет следующие действия:

1. Принимает от пользователя одну цифру от 0 до 9 (ввод осуществляется в виде ASCII-символа).
2. Определяет, четное ли введенное число.
3. Если число четное – вычисляет сумму всех **четных** чисел от 0 до введенного числа (включительно).
4. Если число нечетное – вычисляет сумму всех **нечетных** чисел от 0 до введенного числа (включительно).
5. Выводит результат вычисления в консоль.

После реализации программы ответьте на теоретические вопросы для закрепления полученных знаний.

Теоретические вопросы

1. Что такое безусловный переход в языке ассемблера RISC-V и в каких случаях он применяется?
2. Какие инструкции используются для реализации условных переходов в RISC-V и каковы принципы их работы?
3. Как реализуются циклы в языке ассемблера RISC-V? Опишите общие принципы их построения.
4. Чем отличается j от jal?
5. Как в RISC-V реализуется цикл while? Приведите пример.
6. Как преобразовать цифру из ASCII в число и обратно?

Глава 5. Работа с памятью

Цель

Освоить принципы работы с памятью в архитектуре RISC-V и изучить такие темы, как определение данных в различных секциях программы, загрузка и сохранение данных с помощью инструкций, выравнивание данных, форматы хранения байтов, адресация, а также получить навыки преобразования строк и использования регистров для операций с памятью.

Теоретическая часть

В архитектуре RISC-V реализован принцип Load/Store, при котором все вычисления происходят только в регистрах, а доступ к памяти осуществляется через отдельные инструкции загрузки и сохранения данных. Для задания содержимого памяти используются директивы .byte, .word, .quad, .ascii, .asciz, .space и другие. Инициализированные данные размещаются в секции .data, неинициализированные – в секции .bss. Код программы и, при необходимости, неизменяемые данные располагаются в секции .text.

В архитектуре RISC-V используется порядок следования байтов Little-endian, при котором младшие байты располагаются по меньшим адресам. Для доступа к данным в памяти применяются инструкции загрузки адреса и значений, такие как la, auipc, lw, sw, lb, sb, а также комбинации auipc с lw или sw. При работе со строками нужно учитывать, что строки завершаются нулевым байтом, и требуется проверка на этот байт для корректного завершения цикла обработки.

Преобразование символов в верхний регистр может быть реализовано путем проверки границ диапазона символов в ASCII и арифметической операции со сдвигом. Для обеспечения корректного и эффективного размещения данных в памяти используется директива .align, позволяющая выровнять данные по границе слова или двойного слова.

Задание для лабораторной работы

Разработайте программу, моделирующую работу электронного журнала успеваемости. Опишем функции, которые должна выполнять программа.

Базовое задание. В начале выполнения программа содержит массив из трех целых чисел (оценок), размещенный в секции .data. Значения заданы заранее. Программа вычисляет сумму и среднее значение этих оценок. Результат сохраняется в переменной и используется для формирования строки отчета в формате "Average: X", где X – среднее арифметическое. Строка записывается в буфер и выводится на экран. Для вычисления среднего нужно использовать инструкцию div (*Глава 10. Умножение и деление*).

Продвинутое задание. Программа должна принимать оценки от пользователя через клавиатуру. Введенные значения сохраняются в массив, размещенный в секции .bss. Для получения максимального балла нужно реализовать подсчет положительных оценок (оценок, равных 4 или 5). Для этого использовать два отдельных счетчика:

- Первый счетчик – переменная размером 1 байт для накопления количества четверок;
- Второй счетчик – переменная размером 4 байта, размещается с применением директивы .align и используется для подсчета количества пятерок.

Оба счетчика должны участвовать в логике программы и отображаться на экране после основной части отчета.

После реализации программы ответьте на теоретические вопросы для закрепления полученных знаний.

Важно! При выполнении лабораторной работы в утилите RARS нужно учитывать, что она не поддерживает директиву .bss. Рекомендуется выполнять данную работу в эмуляторе QEMU.

Теоретические вопросы

1. Какие секции памяти используются в ассемблере RISC-V и каково их назначение?
2. Чем отличается директива .word от директивы .byte?
3. Для чего используется директива .align в RISC-V?
4. Что произойдет, если не использовать .align перед 4-байтовой переменной?
5. Что происходит с переменными, размещенными в секции .bss, при запуске программы?
6. Что такое Little-Endian и почему он важен при работе с памятью в RISC-V?

Глава 6. Функции и стек

Цель

Ознакомиться с использованием стека, вызовами функций и рекурсивных вычислений. Изучить структуру стековых кадров, соглашения о вызовах и порядок передачи аргументов. Освоить применение макрофункций для автоматизации операций сохранения и восстановления регистров. Освоить реализацию рекурсивных алгоритмов и методы преобразования чисел в строковый формат с последующим выводом на экран.

Теоретическая часть

Стек и регистр sp

Стек – это область памяти, используемая для временного хранения данных во время выполнения программы. Он нужен для поддержки вложенных вызовов функций, хранения локальных переменных, аргументов, а также промежуточных значений. В архитектуре RISC-V стек организован по принципу LIFO (последним помещен – первым извлечен) и растет в сторону уменьшения адресов. Работа со стеком осуществляется через регистр sp (stack pointer), указывающий на вершину стека. Перед записью в стек значение sp уменьшается, после извлечения – увеличивается.

Указатель стека sp (x2) всегда должен быть выровнен по границе 16 байт – при выделении места в стеке (операции push) значение sp необходимо уменьшать на число, кратное 16. Нарушение этого правила приведет к ошибке шины и аварийному завершению программы.

Вызов функций и регистр ra

Функции вызываются с помощью инструкций jal (переход по метке) или jalr (переход по адресу в регистре). При этом адрес следующей инструкции (адрес возврата) автоматически сохраняется в регистр ra (x1). Для возврата используется псевдоинструкция ret (которая транслируется в jalr x0, ra, 0).

Если функция внутри себя вызывает другую функцию (вложенный вызов или рекурсия), она обязана сохранить ra в стек перед вложенным вызовом и восстановить его перед ret, иначе программа не сможет вернуться в правильное место.

Соглашения о вызовах

Соглашения о вызовах (calling conventions) определяют порядок передачи параметров, возврата результатов и использования регистров:

- Аргументы передаются в регистрах a0-a7 (x10-x17).
- Результат возвращается в регистре a0.
- Регистр ra (x1) содержит адрес возврата и должен быть сохранен вручную, если в теле функции осуществляется вложенный вызов.
- Временные регистры t0-t6 (x5-x7, x28-x31) могут свободно использоваться вызываемой функцией без обязательного сохранения.
- Сохраняемые регистры s0-s11 (x8-x9, x18-x27) должны быть сохранены вызываемой функцией, если они используются.

Соблюдение соглашений о вызовах необходимо для обеспечения совместимости между различными частями программы и корректного возврата управления.

Стековые кадры

При вызове функции формируется стековый кадр – фрагмент стека, содержащий сохраненные значения регистров, локальные переменные и другую необходимую информацию. Стековый кадр формируется в начале функции путем уменьшения указателя sp и сохранения значений через инструкции sw. При завершении функции содержимое кадра восстанавливается, а указатель sp возвращается к исходному значению.

Макрофункции

Макрофункции позволяют группировать повторяющиеся конструкции в именованные блоки с помощью директив .macro и .endm. Их основное применение – автоматизация операций сохранения и восстановления регистров, подготовки стекового кадра и возврата из функций. Использование макросов упрощает структуру кода и снижает вероятность ошибок.

Рекурсия

Рекурсивные функции вызывают сами себя либо напрямую, либо через другие функции. Каждому вызову соответствует собственный стековый кадр с сохраненными аргументами и регистрами. Для корректной работы необходимо точно соблюдать правила сохранения состояния программы и предусматривать условие завершения. При отсутствии условия выхода возникает переполнение стека.

Преобразование чисел в строку

Для вывода числового значения в консоль необходимо выполнить его преобразование в строковый (ASCII) формат. Это достигается последовательным делением числа на основание системы счисления (обычно 10), сохранением остатков и формированием символов ('0' + остаток). Готовая строка размещается в буфере в памяти.

Задание для лабораторной работы

Реализуйте программу, вычисляющую факториал целого числа с использованием рекурсивной функции. Вызов функции должен быть организован через инструкцию jal с передачей аргумента через регистр a0. Внутри функции необходимо формировать стековый кадр, сохранять и восстанавливать регистры с использованием макросов.

Функция должна корректно обрабатывать граничные случаи (n = 0, n = 1) и возвращать результат в регистре a0. Вычисленное значение необходимо преобразовать в строку в десятичном формате и вывести на консоль с использованием системного вызова ecall. Все промежуточные данные, аргументы и адрес возврата должны передаваться через стек.

Исходное значение n должно быть задано непосредственно в коде программы (в регистре a0 перед вызовом функции).

После выполнения работы ответьте на теоретические вопросы для закрепления материала главы.

Теоретические вопросы

1. Какова роль стека в архитектуре RISC-V при работе с функциями?
2. Какие типы регистров используются в RISC-V и каково их назначение в соглашении о вызовах?

3. Для чего используется регистр ra и в каких случаях его необходимо сохранять?
4. Что такое стековый кадр и зачем он формируется при вызове функции?
5. Зачем и в каких случаях применяется рекурсия на уровне ассемблера?
6. Почему в RISC-V стек растет в сторону уменьшения адресов?
7. Что произойдет, если не сохранить регистры s0-11 в стек перед их использованием в функции?

Глава 7. Службы операционной системы Linux

Цель

Изучить взаимодействие программ на языке ассемблера с операционной системой Linux с помощью системных вызовов. Освоить механизм передачи аргументов и получения результатов через регистры, а также правила использования инструкции ecall. Разобраться в принципах работы с файлами на низком уровне: открытие, чтение, запись и закрытие файловых дескрипторов. Сформировать понимание структуры системных вызовов, их идентификаторов и обработки ошибок.

Теоретическая часть

Системный вызов представляет собой механизм, с помощью которого пользовательская программа запрашивает выполнение определенных действий от ядра операционной системы. Это единственный способ легального доступа к ресурсам ядра, таким как файловая система, ввод-вывод, управление процессами и памятью. В архитектуре RISC-V для выполнения системных вызовов используется инструкция ecall.

Перед вызовом ecall параметры передаются в регистры a0–a6. Номер системного вызова указывается в регистре a7. После выполнения системного вызова результат возвращается в регистр a0. Если результат положительный или равен нулю, вызов считается успешным. Отрицательные значения означают ошибку, и их необходимо обрабатывать.

Наиболее часто используемыми вызовами являются: открытие файлов (openat, номер 56), чтение (read, номер 63), запись (write, номер 64), закрытие файла (close, номер 57), завершение программы (exit, номер 93).

Для открытия файла через openat первым аргументом указывается файловый дескриптор директории (обычно -100 для текущей директории), вторым – адрес строки с именем файла, третьим – флаги открытия, четвертым – права доступа.

При выполнении вызовов чтения и записи необходимо указать файловый дескриптор, адрес буфера в памяти и количество байт для обработки. После завершения работы с файлом его необходимо закрыть с помощью вызова close, указав дескриптор.

Системные вызовы являются низкоуровневым интерфейсом, поэтому требуют внимательной проверки возвращаемых значений. В случае ошибки рекомендуется либо повторить попытку выполнения, либо зафиксировать ошибку в журнале или уведомить пользователя. Работа с файлами на уровне ассемблера требует четкого понимания того, как устроена адресация памяти и какие параметры нужно передавать при вызове.

Задание для лабораторной работы

Реализуйте программу, которая копирует содержимое одного файла в другой. Программа должна открыть файл-источник в режиме чтения, файл-приемник – в режиме записи (с созданием, если он не существует), считать из первого файла фиксированный блок данных (не более 32 байт), записать этот блок во второй файл и завершиться. Все действия должны выполняться с использованием системных вызовов операционной системы Linux через инструкцию ecall.

Имена обоих файлов и размер буфера заданы в секции данных. При выполнении задания необходимо обрабатывать коды возврата системных вызовов, корректно закрывать все открытые файловые дескрипторы и использовать буфер в памяти.

После выполнения работы ответьте на теоретические вопросы для закрепления материала главы.

Теоретические вопросы

1. Что такое системный вызов и зачем он используется?
2. Как осуществляется вызов системной функции в архитектуре risc-v?
3. Какие регистры используются для передачи параметров в системный вызов в архитектуре RISC-V? Какие вызовы используются для работы с файлами?
4. Как распознать, что системный вызов закончился ошибкой или успехом?
5. Какое значение используется в RISC-V в системном вызове openat для указания текущей директории?
6. Что следует сделать после окончания работы с открытым файлом?

Глава 8. Программирование контактов GPIO

Цель

Освоить работу с контактами общего назначения GPIO в операционной системе Linux. Изучить доступ к GPIO через виртуальную файловую систему /sys/class/gpio и через прямой доступ к регистрам ввода-вывода по физическим адресам через /dev/mem. Понять принципы инициализации контактов, управления направлением и состоянием вывода, обработки событий, реализации таймеров и оптимизации операций с помощью битовых инструкций.

Теоретическая часть

GPIO – это входы-выходы общего назначения, с помощью которых пользовательская программа может взаимодействовать с внешними электронными компонентами, такими как светодиоды, кнопки, реле и датчики. Каждый GPIO в зависимости от задачи может быть сконфигурирован как вход или выход.

Наиболее распространенный способ работы с GPIO в Linux – использование файловой системы /sys/class/gpio. Управление осуществляется путем записи в специальные файлы: export – для активации контакта, direction – для задания направления (in или out), value – для чтения или записи значения. Эти операции выполняются с помощью системных вызовов open, write и read.

Другой способ – прямой доступ к памяти с использованием /dev/mem. Этот метод требует отображения физических адресов регистров в адресное пространство программы с помощью функции mmap. Такой доступ обеспечивает высокую скорость работы и используется в задачах, чувствительных к задержкам, но требует повышенных привилегий и знания архитектуры конкретного микроконтроллера или процессора.

Для управления отдельными битами используются логические инструкции sll, srl, and, or, xor. Реализация задержек может выполняться с помощью системного вызова nanosleep с передачей параметров времени в структуре timespec. При необходимости обработки событий, например нажатия кнопки, используется функция poll, позволяющая организовать реакцию на изменение состояния контакта без постоянного опроса в цикле.

Задание для лабораторной работы

Реализуйте передачу сигнала SOS с помощью светодиода, подключенного к контакту GPIO, используя прямой доступ к регистрам ввода-вывода. Последовательность сигнала должна соответствовать азбуке Морзе: три коротких мигания (точки), три длинных мигания (тире), три коротких мигания. Каждое мигание означает включение и выключение светодиода с интервалами: точка – 200 мс, тире – 600 мс, пауза между миганиями – 200 мс. Между буквами пауза в 600 мс, между повторениями – 1000 мс.

Программа должна инициализировать системные часы, настроить порт GPIO на вывод, и бесконечно повторять сигнал SOS. Для реализации на плате MIK 32 Амур используйте встроенные функции HAL_GPIO_TogglePin, HAL_GPIO_Init, HAL_DelayMs, HAL_PCC_Config. Реализовать программу на языке ассемблера RISC-V без использования нестандартизированных макроинструкций.

После выполнения работы ответьте на теоретические вопросы для закрепления материала главы.

Теоретические вопросы

1. Что такое GPIO и какие режимы он поддерживает?
2. Как взаимодействовать с GPIO через файловую систему?
3. Что такое mmap и зачем он нужен?
4. В чем заключаются различия /sys/class/gpio и /dev/mem?
5. Почему прямой доступ к регистрам GPIO через /dev/mem требует повышенных привилегий?
6. Какие инструкции RISC-V используются для управления битами?
7. Какая системная функция используется для создания задержек в Linux?

Глава 9. Взаимодействие с языками C и Python

Цель

Изучить методы интеграции программ на языке ассемблера RISC-V с языками высокого уровня, такими как C и Python. Проанализировать то, как происходят вызовы функций, разработанных на C, из ассемблерного кода, и как производится разработка ассемблерных функций для использования в программах на C. Ознакомиться с основами взаимодействия с Python через структуры на C. Разобраться в том, какие преимущества дает сочетание низкоуровневого программирования с языками высокого уровня.

Теоретическая часть

Современные программы часто используют несколько языков программирования. Требовательные к производительности участки кода реализуются на ассемблере, а основная логика — на языках C или Python. Для успешной интеграции требуется соблюдение соглашений о вызовах, корректное представление данных и правильная компоновка.

Особенности линковки C и ассемблера

При совместном использовании C и ассемблера компилятор GNU C (gcc) берет на себя роль главного линковщика. Он автоматически подключает среду выполнения C (C Runtime), которая инициализирует стек, глобальные переменные и вызывает функцию main.

При использовании gcc для линковки точкой входа в программу является функция main (на языке C или ассемблере), а не _start. Использование _start при линковке с gcc приведет к конфликтам. Имена функций в ассемблере и C должны совпадать, однако важно избегать имен, которые заняты стандартной библиотекой C (например, toupper или printf), чтобы избежать ошибок множественного определения.

Вызов функций C из ассемблера

Для вызова из ассемблерного кода функций, разработанных на языке C, параметры передаются через регистры a0–a7, а результат возвращается в регистре a0. Необходимо соблюдать соглашение о вызовах RISC-V, включая управление стеком и сохранение регистров.

Для вызова ассемблерных подпрограмм из программ на C выполняются следующие условия:

1. Подпрограмма объявляется в C с использованием ключевого слова extern.
2. Имена подпрограмм должны быть уникальными, чтобы избежать конфликтов с библиотеками.
3. Ассемблерный код подключается через сборку и компоновку с использованием инструментов GNU, таких как as и ld.

Ассемблерный код может быть собран в статические библиотеки для повторного использования в программах на C. Это требует компоновки с помощью инструментов GNU и соблюдения правил оформления кода.

Ассемблерные вставки в коде C

Для выполнения низкоуровневых операций в программах на C применяется встроенный ассемблер с использованием директивы asm. Это позволяет оптимизировать отдельные участки кода.

Для вызова ассемблерных функций из Python используются C-расширения. Ассемблерный код интегрируется через промежуточные функции на C, что обеспечивает доступ к низкоуровневым операциям из интерпретируемого языка. Примеры кода и пояснения, приведенные далее, демонстрируют применение этих методов в среде Linux с использованием эмулятора QEMU для архитектуры RISC-V.

Разделяемые библиотеки и Python

Ассемблерный код можно скомпилировать в объектный файл (.o), упаковать в статическую (.a) или разделяемую (.so) библиотеку.

Разделяемые библиотеки позволяют вызывать ассемблерные функции из интерпретируемых языков, таких как Python, без написания сложных C-расширений. В Python для этого используется встроенный модуль ctypes, который позволяет загружать .so файлы и вызывать функции, указывая типы их аргументов.

Задание для лабораторной работы

Основное задание. Создайте функцию на ассемблере RISC-V, которая принимает два целых числа и возвращает их произведение. Вызовите эту функцию из программы на языке C, передав два числа и отобразив результат в консоли с использованием функции printf.

Примечание: умножение можно реализовать через цикл сложения.

Кроме того, реализуйте на языке C функцию, выводящую сообщение в консоль, и вызовите эту функцию в коде ассемблера.

Дополнительное задание. Выполните вызов ассемблерной функции из Python через C-расширение с выводом результата в интерпретаторе Python. Убедитесь, что у вас установлен пакет разработчика Python-dev для использования заголовочного файла *<Python.h>.*

```
sudo apt-get update
sudo apt-get install python3.10-dev
```

После выполнения работы ответьте на теоретические вопросы для закрепления материалов главы.

Теоретические вопросы

1. Как передаются параметры в функцию C из ассемблерного кода RISC-V?
2. Как сделать ассемблерную функцию доступной для вызова из программы на C?
3. Что такое статическая библиотека и как ее можно использовать с ассемблерным кодом?
4. Как объявляется ассемблерная функция для использования в коде на C?
5. Какая директива используется для встроенных ассемблерных вставок в коде C?
6. Как передаются результаты выполнения функции из ассемблерного кода обратно в код на C?

Глава 10. Умножение и деление

Цель

Изучить инструкции умножения и деления в архитектуре RISC-V, освоить их практическое применение при разработке программ на ассемблере.

Теоретическая часть

В данной главе представлены особенности обработки знаковых и беззнаковых чисел, примеры использования инструкций для получения полного результата арифметических операций, а также описан процесс обработки исключительных ситуаций, таких как деление на ноль и переполнение. Рассмотрена реализация алгоритма умножения матриц и интеграция ассемблерного кода с функциями языка C, такими как printf, для вывода результатов.

Архитектура RISC-V поддерживает расширение "M" (Multiply/Divide), которое добавляет в базовый набор инструкций RV32I/RV64I команды для выполнения операций умножения и деления. Эти операции применимы к целым числам – как знаковым, так и беззнаковым – и требуют учета особенностей их реализации.

Инструкции умножения

- mul rd, rs1, rs2 – выполняет знаковое умножение значений из регистров rs1 и rs2, сохраняя младшие 32 или 64 бита результата (в зависимости от разрядности архитектуры) в регистр rd.
- mulh rd, rs1, rs2 – возвращает старшие биты результата знакового умножения.
- mulhu rd, rs1, rs2 – возвращает старшие биты результата беззнакового умножения.
- mulhsu rd, rs1, rs2 – выполняет умножение знакового числа из rs1 на беззнаковое из rs2, сохраняя старшие биты результата.

Комбинация этих инструкций позволяет получать полный результат умножения, что важно при работе с числами, превышающими размер одного регистра.

Инструкции деления

- div rd, rs1, rs2 – выполняет знаковое деление rs1 на rs2, записывая частное в rd.
- divu rd, rs1, rs2 – выполняет беззнаковое деление.
- rem rd, rs1, rs2 – вычисляет знаковый остаток от деления.
- remu rd, rs1, rs2 – вычисляет беззнаковый остаток.

Особое внимание уделяется исключительным случаям. При делении на ноль инструкции div и divu возвращают предопределенные значения: -1 для знакового деления и максимальное беззнаковое значение (например, 0xFFFFFFFF для 32-битной архитектуры) для беззнакового.

Переполнение возникает при делении минимального знакового числа (INT_MIN) на -1. В этом случае результатом становится само INT_MIN, что соответствует спецификации RISC-V.

Умножение матриц

Умножение матриц – это фундаментальная операция в вычислительных задачах, требующая оптимизации работы с памятью и регистрами. Для матриц размером 3x3 необходимо выполнить 27 умножений и 18 сложений, что делает задачу ресурсоемкой на уровне ассемблера.

Интеграция с языком C

Для удобства отладки и вывода результатов используется функция printf из языка C, которая вызывается из ассемблерного кода. Это требует правильной передачи аргументов через регистры в соответствии с соглашением о вызовах RISC-V.

Задание для лабораторной работы

Разработайте программу, которая:

1. Обрабатывает массив из 5-ти целых 64-битных чисел, заданных в секции .data.
2. Умножает каждый элемент массива на заданный коэффициент (например, 3), используя инструкцию mul.
3. Вычисляет сумму результатов умножения.
4. Делит сумму на количество элементов массива для получения среднего значения, используя инструкцию div.
5. Обрабатывает исключительный случай: деление на ноль (если размер массива равен 0), возвращая код ошибки -1.
6. Преобразует результат (среднее значение или код ошибки) в строку и выводит его в консоль с помощью системного вызова write.

После выполнения работы ответьте на теоретические вопросы для закрепления материалов главы.

Пример входных данных:

- Массив: [2, 4, 6, 8, 10];
- Коэффициент умножения: 3;
- Размер массива: 5.

Ожидаемый результат: *(2*3 + 4*3 + 6*3 + 8*3 + 10*3) / 5 = 90 / 5 = 18.*

Теоретические вопросы

1. Какие инструкции RISC-V используются для умножения? Чем они отличаются?
2. В чем различие между инструкциями div и divu. Как отличаются результаты при их использовании?
3. Как обрабатывается деление на ноль в архитектуре RISC-V?
4. Что такое переполнение при делении и как оно проявляется в архитектуре RISC-V?
5. Какой результат вернет инструкция div при знаковом делении на ноль?
6. Для чего необходима инструкция mulh в архитектуре RISC-V?
7. С помощью каких инструкций можно получить полный результат умножения чисел, превышающих размер регистра?

Глава 11. Операции с числами с плавающей точкой

Цель

Изучить обработку чисел с плавающей точкой в архитектуре RISC-V (RV64FD) в формате IEEE-754 с двойной точностью (double). Освоить арифметические операции, сравнение чисел с учетом небольшой погрешности и работу с регистрами для чисел с плавающей точкой.

Теоретическая часть

Числа с плавающей точкой в RISC-V хранятся по стандарту IEEE-754. В формате двойной точности (double, 64 бита) число состоит из трех частей: один бит определяет знак (0 для положительного, 1 для отрицательного), 11 бит задают экспоненту, а 52 бита описывают дробную часть, называемую мантиссой. Экспонента показывает, насколько нужно сдвинуть мантиссу, чтобы получить значение числа. Такой формат позволяет работать с очень большими или очень маленькими числами, а также с особыми значениями, такими как бесконечность или «не число» (NaN), которые появляются, например, при делении на ноль или при ошибочных вычислениях.

Для работы с числами с плавающей точкой в RISC-V используются 32 регистра f0-f31. Каждый из них в RV64FD может хранить 64-битное число. Эти регистры применяются для арифметических операций, например, сложения или умножения. Числа загружаются из памяти в регистры с помощью команды fld и сохраняются обратно командой fsd. Для вычислений доступны команды вроде fadd.d (сложение), fsub.d (вычитание), fmul.d (умножение) или fdiv.d (деление). Есть также команда fsqrt.d для вычисления квадратного корня.

Сравнение чисел с плавающей точкой требует осторожности, потому что из-за ограничений в хранении чисел могут возникать небольшие погрешности. Например, вместо проверки на точное равенство двух чисел обычно вычисляют, насколько они близки, сравнивая их разность с маленьким числом, называемым epsilon. Для этого используются команды flt.d (меньше) или fle.d (меньше или равно).

Ошибки в вычислениях, такие как деление на ноль или переполнение, отслеживаются через специальный регистр fcsr. Он сохраняет информацию о возникших событиях, например, если результат слишком велик или произошла неточность. Также fcsr управляет тем, как округляются числа во время вычислений, например к ближайшему четному.

Иногда нужно преобразовать число с плавающей точкой в целое или наоборот. Для этого используются команды fcvt.d.w и fcvt.w.d. Если же нужно просто скопировать данные между регистрами без изменения, применяются команды fmv.d.x или fmv.x.d.

Задание для лабораторной работы

Создайте на ассемблере RISC-V (RV64FD) программу, которая вычисляет значение выражения a + b × c для трех чисел с плавающей точкой (в формате double) и проверяет, насколько результат близок к ожидаемому значению. Программа должна выполнить следующие шаги:

1. Загрузить из памяти числа a, b, c и ожидаемое значение ref.
2. Вычислить b × c, затем прибавить a, сохранив результат в регистр f3.
3. Найти абсолютную разность |f3 - ref| и сохранить ее в регистр f4.

4. Проверить, меньше ли эта разность заданного значения $\varepsilon = 1 \times 10^{-10}$, и записать результат сравнения (1 или 0) в регистр x10.
5. Вывести в консоль сообщение "OK", если $|f3 - ref| < \varepsilon$, или "NOT OK", если $|f3 - ref| \geq \varepsilon$, используя системные вызовы Linux.

После выполнения работы ответьте на теоретические вопросы для закрепления материала главы.

Теоретические вопросы

1. Как устроено число с плавающей точкой в формате IEEE-754 (double)? Из каких частей оно состоит?
2. Почему нельзя сравнивать числа с плавающей точкой на точное равенство? Как правильно сравнивать?
3. Какие команды используются для загрузки и сохранения чисел с плавающей точкой в RISC-V?
4. Что хранит регистр fcsr и для чего он нужен?
5. В чем разница между командами fcvt.d.w и fmv.d.x?
6. Какое количество битов используется для хранения экспоненты и мантиссы в формате double?
7. Какая инструкция RISC-V используется для проверки условия «меньше или равно» для чисел с плавающей точкой?
8. Какое значение обычно используют для проверки погрешности при сравнении чисел с плавающей точкой?

Глава 12. Оптимизация исходного кода

Цель

Изучить методы оптимизации программ, разработанных на языке ассемблера RISC-V, и применить эти методы для повышения производительности кода, а также освоить навыки разработки и тестирования оптимизированных программ.

Теоретическая часть

Оптимизация программ на языке ассемблера RISC-V представляет собой важный этап разработки для повышения эффективности кода. В главе подробно рассматриваются ключевые подходы к оптимизации, среди которых: минимизация количества инструкций, эффективное использование регистров, оптимизация циклов и ветвлений, а также применение специализированных инструкций, характерных для архитектуры RISC-V.

Минимизация инструкций достигается за счет устранения избыточных операций и выбора наиболее подходящих команд для выполнения задачи. Эффективное использование регистров позволяет сократить время доступа к данным, так как время обращения к регистрам значительно меньше, чем к оперативной памяти.

Оптимизация циклов, в частности, разворачивание циклов, помогает уменьшить накладные расходы на управление и увеличить пропускную способность программы. Ветвления, которые часто приводят к задержкам из-за неправильного предсказания переходов, можно минимизировать или заменить более предсказуемыми конструкциями.

Наконец, специализированные инструкции RISC-V, такие как инструкции для работы с целыми числами или плавающей точкой, могут быть использованы для ускорения выполнения специфических задач.

Задание для лабораторной работы

Предлагается неоптимизированная версия программы, которая:

1. Инициализирует массив из 1 000 000 элементов, где каждый элемент равен его индексу (0, 1, 2, ..., 999999).
2. Выполняет суммирование квадратов элементов массива в цикле, повторяя эту операцию 100 раз.
3. Возвращает последнюю сумму как код выхода из программы.

Необходимо оптимизировать эту программу, чтобы уменьшить время ее выполнения, сохранив при этом корректность результата. Нужно использовать разворачивание цикла (loop unrolling), а также применить другие методы оптимизации (например, минимизацию лишних инструкций или улучшение доступа к памяти).

```
      .section .data
array: .space 4000000  # 1,000,000 элементов * 4 байта
array_end: .quad array + 4000000

.section .text
.global main

main:
    # Инициализация массива: array[i] = i
```

```
    la t0, array
    li t1, 0          # i = 0
    li t2, 1000000    # размер массива
init_loop:
    bge t1, t2, init_end
    sw t1, 0(t0)      # array[i] = i
    addi t0, t0, 4
    addi t1, t1, 1
    j init_loop
init_end:

    # Внешний цикл: повторить суммирование 100 раз
    li t3, 0          # счетчик внешнего цикла
    li t4, 100        # количество итераций
outer_loop:
    bge t3, t4, outer_end
    # Суммирование квадратов элементов
    la t0, array
    la t1, array_end
    li t2, 0          # сумма = 0
sum_loop:
    bge t0, t1, sum_end
    lw t5, 0(t0)      # загрузить array[i]
    mul t5, t5, t5    # t5 = array[i] * array[i]
    add t2, t2, t5    # sum += t5
    addi t0, t0, 4    # следующий элемент
    j sum_loop
sum_end:
    addi t3, t3, 1    # увеличить счетчик внешнего цикла
    j outer_loop
outer_end:
    mv a0, t2         # последняя сумма в a0
    li a7, 93         # системный вызов для выхода
    ecall
```

Программа предназначена для запуска в среде эмуляции QEMU. Проверьте скорость работы программы с помощью команд ниже (при условии, что файл с программой называется arr.S).

```
      riscv64-linux-gnu-gcc arr.S -o arr
./arr
echo $?
time ./arr
> real 0m0,215s
> user 0m0,208s
> sys  0m0,007s
```

Код выхода программы в RISC-V передается через регистр a0, но он обрезается до 8 бит (значения от 0 до 255) при вызове команды > echo $? , поэтому в результате число будет небольшим. Не придавайте этому значение. Это отладочный вывод для того, чтобы убедиться, что программа выполняет арифметические вычисления. Замерить скорость работы программы можно с помощью утилиты time.

Теоретические вопросы

1. Какие методы оптимизации наиболее эффективны для программ на языке ассемблера RISC-V? Почему важно их применять?
2. Как использование регистров влияет на производительность программы и какие ограничения при этом возникают?
3. В чем заключаются преимущества и недостатки разворачивания циклов как метода оптимизации?
4. Как можно минимизировать количество инструкций в программе и к принятию каких компромиссов это приводит?
5. Какие специализированные инструкции RISC-V можно использовать для оптимизации и в каких случаях они применимы?
6. Как разворачивание циклов влияет на размер итогового кода?
7. Почему минимизация ветвлений важна для производительности программы?
8. Какой доступ быстрее: к регистрам или к оперативной памяти? Почему?
9. В чем заключается суть оптимизации через устранение избыточных операций?

Глава 13. Чтение и понимание исходного кода

Цель

Развить навыки анализа и интерпретации исходного кода на языке ассемблера RISC-V, включая код, сгенерированный компиляторами, и освоить использование инструмента Ghidra для декомпиляции и реверс-инжиниринга бинарных файлов. Научиться понимать структуру кода, находить оптимизации и специфические инструкции RISC-V.

Теоретическая часть

Глава посвящена тому, как читать и понимать исходный код на языке ассемблера RISC-V. В главе основное внимание уделяется анализу кода, сгенерированного компиляторами (например, GCC), и использованию инструментов для реверс-инжиниринга, таких как Ghidra. Рассматриваются следующие ключевые аспекты:

Зачем нужно понимать ассемблерный код

Знание структуры кода помогает в отладке, оптимизации программ и выявлении уязвимостей. Например, анализ ядра Linux на уровне ассемблера позволяет понять, как реализованы макрофункции для повышения производительности.

Компиляция высокоуровневого кода

Компиляторы, такие как GCC, преобразуют код на C в инструкции RISC-V. В том числе касается использования регистров, оптимизации циклов и применения расширений, таких как Zbb (битовые манипуляции).

Инструмент Ghidra

Ghidra – это продвинутый инструмент для декомпиляции бинарных файлов. Он преобразует машинный код в читаемый вид, позволяя восстановить логику программы даже без исходного кода. Пример использования: анализ функции сравнения строк может показать, как компилятор оптимизирует ее с помощью инструкций RISC-V.

Сопоставление декомпилированного кода с его поведением при выполнении (например, в QEMU) помогает понять, как архитектура RISC-V влияет на исполнение программы.

Задание для лабораторной работы

Необходимо проанализировать бинарный файл, скомпилированный из простого кода на C, с использованием Ghidra и QEMU. Нужно выполнить следующие шаги:

1. Загрузить файл mystery.bin в Ghidra и выполнить его автоматический анализ.
2. Найти основную функцию и изучить декомпилированный код.
3. Определить, что делает программа.
4. Найти не менее двух инструкций RISC-V (например, add, lw), используемых в коде, и объяснить их роль в программе.
5. Попробовать изменить код программы в Ghidra и запустить бинарный файл с изменениями. Описать результат.

В качестве решения нужно предоставить отчет с ответами на вопросы и скриншотами работы с Ghidra.

Теоретические вопросы

1. Как использование сжатых инструкций RISC-V влияет на анализ ассемблерного кода в Ghidra?
2. Почему анализ точки входа _start важен для понимания программы на RISC-V?
3. Какие преимущества и недостатки дает использование динамической линковки при анализе RISC-V кода в Ghidra?
4. Что позволяет сделать инструмент Ghidra с бинарным файлом
5. Почему важно сопоставлять декомпилированный код с его поведением при выполнении?
6. Какие виды инструкций можно обнаружить при анализе сгенерированного компилятором кода?

Глава 14. Взлом исходного кода

Цель

Ознакомиться с методами защиты программ, разработанных на языке ассемблера RISC-V, от распространенных видов атак, таких как переполнение буфера. Рассмотреть типичные уязвимости, их причины и последствия, а также практические подходы и техники, направленные на повышение безопасности кода. Изучить методы предотвращения несанкционированного доступа к данным и исполнению вредоносного кода, имеющие ключевое значение для обеспечения надежности и безопасности программного обеспечения.

Теоретическая часть

Глава посвящена анализу уязвимостей программ, разработанных на языке ассемблера RISC-V, и методам их защиты от различных атак. Одна из них – переполнение буфера, которая возникает, когда программа записывает данные за пределы выделенного массива, что может привести к повреждению памяти, выполнению произвольного кода или сбою системы.

К основным причинам возникновения таких уязвимостей относится отсутствие проверки размеров входных данных и небезопасное использование функций обработки строк, таких как strcpy или gets в высокоуровневых языках, которые могут быть отражены в ассемблерном коде через прямую работу с памятью. В контексте RISC-V переполнение буфера опасно из-за прямого управления стеком и регистрами, где ошибочная запись может перезаписать адрес возврата или критические данные. Рассматриваются типы атак, использующие переполнение буфера, включая внедрение вредоносного кода в стек и перенаправление потока управления на злонамеренные инструкции.

Для предотвращения подобных уязвимостей применяются несколько подходов. Первый – проверка границ массивов, которая требует явного контроля индексов и размеров данных в коде. Второй – использование «канареек стека», специальных значений, размещаемых перед адресом возврата, для обнаружения попыток перезаписи стека. Третий – рандомизация адресного пространства (ASLR), усложняющая предсказание адресов памяти для атак. Также применяются безопасные функции обработки данных, минимизирующие риск возникновения переполнения.

В RISC-V архитектуре защита осложняется из-за использования минималистичного набора инструкций, требующего ручного управления памятью, но это же позволяет реализовать точечные оптимизации безопасности. Анализируются инструменты статического и динамического анализа кода, включая Ghidra, для выявления уязвимостей. Ghidra позволяет декомпилировать бинарный код, выявляя потенциально опасные операции, такие как прямой доступ к памяти без проверки границ.

Рассматриваются примеры уязвимого кода на RISC-V, где отсутствие проверки входных данных приводит к переполнению, и примеры защищенного кода, использующего канарейки и проверки. Обсуждаются последствия наличия уязвимостей, включая утечку данных, нарушение целостности системы и потенциальный доступ к привилегиям. Особое внимание уделяется разработке кода с учетом принципов безопасного программирования, таких как минимизация привилегий и изоляция критических данных.

Задание для лабораторной работы

Предлагается изучить файл buffer.s, содержащий функцию на языке ассемблера RISC-V, уязвимую к атаке переполнения буфера. Требуется выполнить следующие шаги:

1. Проанализировать функцию, выявить уязвимость переполнения буфера и объяснить, как она позволяет перезаписать адрес возврата.
2. Модифицировать ассемблерный код, внедрив механизм канарейки стека для защиты от переполнения буфера.
3. Скомпилировать измененный код и протестировать его, сравнив поведение исходной и защищенной версий при вводе данных, вызывающих переполнение буфера.

Если запустить исходную версию программы, она попросит ввести значение и выведет его на экран. При этом буфер составляет 128 символов. Все символы после 128 не попадают в буфер, а считываются как команда. В данном случае была введена строка с последовательностью символов, разделенных запятой и пробелом. После числа 34 считывается только разряд десятков, а после него можно ввести любую команду.

```
root@riscv-ubuntu:/home/myfolder# qemu-riscv64 ./buffer_vuln.bin

Enter input:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 3ls

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 3?

root@riscv-ubuntu:/home/myfolder# ls

buffer.bin  buffer.s  buffer_secure.bin  buffer_secure.s  buffer_vuln.bin
```

Теоретические вопросы

1. Почему стек наиболее уязвим к атакам переполнения буфера? Объясните механизм перезаписи адреса возврата.
2. Что такое shellcode и как он используется при эксплуатации переполнения стека?
3. Что такое «канарейка» в контексте главы и в какой части памяти она размещается? Опишите принцип ее работы.
4. Что такое ASLR и как оно усложняет эксплуатацию уязвимостей?
5. В чем разница между защитами: DEP (NX), ASLR и PIE? Как они взаимодействуют для предотвращения атак?
6. Как можно обойти защиту на уровне компиляции (например, `-fstack-protector`, PIE), и почему ручной анализ остается важным методом предотвращения атак на код программы?
7. Почему важно минимизировать привилегии при разработке программ?

Приложение 1

Лабораторная работа 1

В первой работе не требуется разарбатывать код. Нужно установить и настроить среду и окружение для работы.

Лабораторная работа 2

Пример возможной реализации программы

```
    .data
x:      .word 10            # данные: x = 10
y:      .word 20            #         y = 20
msg:    .ascii "Result: "   # резервируем строку "result" для вывода в консоль
buf:    .space 2            # буфер под две ASCII-цифры результата
nl:     .ascii "\n"         # перевод строки

    .text
    .globl _start
    .align 2
_start:
    # 1.1 Загрузка x и y из памяти
    la    t0, x             # t0 ← адрес x
    lw    t0, 0(t0)         # t0 ← x
    la    t1, y             # t1 ← адрес y
    lw    t1, 0(t1)         # t1 ← y

    # 1.2 Вычисление выражения E = (x + y) * 2 + 4
    add   t2, t0, t1        # t2 = x + y
    slli  t2, t2, 1         # t2 = t2 * 2
    addi  t2, t2, 4         # t2 = t2 + 4

    # Разбиваем результат на десятки и единицы (допускаем только E меньше 100)
    li    t3, 10            # делитель = 10
    div   t4, t2, t3        # t4 = t2 / 10  → цифра в разряде десятков
    rem   t5, t2, t3        # t5 = t2 % 10  → цифра в разряде единиц

    addi  t4, t4, '0'       # переводим цифры в ASCII
    addi  t5, t5, '0'

    # Записываем их в буфер buf[0..1]
    la    t6, buf           # t6 ← адрес buf
```

```
addi  t6, t6, 2        # t6 ← buf + 2 (конец буфера)
addi  t6, t6, -1       # t6 ← buf + 1
sb    t5, 0(t6)        # buf[1] = разряд единиц
addi  t6, t6, -1       # t6 ← buf + 0
sb    t4, 0(t6)        # buf[0] = разряд десятков

# Пояснения:
#  - для вывода строк и буферов используется функция sys_write (номер 64)
#  - в a0 -- дескриптор потока (1 = stdout)
#  - в a1 -- адрес данных, в a2 -- длина, в a7 -- номер системного вызова
#  - инструкция ecall запускает системный вызов

# 3.1 Вывод префикса "Result: ", хранимого в переменной msg
li    a0, 1            # a0 = 1 → stdout
la    a1, msg          # a1 = адрес строки "Result: "
li    a2, 8            # a2 = 8 байт
li    a7, 64           # a7 = sys_write
ecall                  # печать префикса

# 3.2 Вывод цифр результата из buf
li    a0, 1
mv    a1, t6           # t6 сейчас = buf
li    a2, 2            # длина = 2 байта
li    a7, 64
ecall                  # печать цифр

# 3.3 Перевод строки
li    a0, 1
la    a1, nl
li    a2, 1
li    a7, 64
ecall                  # печать "\n"

li    a0, 0            # код возврата (0 - код успеха)
li    a7, 93           # sys_exit
ecall                  # выход
```

Оценивание лабораторной работы

Пункт	Баллы
Программа имеет подробное описание и комментарии.	4
Реализованы вычисление выражения из условия и сохранение результата.	4
Перевод результата в ASCII и вывод в консоль.	2

Лабораторная работа 3.1

Пример возможной реализации программы

Корректно работающий код расположен в главной ветке репозитория: **https://github.com/RomeoMe5/riscv-assembly-programming/tree/master/lab_03/1_make/solution**

Оценивание лабораторной работы

Пункт	Баллы
Программа собирается корректно.	4
Реализована поддержка инкрементной сборки.	2
Компактное определение переменных SOURCES и OBJECTS. Использование функции patsubst при определении OBJECTS.	2
Компактные и понятные правила сборки (имеется наглядная зависимость объектных файлов от исходных). Отсутствие дублирования кода.	2

Лабораторная работа 3.2

Пример возможной реализации программы

Корректно работающий код расположен в репозитории: **https://github.com/RomeoMe5/riscv-assembly-programming/tree/master/lab_03/2_gdb/solution**

Оценивание лабораторной работы

Пункт	Баллы
Приведен анализ работы программы с ошибкой в GDB.	4
Приведено описание и исправлены ошибки, выполнена демонстрация работы исправленной программы.	4
Выполнена демонстрация работы с микроконтроллером MIK32 Амур (опционально).	2

Лабораторная работа 4

Пример возможной реализации программы

```
      data
    prompt:       .asciz "Введите цифру от 0 до 9: "   # Сообщение пользователю
    result_text:  .asciz "\nСумма: "                   # Текст перед выводом суммы
    newline:      .asciz "\n"                          # Перенос строки
    input_buffer: .space 4                             # Буфер для хранения ввода
    zero_ascii:   .byte '0'                            # ASCII-символ '0'
                                                       # (для вывода нуля)

.text
.globl main
main:
    # Вывод сообщения: "Введите цифру от 0 до 9: "
    li a7, 4
    la a0, prompt
    ecall

    # Чтение строки (до 2 символов) в input_buffer
    li a7, 8
    la a0, input_buffer
    li a1, 2
    ecall

    # Преобразование ASCII-символа в число
    la t6, input_buffer       # Адрес буфера
    lb t0, 0(t6)              # Считываем первый байт (введенный символ)
    addi t0, t0, -48          # Перевод из ASCII в число ('0' → 0, ..., '9' → 9)

    # Определение четности: t1 = t0 % 2
    andi t1, t0, 1

    li t2, 0     # t2 -- аккумулятор суммы
    li t3, 0     # Не используется, но инициализирован

    # Если число четное, переходим к even_loop
    beqz t1, even_loop

# Если число нечетное -- начинаем с 1 и суммируем нечетные числа
```

```
odd_loop:
    li t4, 1                    # Счетчик начинается с 1
odd_loop_start:
    bgt t4, t0, done            # Пока счетчик <= введенное число
    add t2, t2, t4              # t2 += t4
    addi t4, t4, 2              # Переход к следующему нечетному числу
    j odd_loop_start

# Если число четное -- начинаем с 0 и суммируем четные числа
even_loop:
    li t4, 0
even_loop_start:
    bgt t4, t0, done
    add t2, t2, t4
    addi t4, t4, 2
    j even_loop_start

done:
    # Вывод строки "Сумма: "
    li a7, 4
    la a0, result_text
    ecall

    # Перевод значения суммы (t2) в строку (ASCII)
    li t3, 10          # Основание 10
    li t4, 0           # Счетчик количества цифр
    mv t6, t2          # Копируем результат в t6 для преобразования

print_loop:
    # Если t6 == 0, значит либо сумма была 0, либо деление закончено
    beqz t6, check_zero

    # Получаем последнюю цифру числа
    remu t5, t6, t3           # Остаток от деления на 10 (t5 = t6 % 10)
    addi t5, t5, 48           # Преобразуем в ASCII (0 → '0')
    addi sp, sp, -1           # Сдвигаем стек (готовим место под байт)
    sb t5, 0(sp)              # Сохраняем символ на вершину стека
    addi t4, t4, 1            # Увеличиваем счетчик символов
    divu t6, t6, t3           # t6 = t6 / 10
    j print_loop
```

```
check_zero:
    # Если не было ни одной цифры (t4 == 0), значит результат был 0
    # Нужно явно вывести символ '0'
    beq t4, zero, print_zero

    # Иначе печатаем строку с цифрами из стека
    li a7, 64               # write
    li a0, 1                # stdout
    mv a1, sp               # Адрес начала цифр (в стеке)
    mv a2, t4               # Количество байт
    ecall
    j newline_out

print_zero:
    # Специальный случай: результат = 0. Печатаем символ '0'
    li a7, 64
    li a0, 1
    la a1, zero_ascii       # Указатель на символ '0'
    li a2, 1
    ecall

newline_out:
    # Перевод строки
    li a7, 4
    la a0, newline
    ecall

    # Завершение программы
    li a7, 10
    ecall
```

Оценивание лабораторной работы

Пункт	Баллы
Программа принимает ввод, определяет четность числа и корректно выводит результат (является число четным или нечетным).	6
Реализован расчет суммы четных или нечетных чисел.	2
Программа выводит итоговую сумму в консоль в формате ASCII.	2

Лабораторная работа 5

Пример возможной реализации программы

```
  .data
    prompt:      .asciz "Enter grades:\n"       # Ввод в одну строку через пробел
                                                                                #
(пример: 4 5 3)

      avg_text:    .asciz "Average: "
    fours_text:  .asciz "\nNumber of 4s: "
    fives_text:  .asciz "\nNumber of 5s: "
    newline:     .asciz "\n"
    input_buf:   .space 32
    output_buf:  .space 4

.bss
    .lcomm grades, 12
    .lcomm fours_count, 1
    .align 2
    .lcomm fives_count, 4

.text
.globl _start
_start:
    # Вывод строки prompt
    li a0, 1
    la a1, prompt
    li a2, 15
    li a7, 64
    ecall

    # Чтение ввода
    li a0, 0
    la a1, input_buf
    li a2, 32
    li a7, 63
    ecall

    # Инициализация
    la t0, input_buf
```

```
    la t1, grades
    li t2, 0
    la t3, fours_count
    sb zero, 0(t3)
    la t4, fives_count
    sw zero, 0(t4)

parse_loop:
    lb t5, 0(t0)
    beq t5, zero, end_parse
    li t6, 32
    beq t5, t6, skip
    li t6, 10
    beq t5, t6, end_parse
    li t6, 48
    blt t5, t6, skip
    li t6, 57
    bgt t5, t6, skip

    addi t5, t5, -48
    sw t5, 0(t1)
    addi t1, t1, 4

    # Подсчет оценок
    li t6, 4
    beq t5, t6, inc_four
    li t6, 5
    beq t5, t6, inc_five
    j skip_inc

inc_four:
    la t3, fours_count
    lb t6, 0(t3)
    addi t6, t6, 1
    sb t6, 0(t3)
    j skip_inc

inc_five:
    la t4, fives_count
    lw t6, 0(t4)
```

```
    addi t6, t6, 1
    sw t6, 0(t4)

skip_inc:
    addi t2, t2, 1
    li t6, 3
    beq t2, t6, end_parse

skip:
    addi t0, t0, 1
    j parse_loop

end_parse:
    la t1, grades
    lw t3, 0(t1)
    lw t4, 4(t1)
    lw t5, 8(t1)
    add s0, t3, t4
    add s0, s0, t5
    li s1, 3
    div s2, s0, s1
    la t0, output_buf
    addi s2, s2, 48
    sb s2, 0(t0)
    sb zero, 1(t0)

    # Вывод среднего арифметического
    li a0, 1
    la a1, avg_text
    li a2, 9
    li a7, 64
    ecall
    li a0, 1
    la a1, output_buf
    li a2, 1
    li a7, 64
    ecall

    # Вывод кол-ва оценок 4
    li a0, 1
```

```
    la a1, fours_text
    li a2, 17
    li a7, 64
    ecall
    la t0, fours_count
    lb a0, 0(t0)
    addi a0, a0, 48
    la t1, output_buf
    sb a0, 0(t1)
    li a0, 1
    la a1, output_buf
    li a2, 1
    li a7, 64
    ecall

    # Вывод кол-ва оценок 5
    li a0, 1
    la a1, fives_text
    li a2, 17
    li a7, 64
    ecall
    la t0, fives_count
    lw a0, 0(t0)
    addi a0, a0, 48
    la t1, output_buf
    sb a0, 0(t1)
    li a0, 1
    la a1, output_buf
    li a2, 1
    li a7, 64
    ecall

    # Перенос строки
    li a0, 1
    la a1, newline
    li a2, 1
    li a7, 64
    ecall

    # Выход из программы
```

```
    li a0, 0
    li a7, 93
    ecall
```

Оценивание лабораторной работы

Пункт	Баллы
Программа использует заранее заданные оценки, вычисляет сумму и среднее значение, формирует строку отчета и выводит ее на экран.	6
Оценки вводятся пользователем и сохраняются в секции .bss.	2
Реализован подсчет количества четверок и пятерок с использованием 1-байтовой и 4-байтовой переменных соответственно. Переменная для пятерок размещена с применением директивы .align. Значения из счетчиков выводятся на экран.	2

Лабораторная работа 6

Пример возможной реализации программы

```
   .data
buffer:         .space 32               # буфер под строку результата
label:          .asciz "Factorial:\n"   # строка с переводом строки
newline:        .byte 10                # отдельный перевод строки

    .text
    .globl _start

.macro SAVE_REGS
    addi sp, sp, -12                    # резервируем место в стеке
    sw ra, 8(sp)                        # сохраняем ra
    sw s0, 4(sp)                        # сохраняем s0
    sw s1, 0(sp)                        # сохраняем s1
.endm

.macro RESTORE_REGS
    lw s1, 0(sp)                        # восстанавливаем s1
    lw s0, 4(sp)                        # восстанавливаем s0
    lw ra, 8(sp)                        # восстанавливаем ra
    addi sp, sp, 12                     # возвращаем sp
.endm
```

```
# функция факториала: вход -- a0, выход -- a0
factorial:
    SAVE_REGS
    li t0, 1
    ble a0, t0, base_case

    mv s0, a0
    addi a0, a0, -1
    jal ra, factorial
    mul a0, a0, s0
    j factorial_end

base_case:
    li a0, 1

factorial_end:
    RESTORE_REGS
    jalr zero, 0(ra)

# преобразование числа в строку
int_to_str:
    li t0, 10
    la t1, buffer
    mv t2, t1
    li t3, 0

to_str_loop:
    rem t4, a0, t0
    addi t4, t4, 48
    sb t4, 0(t1)
    addi t1, t1, 1
    addi t3, t3, 1
    div a0, a0, t0
    bnez a0, to_str_loop

    sb zero, 0(t1)

    li t5, 0
    addi t3, t3, -1
reverse_loop:
```

```
    bge t5, t3, reverse_done

    lb t6, 0(t2)
    add t4, t2, t3
    lb a4, 0(t4)

    sb a4, 0(t2)
    sb t6, 0(t4)

    addi t5, t5, 1
    addi t2, t2, 1
    addi t3, t3, -1
    j reverse_loop

reverse_done:
    # добавляем символ новой строки в конец
    li t0, 10
    sb t0, 0(t1)
    addi t1, t1, 1
    sb zero, 0(t1)
    ret

# вывод строки по адресу в a1, длина в a2
print_string:
    li a0, 1
    li a7, 64
    ecall
    ret

# точка входа
_start:
    li a0, 5          # кладем в a0 значение числа, факториал которого будем
вычислять
    jal ra, factorial
    mv s1, a0

    la a1, label
    li a2, 11                    # длина строки "Factorial:\n"
    jal ra, print_string
```

```
    mv a0, s1
    jal ra, int_to_str

    la a1, buffer
    li a2, 32
    jal ra, print_string

    li a7, 93
    ecall
```

Оценивание лабораторной работы

Пункт	Баллы
Реализована функция факториала без макросов, с возвратом значения через a0, без строкового преобразования и форматированного вывода.	6
Использованы макросы для сохранения регистров, преобразования результата в строку вручную. Реализован вывод результата в консоль.	2
Использованы все инструменты: стековый кадр, макросы, рекурсивная функция, ручное преобразование в строку, форматированный вывод с заголовком и корректное завершение программы.	2

Лабораторная работа 7

Пример возможной реализации программы

```
     .data
src_path:       .asciz "input.txt"              # имя файла-источника
dst_path:       .asciz "output.txt"             # имя файла-приемника
buf:            .space 64                       # буфер для копирования

error_open:     .asciz "Не удалось открыть файл\n"
error_read:     .asciz "Ошибка чтения файла\n"
error_write:    .asciz "Ошибка записи в файл\n"

    .text
    .globl _start

_start:
    # открываем файл-источник на чтение
    li a0, -100
```

```
    la a1, src_path
    li a2, 0
    li a3, 0
    li a7, 56
    ecall
    blt a0, zero, open_src_failed
    mv s0, a0               # дескриптор источника

    # открываем файл-приемник с созданием
    li a0, -100
    la a1, dst_path
    li a2, 577
    li a3, 420
    li a7, 56
    ecall
    blt a0, zero, open_dst_failed
    mv s1, a0               # дескриптор приемника

read_loop:
    # читаем блок из источника
    mv a0, s0
    la a1, buf
    li a2, 64
    li a7, 63
    ecall
    blt a0, zero, read_failed
    beqz a0, close_all      # если a0 == 0, конец файла
    mv s2, a0               # сохраняем количество байт

    # записываем блок в приемник
    mv a0, s1
    la a1, buf
    mv a2, s2
    li a7, 64
    ecall
    blt a0, zero, write_failed

    j read_loop             # продолжаем

close_all:
```

```
    # закрываем оба файла
    mv a0, s0
    li a7, 57
    ecall

    mv a0, s1
    li a7, 57
    ecall

    li a7, 93
    li a0, 0
    ecall

open_src_failed:
    la a1, error_open
    li a2, 29
    li a0, 1
    li a7, 64
    ecall
    j exit_error

open_dst_failed:
    la a1, error_open
    li a2, 29
    li a0, 1
    li a7, 64
    ecall
    j exit_error

read_failed:
    la a1, error_read
    li a2, 27
    li a0, 1
    li a7, 64
    ecall
    j exit_error

write_failed:
    la a1, error_write
    li a2, 28
```

```
    li a0, 1
    li a7, 64
    ecall
    j exit_error

exit_error:
    li a7, 93
    li a0, 1
    ecall
```

Оценивание лабораторной работы

Пункт	Баллы
Программа открывает один файл, читает фиксированное количество байт и выводит их в консоль. Работа с одним дескриптором, нет копирования в другой файл.	6
Программа открывает два файла, копирует блок фиксированной длины из одного файла в другой, но не реализует проверку ошибок или закрытие файлов.	2
Использованы все инструменты: открытие двух файлов, копирование блока через буфер, обработка кода возврата, закрытие обоих файлов, все данные и параметры заданы в секции данных.	2

Лабораторная работа 8

Пример возможной реализации программы

```
  .text
.align 2

# Точка входа программы
.globl main
.type main, @function
main:
    # Пролог: настройка стека
    addi sp, sp, -16      # Выделяем 16 байт на стеке
    sw   ra, 12(sp)       # Сохраняем адрес возврата
    sw   s0, 8(sp)        # Сохраняем регистр s0
    addi s0, sp, 16       # Устанавливаем s0 как указатель на базу стека

    # Инициализация системы
```

```
    call SystemClock_Config  # Настраиваем системные часы
    call GPIO_Init           # Инициализируем GPIO

    # Бесконечный цикл переключения пина для сигнала SOS
loop:
    # Три коротких сигнала
    li   a0, 0x00084400       # Адрес порта GPI1 (GPIOA_BASE)
    li   a1, 4                # Номер пина (GPIO_PIN_9)
    call HAL_GPIO_TogglePin   # Включить
    li   a0, 200              # Задержка 200 мс
    call HAL_DelayMs
    li   a0, 0x00084400
    li   a1, 4
    call HAL_GPIO_TogglePin   # Выключить
    li   a0, 200
    call HAL_DelayMs

    li   a0, 0x00084400
    li   a1, 4
    call HAL_GPIO_TogglePin   # Включить
    li   a0, 200
    call HAL_DelayMs
    li   a0, 0x00084400
    li   a1, 4
    call HAL_GPIO_TogglePin   # Выключить
    li   a0, 200
    call HAL_DelayMs

    li   a0, 0x00084400
    li   a1, 4
    call HAL_GPIO_TogglePin   # Включить
    li   a0, 200
    call HAL_DelayMs
    li   a0, 0x00084400
    li   a1, 4
    call HAL_GPIO_TogglePin   # Выключить
    li   a0, 200
    call HAL_DelayMs

    # Три длинных сигнала
```

```
    li   a0, 0x00084400
    li   a1, 4
    call HAL_GPIO_TogglePin   # Включить
    li   a0, 600              # Задержка 600 мс
    call HAL_DelayMs
    li   a0, 0x00084400
    li   a1, 4
    call HAL_GPIO_TogglePin   # Выключить
    li   a0, 200
    call HAL_DelayMs

    li   a0, 0x00084400
    li   a1, 4
    call HAL_GPIO_TogglePin   # Включить
    li   a0, 600
    call HAL_DelayMs
    li   a0, 0x00084400
    li   a1, 4
    call HAL_GPIO_TogglePin   # Выключить
    li   a0, 200
    call HAL_DelayMs

    li   a0, 0x00084400
    li   a1, 4
    call HAL_GPIO_TogglePin   # Включить
    li   a0, 600
    call HAL_DelayMs
    li   a0, 0x00084400
    li   a1, 4
    call HAL_GPIO_TogglePin   # Выключить
    li   a0, 200
    call HAL_DelayMs

    # Три коротких сигнала
    li   a0, 0x00084400
    li   a1, 4
    call HAL_GPIO_TogglePin   # Включить
    li   a0, 200
    call HAL_DelayMs
    li   a0, 0x00084400
```

```
    li   a1, 4
    call HAL_GPIO_TogglePin   # Выключить
    li   a0, 200
    call HAL_DelayMs

    li   a0, 0x00084400
    li   a1, 4
    call HAL_GPIO_TogglePin   # Включить
    li   a0, 200
    call HAL_DelayMs
    li   a0, 0x00084400
    li   a1, 4
    call HAL_GPIO_TogglePin   # Выключить
    li   a0, 200
    call HAL_DelayMs

    li   a0, 0x00084400
    li   a1, 4
    call HAL_GPIO_TogglePin   # Включить
    li   a0, 200
    call HAL_DelayMs
    li   a0, 0x00084400
    li   a1, 4
    call HAL_GPIO_TogglePin   # Выключить
    li   a0, 1000             # Пауза 1000 мс
    call HAL_DelayMs

    j    loop             # Повторяем цикл

# Функция настройки часов
.globl SystemClock_Config
.type SystemClock_Config, @function
SystemClock_Config:
    # Пролог: настройка стека
    addi sp, sp, -80       # Выделяем 80 байт на стеке
    sw   ra, 76(sp)        # Сохраняем адрес возврата
    sw   s0, 72(sp)        # Сохраняем регистр s0
    addi s0, sp, 80        # Устанавливаем s0 как указатель на базу стека

    # Инициализация структуры PCC (Peripheral Clock Control)
```

```
    li   t0, 15           # Источник тактирования (например, внешний осциллятор)
    sb   t0, -56(s0)      # Записываем в поле структуры
    li   t0, 2            # Делитель частоты
    sb   t0, -52(s0)      # Записываем в поле структуры
    sw   zero, -48(s0)    # Остальные поля обнуляем
    li   t0, 2            # Режим работы PLL
    sw   t0, -44(s0)      # Записываем в поле структуры
    sw   zero, -40(s0)    # Обнуляем остальные поля
    sw   zero, -36(s0)
    sw   zero, -32(s0)
    li   t0, -128         # Конфигурация тактирования (старший байт)
    sb   t0, -28(s0)
    li   t0, 8            # Множитель частоты
    sb   t0, -27(s0)
    sw   zero, -24(s0)    # Обнуляем оставшиеся поля
    sw   zero, -20(s0)

    # Вызов функции настройки часов
    addi a0, s0, -80      # Указатель на временную структуру
    addi a1, s0, -56      # Указатель на конфигурацию PCC
    call HAL_PCC_Config   # Применяем настройки часов

    # Эпилог: восстановление стека
    lw   ra, 76(sp)       # Восстанавливаем адрес возврата
    lw   s0, 72(sp)       # Восстанавливаем s0
    addi sp, sp, 80       # Освобождаем стек
    jr   ra               # Возвращаемся

# Функция инициализации GPIO
.globl GPIO_Init
.type GPIO_Init, @function


  GPIO_Init:
    # Пролог: настройка стека
    addi sp, sp, -32      # Выделяем 32 байта на стеке
    sw   ra, 28(sp)       # Сохраняем адрес возврата
    sw   s0, 24(sp)       # Сохраняем регистр s0
    addi s0, sp, 32       # Устанавливаем s0 как указатель на базу стека
```

```
    # Настройка режима порта
    li   t0, 327680       # Базовый адрес регистра GPIO (PORTA)
    li   t1, 4096         # Режим вывода (GPIO_MODE_OUTPUT)
    sw   t1, 28(t0)       # Устанавливаем режим для пина
    li   t1, 8192         # Альтернативная функция (если требуется)
    sw   t1, 28(t0)       # Записываем в тот же регистр

    # Инициализация структуры GPIO
    li   t0, 4          # Номер пина (GPIO_PIN_9)
    sw   t0, -32(s0)      # Записываем в поле структуры
    sw   zero, -28(s0)    # Обнуляем остальные поля
    sw   zero, -24(s0)
    sw   zero, -20(s0)

    # Вызов функции инициализации GPIO
    li   a0, 0x00084400       # Адрес порта GPIO (GPIOA_BASE)
    addi a1, s0, -32      # Указатель на структуру GPIO
    call HAL_GPIO_Init    # Инициализируем пин

    # Эпилог: восстановление стека
    lw   ra, 28(sp)       # Восстанавливаем адрес возврата
    lw   s0, 24(sp)       # Восстанавливаем s0
    addi sp, sp, 32       # Освобождаем стек
    jr   ra               # Возвращаемся
```

Оценивание лабораторной работы

Пункт	Баллы
Светодиод мигает в одном режиме (с паузами, но без соблюдения шаблона SOS); периферия настроена корректно.	6
Передача сигнала SOS реализована правильно (включая точки, тире и паузы).	2
Инициализированы часы и GPIO, используется HAL, реализован шаблон SOS, корректно обрабатываются все паузы.	2

Лабораторная работа 9

Пример возможной реализации программы

1. *multiply.S*

```
  .section .text
.global multiply_asm          # Объявление функции для доступа из C

multiply_asm:
    mul a0, a0, a1            # Умножение a0 на a1, результат в a0
    jr ra                     # Возврат к вызывающей программе
```

2. *call_c.S – Задание на оценку 8*

```
  .section .text
.global call_c_function       # Объявление функции для доступа из C

call_c_function:
    addi sp, sp, -16          # Выделение места в стеке
    sd ra, 8(sp)              # Сохранение адреса возврата
    jal print_message         # Вызов C-функции print_message
    ld ra, 8(sp)              # Восстановление адреса возврата
    addi sp, sp, 16           # Освобождение стека
    jr ra                     # Возврат
```

3. *main.c*

```
  #include <stdio.h>
extern int multiply_asm(int a, int b);    // Объявление ассемблерной функции
extern void call_c_function(void);        // Объявление функции для вызова C

void print_message(void) {
    printf("Message from C function!\n"); // Функция, вызываемая из ассемблера
}

int main() {
    int a = 4, b = 5;
    int result = multiply_asm(a, b);      // Вызов ассемблерной функции
```

```
    printf("Product of %d and %d is: %d\n", a, b, result);
    call_c_function();                    // Вызов ассемблера, который вызывает C
    return 0;
}
```

4. *build.sh – скрипт для компиляции*

```
  #!/bin/bash
# Сборка ассемблерных файлов
riscv64-linux-gnu-as -o multiply.o multiply.s
riscv64-linux-gnu-as -o call_c.o call_c.s

# Компиляция C-файла
riscv64-linux-gnu-gcc -c -o main.o main.c

# Компоновка
riscv64-linux-gnu-gcc -o program main.o multiply.o call_c.o

# Запуск в QEMU
qemu-riscv64 ./program
```

5. *extension.c*

```
  #include <Python.h>
extern int multiply_asm(int a, int b);

static PyObject* py_multiply_asm(PyObject* self, PyObject* args) {
    int a, b;
    if (!PyArg_ParseTuple(args, "ii", &a, &b)) {
        return NULL;
    }
    int result = multiply_asm(a, b);
    return PyLong_FromLong(result);
}

static PyMethodDef methods[] = {
    {"multiply_asm", py_multiply_asm, METH_VARARGS, "Multiply two integers using
assembly"},
```

```
    {NULL, NULL, 0, NULL}
};

static struct PyModuleDef module = {
    PyModuleDef_HEAD_INIT, "myasm", NULL, -1, methods
};

PyMODINIT_FUNC PyInit_myasm(void) {
    return PyModule_Create(&module);
}
```

6. *test.py – задание на оценку 10*

```
  import myasm
result = myasm.multiply_asm(4, 5)
print(f"Result from assembly: {result}")
```

7. *build_python.sh – скрипт компиляции кода*

```
  #!/bin/bash
# Компиляция C-расширения
riscv64-linux-gnu-gcc -shared -fPIC -o myasm.so extension.c multiply.o -
I/usr/include/python3.12 -L/usr/lib -lpython3.12

# Запуск Python-скрипта. Опция -B выполняет сдвиг адресов. В случае ошибки
компиляции, попробуйте изменить это значение
qemu-riscv64 -B 0x100000000 -L /usr/riscv64-linux-gnu /usr/bin/python3 test.py
```

Оценивание лабораторной работы

Пункт	Баллы
Реализована ассемблерная функция, корректно вычисляющая произведение двух чисел. Функция вызывается из программы на C, результат возвращается в программу, но вывод в консоль не требуется.	6
В дополнение к требованиям для 6 баллов, реализован вызов функции на C, выводящей текстовое сообщение, из ассемблерного кода. Программа демонстрирует двустороннее взаимодействие между C и ассемблером с выводом результата в консоль.	2
Выполнены все указанные выше требования, а также реализована интеграция с Python через C-расширение. Ассемблерная функция вызывается из Python, результат выводится в консоль интерпретатора Python.	2

Лабораторная работа 10

Пример возможной реализации программы

```
   .data
array: .dword 2, 4, 6, 8, 10  # Массив из 5 чисел (64-битные слова для RV64)
size: .dword 5                # Размер массива
multiplier: .dword 3          # Коэффициент умножения
buffer: .space 20             # Буфер для строки числа
newline: .asciz "\n"          # Символ новой строки

.text
.global main
main:
    # Сохраняем регистр возврата
    addi sp, sp, -16
    sd ra, 8(sp)

    # Инициализация
    la t0, array              # Указатель на начало массива
    ld t1, size               # Загрузка размера массива (64 бита)
    ld t2, multiplier         # Загрузка коэффициента умножения (64 бита)
    li t3, 0                  # Сумма
    li t4, 0                  # Счетчик цикла

    # Проверка на нулевой размер
    beqz t1, error_size_zero
```

```
multiply_loop:
    bge t4, t1, calculate_average  # Если обработали все элементы, переходим к
вычислению среднего
    ld t5, 0(t0)                   # Загружаем текущий элемент массива (64 бита)
    mul t6, t5, t2                 # Умножаем элемент на коэффициент
    add t3, t3, t6                 # Добавляем результат к сумме
    addi t0, t0, 8                 # Сдвигаем указатель на следующий элемент (8
байт для RV64)
    addi t4, t4, 1                 # Увеличиваем счетчик
    j multiply_loop

calculate_average:
    div a0, t3, t1                 # Делим сумму на количество элементов
    jal number_to_string           # Преобразуем результат в строку
    jal print_string               # Выводим результат
    j end

error_size_zero:
    li a0, -1                      # Код ошибки: деление на ноль
    jal number_to_string           # Преобразуем -1 в строку
    jal print_string               # Выводим ошибку
    j end

# Преобразование числа в a0 в строку в buffer
number_to_string:
    la t0, buffer                # Указатель на буфер
    addi t0, t0, 19              # Конец буфера (записываем справа налево)
    sb zero, 0(t0)               # Завершающий нуль
    mv t1, a0                    # Число для преобразования
    li t2, 10                    # Основание системы счисления
    bgez t1, convert_loop        # Если число неотрицательное, пропускаем знак
    li t3, '-'                   # Знак минус
    sb t3, -1(t0)                # Записываем минус
    addi t0, t0, -1
    neg t1, t1                   # Преобразуем в положительное

convert_loop:
    remu t3, t1, t2              # Остаток от деления на 10
    divu t1, t1, t2              # Делим на 10
```

```
    addi t3, t3, '0'            # Преобразуем цифру в символ
    addi t0, t0, -1             # Сдвигаем указатель
    sb t3, 0(t0)                # Записываем символ
    bnez t1, convert_loop       # Продолжаем, если число не 0
    mv a0, t0                   # Указатель на начало строки
    ret

# Вывод строки из a0
print_string:
    mv t0, a0                   # Сохраняем указатель на строку
    li t1, 0                    # Длина строки


  strlen_loop:
    lb t2, 0(t0)                # Загружаем байт
    beqz t2, strlen_done        # Если нуль, конец строки
    addi t0, t0, 1              # Следующий байт
    addi t1, t1, 1              # Увеличиваем длину
    j strlen_loop


  strlen_done:
    mv a2, t1                   # Длина строки
    mv a1, a0                   # Указатель на начало строки
    li a0, 1                    # Дескриптор stdout
    li a7, 64                   # Системный вызов write
    ecall                       # Вывод строки
    # Вывод новой строки
    li a0, 1                    # Дескриптор stdout
    la a1, newline              # Указатель на "\n"
    li a2, 1                    # Длина строки
    li a7, 64                   # Системный вызов write
    ecall
    ret

end:
    # Восстанавливаем регистр возврата
    ld ra, 8(sp)
    addi sp, sp, 16
    li a0, 0                    # Код возврата 0
```

```
    ret                         # Возврат для libc
```

Оценивание лабораторной работы

Пункт	Баллы
Умножения элементов, вычисление суммы и среднего значения реализованы корректно.	6
Добавлена обработка деления на ноль.	2
Выполнен вывод результата в консоль.	2

Лабораторная работа 11

Пример возможной реализации программы

```
  .section .data
.align 8                    # Выравнивание на 8 байт для double
a:          .double 1.0     # Число a
b:          .double 2.0     # Число b
c:          .double 3.0     # Число c
ref:        .double 7.0     # Ожидаемое значение (1 + 2 * 3 = 7)
eps:        .double 1e-10   # Epsilon для проверки
ok_msg:     .asciz "OK\n"   # Сообщение для верного результата
notok_msg:  .asciz "NOT OK\n" # Сообщение для неверного результата

.section .text
.global _start
_start:
    # Загрузка чисел из памяти
    la t0, a                # t0 = адрес a
    fld f0, 0(t0)           # f0 = a
    la t0, b
    fld f1, 0(t0)           # f1 = b
    la t0, c
    fld f2, 0(t0)           # f2 = c
    la t0, ref
    fld f6, 0(t0)           # f6 = ref
    la t0, eps
    fld f5, 0(t0)           # f5 = eps

    # Вычисление a + b * c
```

```
    fmul.d f4, f1, f2       # f4 = b * c
    fadd.d f3, f0, f4       # f3 = a + b * c

    # Вычисление |f3 - ref|
    fsub.d f4, f3, f6       # f4 = f3 - ref
    fabs.d f4, f4           # f4 = |f3 - ref|

    # Проверка |f3 - ref| < eps
    flt.d t1, f4, f5        # t1 = 1, если |f3 - ref| < eps, иначе 0

    # Вывод результата
    beq t1, zero, print_not_ok

print_ok:
    li a7, 64               # Системный вызов write (64 на RISC-V, 4 – на x86)
    li a0, 1                # Дескриптор файла (stdout = 1)
    la a1, ok_msg           # Адрес строки OK
    li a2, 3                # Длина строки OK\n
    ecall
    j exit

print_not_ok:
    li a7, 64               # Системный вызов write
    li a0, 1                # Дескриптор файла (stdout = 1)
    la a1, notok_msg        # Адрес строки NOT OK
    li a2, 8                # Длина строки NOT OK\n
    ecall

exit:
    li a7, 93               # Системный вызов exit (93 вместо 10 для QEMU)
    li a0, 0                # Код возврата 0
    ecall
```

Оценивание лабораторной работы

Пункт	Баллы
Реализованы загрузка чисел из памяти, вычисление выражения a + b × c, вычисление абсолютной разницы.	6
Добавлено сравнение с epsilon.	2
Реализован вывод результата.	2

Лабораторная работа 12

Пример возможной реализации программы

```
  # Оптимизированная версия: суммирование квадратов с разворачиванием цикла
.section .data
array: .space 4000000  # 1,000,000 элементов * 4 байта
array_end: .quad array + 4000000
.section .text
.global main


  main:
    # Инициализация массива: array[i] = i
    la t0, array
    li t1, 0          # i = 0
    li t2, 1000000    # размер массива


  init_loop:
    bge t1, t2, init_end
    sw t1, 0(t0)      # array[i] = i
    addi t0, t0, 4
    addi t1, t1, 1
    j init_loop


  init_end:
    # Внешний цикл: повторить суммирование 100 раз
    li t3, 0          # счетчик внешнего цикла
    li t4, 100        # количество итераций


  outer_loop:
```

```
    bge t3, t4, outer_end
    # Суммирование квадратов с разворачиванием на 4 элемента
    la t0, array
    la t1, array_end
    li t2, 0          # сумма = 0

sum_loop_unrolled:
    bge t0, t1, sum_end
    lw t5, 0(t0)      # первый элемент
    mul t5, t5, t5
    add t2, t2, t5
    lw t5, 4(t0)      # второй элемент
    mul t5, t5, t5
    add t2, t2, t5
    lw t5, 8(t0)      # третий элемент
    mul t5, t5, t5
    add t2, t2, t5
    lw t5, 12(t0)     # четвертый элемент
    mul t5, t5, t5
    add t2, t2, t5
    addi t0, t0, 16   # сдвиг на 4 элемента (16 байт)
    j sum_loop_unrolled

sum_end:
    addi t3, t3, 1    # увеличить счетчик внешнего цикла
    j outer_loop

outer_end:
    mv a0, t2         # последняя сумма в a0
    li a7, 93         # системный вызов для выхода
    ecall
```

Разница в скорости работы программ следующая:

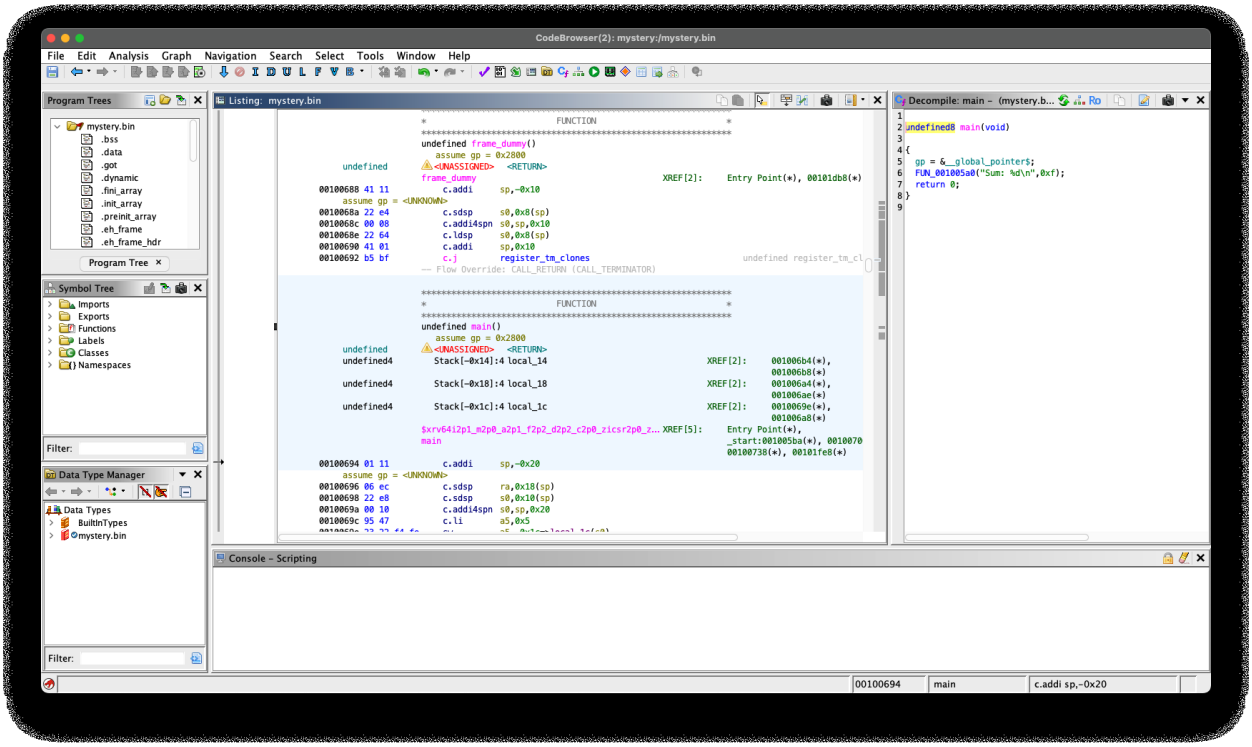
Неоптимизированная версия	Оптимизированная версия
real 0m0,215s user 0m0,209s sys 0m0,006s	real 0m0,115s user 0m0,112s sys 0m0,003s

Оценивание лабораторной работы

Пункт	Баллы
Применены методы оптимизации, но время работы программы сокращено менее чем на 30%	4
Применение оптимизаций, таких как использование регистров и минимизация инструкций, в результате чего достигнуто сокращение времени работы программы более чем на 30%	6

Лабораторная работа 13

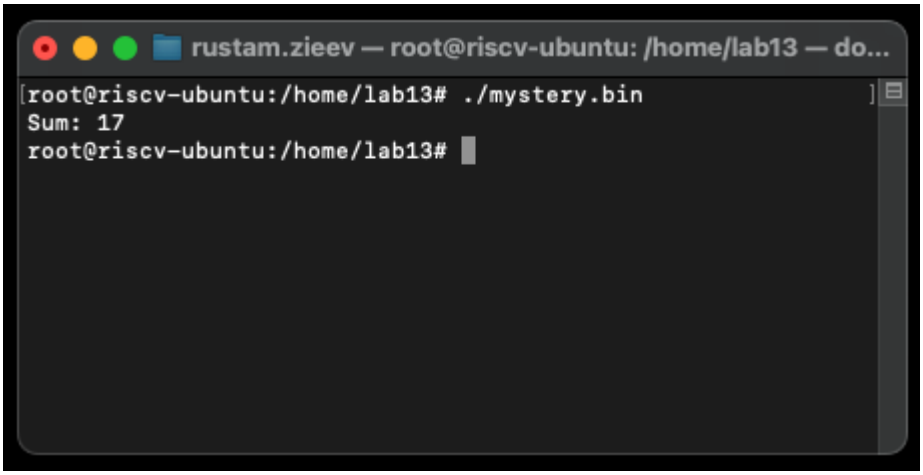
Пример возможной реализации программы



Файл mystery.bin был загружен в утилиту Ghidra, найдена функция main по адресу 0x00100694. В ней программа выделяет место в стеке, чтобы сохранить два числа: 5 и 10. Число 5 записывается командой c.li a5, 0x5, а 10 – командой c.li a5, 0xa. Потом эти числа загружаются из памяти с помощью lw (load word), суммируются с помощью команды c.addw, и результат сохраняется. Затем программа вызывает printf с помощью команды jalr, передавая строку "Sum: %d\n" и результат сложения. Программа использует auipc для загрузки адресов, чтобы быть независимой от конкретного места в памяти.

Программа была запущена командой qemu-riscv64 ./mystery.bin, и вывела: Sum: 15. Это подтверждает, что программа складывает 5 и 10 и выводит результат в консоль.

В Ghidra была найдена инструкция c.li a5, 0x5 и заменена на c.li a5, 0x7. После запуска измененного бинарного файла, программа вывела Sum: 17, потому что 7 + 10 = 17, следовательно, изменения в Ghidra непосредственно влияют на работу программы.



Оценивание лабораторной работы

Пункт	Баллы
Продемонстрирован запуск утилиты Ghidra, выполнен базовый анализ кода программы и определено, что делает программа.	6
Определены инструкции RISC-V, использованные в программе. Бинарный файл запущен, гипотеза подтверждена с помощью вывода программы.	2
Выполнены манипуляции с исходным кодом в Ghidra, приведены результаты анализа.	2

Лабораторная работа 14

Пример возможной реализации программы

1. Уязвимая версия кода

```
  .section .text
.global main

main:
    # Пролог функции
    addi sp, sp, -32
    sw ra, 28(sp)    # Сохранение адреса возврата
    sw s0, 24(sp)    # Сохранение s0
```

```
    # Выделить буфер на стеке (8 байт)
    addi s0, sp, 16  # s0 указывает на начало буфера (смещение 16 байт)

    # Вывести приглашение для ввода
    la a0, prompt
    call puts

    # Читать ввод в буфер с помощью read (без проверки длины)
    li a7, 63        # Системный вызов read
    li a0, 0         # Файловый дескриптор stdin
    mv a1, s0        # Адрес буфера
    li a2, 32        # Максимальная длина (чтобы гарантировать переполнение)
    ecall

    # Вывести введенные данные
    mv a0, s0
    call puts

    # Эпилог функции
    lw s0, 24(sp)
    lw ra, 28(sp)
    addi sp, sp, 32
    # Выход из программы
    li a7, 93        # Системный вызов exit
    li a0, 0         # Код возврата 0
    ecall

.section .rodata
prompt: .string "Enter input: "

# Внешние функции из libc
.global puts
```

2. Защищенная версия кода

```
  .section .text
.global main

main:
```

```
# Пролог функции
addi sp, sp, -40    # Выделение места для канарейки (4 байта) и буфера (8 байт)
sw ra, 36(sp)       # Сохранение адреса возврата
sw s0, 32(sp)       # Сохранение s0

# Установка канарейки
li t0, 0xDEADBEEF   # Фиксированное значение канарейки
sw t0, 32(sp)       # Сохранение канарейки сразу после s0

# Выделить буфер на стеке (8 байт)
mv s0, sp           # s0 указывает на начало буфера (смещение 0 от текущего sp)

# Вывести приглашение для ввода
la a0, prompt
call puts

# Читать ввод в буфер с помощью read (ограничим длину до 8 байт)
li a7, 63           # Системный вызов read
li a0, 0            # Файловый дескриптор stdin
mv a1, s0           # Адрес буфера
li a2, 8            # Максимальная длина, равная размеру буфера
ecall

# Вывести введенные данные
mv a0, s0
call puts

# Проверка канарейки
lw t0, 32(sp)       # Загрузка сохраненной канарейки
li t1, 0xDEADBEEF   # Ожидаемое значение канарейки
bne t0, t1, error   # Если канарейки не совпадают, переход к ошибке

# Эпилог функции
lw s0, 32(sp)
lw ra, 36(sp)
addi sp, sp, 40
# Выход из программы
li a7, 93           # Системный вызов exit
li a0, 0            # Код возврата 0
ecall
```

```
error:
    # Завершение с ошибкой при обнаружении атаки
    la a0, error_msg
    call puts
    # Явное прерывание для сброса ввода (используем бесконечный цикл)
    infinite_loop:
        j infinite_loop

.section .rodata
prompt: .string "Enter input: "
error_msg: .string "Buffer overflow detected!"

# Внешние функции из libc
.global puts
```

Оценивание лабораторной работы

Пункт	Баллы
Проведен анализ функции, выявлена уязвимость переполнения буфера, описан механизм ее эксплуатации для перезаписи адреса возврата. Проведено тестирование исходной программы в QEMU с демонстрацией переполнения буфера. В ассемблерный код добавлен механизм канарейки стека с инструкциями для установки и проверки ее значения.	8
Модифицированный код скомпилирован и проведено его тестирование в QEMU. Показано, что канарейка стека предотвращает атаку с использованием переполнения буфера.	2

Приложение 2

Ответы на вопросы к главе 1

1. Флаг -nostartfiles нужен, если в программе определяется метка _start. Без него произойдёт ошибка линковки из-за переопределения системной метки. Флаг отключает подключение стандартных стартовых файлов.
2. Кросс-компиляция — это сборка исполняемого файла для архитектуры, отличной от архитектуры хост-системы. Она необходима, когда нужно собрать RISC-V программу на компьютере без RISC-V процессора. Для этого используется пакет кросс-компилятора, например riscv-none-elf-gcc-xpack.
3. RARS — это Java-приложение (файл формата .jar). Для его запуска необходима среда выполнения Java Runtime Environment. Без неё файл .jar не откроется и утилита не заработает.
4. QEMU позволяет эмулировать процессор RISC-V на обычном компьютере без необходимости иметь реальное железо. Это удобно для отладки и изучения ассемблера. Реальный микроконтроллер требует дополнительных библиотек, скриптов линковки и процедуры прошивки.
5. mik32-hal содержит библиотеки для работы с периферийными блоками микроконтроллера. mik32v2-shared включает заголовочные файлы, стартовые скрипты и скрипты линковки. Без них программа не сможет корректно собраться и запуститься на MIK32 Амур.
6. Для сборки используется команда: riscv-none-elf-gcc -nostartfiles example.S -o example.elf. Флаг -nostartfiles обязателен при наличии метки _start.
7. Размер образа увеличивают командой: sudo qemu-img resize ubuntu-24.04.1-server-riscv64.img +10G. Это добавляет 10 ГБ к доступному хранилищу виртуальной машины.
8. Подключение выполняется командой: ssh -p 2222 localhost. Предварительно в виртуальной машине нужно установить и запустить openssh-server.
9. На Linux перед сборкой контейнеров под riscv64 выполняют: docker run --privileged --rm tonistiigi/binfmt --install all. Это настраивает Docker для работы с эмуляцией QEMU.
10. Прошивка микроконтроллера выполняется с помощью цели upload в Makefile. Для этого нужно вызвать команду make upload. Эта цель запускает скрипт mik32_upload.py через openOCD.

Ответы на вопросы к главе 2

11. RISC-V – это «load/store» архитектура: все арифметические и логические операции работают только над регистрами, а инструкции загрузки (lw, lb) и сохранения (sw, sb) – только с памятью.
12. В RISC-V (как и в большинстве современных архитектур) целые значения хранятся в дополнительном коде. В этом формате бит-знака (старший бит слова) автоматически расширяется при выполнении арифметических операций и загрузке регистров со знаковым расширением. Отрицательное число –N–N–N кодируется как «инверсия битов числа NNN плюс единица». Это позволяет инструкциям add, sub корректно вычислять результат с учетом знаков аргументов.
13. Для умножения на 2 используется логический сдвиг влево на единицу:

```
slli rd, rs, 1   # rd = rs << 1
```

1. add складывает два регистра (add t0, t1, t2), addi – складывает регистр с константой (addi t0, t1, 42)
2. Различия slli, srai и srli:
   - slli – логический сдвиг влево (все биты сдвигаются, справа нули):
     ```
     slli t0, t1, 2   # t0 = t1 << 2 (умножение на 4).
     ```
   - srai – арифметический сдвиг вправо (сохраняет знак):
     ```
     srai t0, t1, 3   # t0 = t1 >>> 3 (деление на 8 с сохранением знака)
     ```
   - srli – логический сдвиг вправо (справа нули, знак теряется):
     ```
     srli t0, t1, 1   # t0 = t1 >> 1 (беззнаковое деление на 2)
     ```
3. mv – псевдоинструкция для копирования регистров (транслируется в addi):
   ```
   mv t0, t1   # Эквивалентно: addi t0, t1, 0
   ```

Ответы на вопросы к главе 3

Глава 3.1

1. Makefile – это файл с инструкциями для утилиты make, который описывает зависимости между файлами и команды для их обработки. Используется для автоматизации сборки проектов, перекомпиляции только измененных файлов и управления сложными процессами сборки.
2. Правило состоит из:
   - цели: файл или действие, которое нужно создать/выполнить;
   - зависимостей: файлы, от которых зависит цель;
   - команд: действия для сборки цели.

   Пример:

```
main.o: main.c
	gcc -c main.c -o main.o
```

3. wildcard находит все файлы «.c» в директории «src/» и сохраняет их в SOURCES. patsubst заменяет расширения «.c» на «.o» для каждого файла в SOURCES, создавая список объектных файлов в OBJECTS.
4. Значения переменных:
   - $@ – имя цели;
   - $< – первая зависимость;
   - $^ – все зависимости через пробел.

   Пример:

```
%.o: %.c
	$(CC) -c $< -o $@
```

где: $< – исходный «.c»-файл, $@ – целевой «.o»-файл.

5. Проверка наличия файла в Makefile через $(wildcard) или shell:

```
ifeq ($(wildcard config.h),)
	$(error "config.h not found!")
endif
```

6. Это шаблонное правило для компиляции любого «.c»-файла в соответствующий «.o»-файл:
   - «%.o» – цель (объектный файл):
   - «%.c» – зависимость (исходный код),
   - автоматические переменные «$<» и «$@» подставляют имена файлов.

Глава 3.2

1. a. Она останавливает программу, когда меняются значения переменных.
2. b. `next`.
3. c. `print x`.
4. Данная команда выводит стек вызовов программы.
5. Подключается к удаленному серверу по адресу localhost:1234, устанавливает точку останова по адресу 0x400256, продолжает выполнение кода до этой точки останова, выполняет следующую инструкцию.

Ответы на вопросы к главе 4

1. Безусловный переход — это инструкция, при выполнении которой программа всегда передает управление другой части кода, независимо от каких-либо условий. В языке RISC-V такие переходы реализуются с помощью инструкций j (прыжок по метке) и jal (прыжок с сохранением адреса возврата). Безусловные переходы применяются, например, для перехода в начало цикла, перехода к завершению программы или вызова подпрограмм.
2. Условные переходы в RISC-V осуществляются при помощи инструкций beq (переход, если равны), bne (если не равны), blt (меньше), bge (больше или равно), bltu, bgeu (беззнаковые аналоги). Эти инструкции сравнивают значения в двух регистрах и выполняют переход к указанной метке, если условие истинно. Если условие ложно, программа продолжает исполнение по следующей инструкции.
3. Циклы в RISC-V реализуются вручную с использованием меток и условных или безусловных переходов. Типичная структура включает:
   - инициализацию счетчика;
   - метку начала цикла;
   - тело цикла;
   - изменение счетчика;

– условный переход к началу цикла при выполнении условия.

Например, цикл for можно реализовать с помощью addi для изменения счетчика и blt или bge для управления переходами.

4. Различия j и jal:
   – j label – безусловный переход (псевдоинструкция для jal zero, label).
   – jal label – переход и сохранение адреса возврата в ra (для вызовов функций).
5. Пример реализации цикла while (i < 10):

```
li   t0, 0           # i = 0
li   t1, 10          # Верхняя граница
loop:
    bge  t0, t1, end # Если i >= 10 → выйти
    # ... тело цикла ...
    addi t0, t0, 1   # i++
j    loop
end:
```

6. ASCII в число: вычесть 0x30 (или 48):

```
lb   t0, char      # Загрузка символа '7' (0x37)
addi t0, t0, -48   # t0 = 7
```

Число в ASCII: прибавить 0x30:

```
addi t0, t0, 48    # t0 = '7'
sb   t0, buffer    # Сохраняем в память
```

Ответы на вопросы к главе 5

1. Секции памяти и их назначение:
   – .data — секция для инициализированных глобальных переменных (имеют начальное значение).
   – .bss — секция для неинициализированных глобальных переменных (будут заполнены во время выполнения).
   – .text — секция с исполняемым кодом программы.
2. .word выделяет 4 байта (32 бита) памяти для одного значения, а .byte — только 1 байт (8 бит). Они используются для размещения переменных разного размера.
3. .align выравнивает адрес следующего элемента в памяти до степени двойки. Например, .align 2 выравнивает адрес до 4 байт (2^2), что важно для корректной и быстрой загрузки 32-битных данных.
4. Если переменная размещена не по выровненному адресу, это может привести к ошибке исполнения (в некоторых средах) или снижению производительности из-за необходимости обращения к невыровненным адресам.

5. Все переменные в секции .bss автоматически инициализируются нулями во время запуска программы. Это позволяет объявлять переменные без начального значения, экономя место в бинарном файле.
6. Little-Endian – это порядок байтов, при котором младший байт хранится по младшему адресу. Такой порядок используется, в первую очередь, для упрощения и ускорения операций преобразования типов данных: например, для приведения числа к 8-битному представлению достаточно просто взять первые 8 бит числа.

Ответы на вопросы к главе 6

1. Стек используется для временного хранения информации при вызове функций. Он позволяет сохранять значения регистров, аргументы, локальные переменные и адрес возврата, что необходимо для корректного выполнения вложенных вызовов и рекурсии. Стек растет в сторону уменьшения адресов, управление им осуществляется через регистр sp.
2. В RISC-V используются регистры аргументов a0-a7, временные регистры t0-t6, сохраняемые регистры s0-s11 и регистр возврата ra. Регистры a0-a7 применяются для передачи параметров и возврата значений. Регистры t0-t6 могут свободно использоваться без сохранения. Регистры s0-s11 должны быть сохранены вызываемой функцией, если они используются. Регистр ra хранит адрес возврата после вызова функции.
3. Регистр ra используется для хранения адреса возврата при вызове функции. Он автоматически заполняется инструкцией перехода с сохранением возврата. Если внутри функции происходит вложенный вызов другой функции, значение ra должно быть сохранено в стек, иначе оно будет перезаписано и возврат в исходную точку станет невозможен.
4. Стековый кадр представляет собой часть стека, выделенную при входе в функцию для хранения ее локальных данных и сохраненных регистров. Он формируется путем уменьшения значения указателя стека и используется для изоляции контекста выполнения функции. После завершения работы кадр освобождается путем увеличения значения регистра sp.
5. Рекурсия применяется при разбиении задачи на более простые подзадачи одного типа, например, факториал или последовательность Фибоначчи. На уровне ассемблера рекурсия реализуется посредством вызова функции самой себя с новыми аргументами. При этом требуется организовать стек для хранения всех промежуточных значений и адресов возврата, иначе программа будет работать некорректно или приведет к повреждению данных.
6. Это стандартный подход в большинстве архитектур. При выделении места в стеке (addi sp, sp, -N) указатель перемещается к младшим адресам, что позволяет:
   - легко выделять/освобождать блоки памяти;
   - избегать коллизий с кучей, которая растет вверх;
   - соответствовать общепринятым соглашениям.
7. Произойдет нарушение соглашения о вызовах. Вызывающая функция может полагаться на сохранность этих регистров, поэтому, если не сохранить регистры s0-11 в стек:

- могут быть испорчены данные вызывающей функции,
- программа начнет вести себя непредсказуемо,
- отсутствие сохраненных регистров особенно критично для рекурсивных вызовов.

Ответы на вопросы к главе 7

1. Системный вызов – это механизм, с помощью которого пользовательская программа запрашивает выполнение определенных действий от ядра операционной системы. Это единственный безопасный способ доступа к ресурсам ядра, таким как файловая система, ввод-вывод, управление процессами и памятью.
2. В архитектуре RISC-V для выполнения системных вызовов используется инструкция ecall.
3. Перед вызовом ecall параметры передаются в регистры a0-a6 (до семи параметров), а номер системного вызова указывается в регистре a7. После выполнения системного вызова результат возвращается в регистр a0.
4. Наиболее часто используемыми системными вызовами для работы с файлами являются: openat (номер 56) для открытия, read (номер 63) для чтения, write (номер 64) для записи и close (номер 57) для закрытия файла.
5. После выполнения системного вызова результат возвращается в регистр a0. Если результат положительный или равен нулю, вызов считается успешным. Отрицательные значения означают ошибку, и их необходимо обрабатывать (коды ошибок соответствуют определениям в файле errno.h).
6. Для указания текущей директории в системном вызове openat используется файловый дескриптор директории со значением -100 (AT_FDCWD).
7. После завершения работы с файлом его необходимо закрыть с помощью вызова close, указав соответствующий дескриптор.

Ответы на вопросы к главе 8

1. GPIO (General Purpose I/O) — это контакты общего назначения, с помощью которых пользовательская программа может взаимодействовать с внешними электронными компонентами, такими как светодиоды, кнопки, реле и датчики. Каждый GPIO может быть сконфигурирован как вход или выход.
2. Наиболее распространенный способ работы с GPIO в Linux – использование файловой системы /sys/class/gpio. Управление осуществляется путем записи в специальные файлы: export – для активации контакта, direction – для задания направления (in или out), value – для чтения или записи значения. Эти операции выполняются с помощью системных вызовов open, write и read.
3. mmap – это функция, используемая для прямого доступа к памяти, которая отображает физические адреса регистров в адресное пространство программы. Такой доступ обеспечивает высокую скорость работы и используется в задачах, чувствительных к задержкам.
4. Различия /sys/class/gpio и /dev/mem:
   - /sys/class/gpio: является стандартным подходом для управления оборудованием, использующим драйверы устройств Linux. Он обеспечивает более высокий уровень абстракции и безопасности. Операции через него обычно медленнее.

- /dev/mem: позволяет напрямую обращаться к физической памяти, отображая ее адреса в виртуальное пространство программы с помощью mmap. Этот метод быстрее, но требует повышенных привилегий (root-доступ) и детального знания архитектуры конкретного микроконтроллера или процессора, так как при обращении к памяти происходит обход встроенных средств безопасности драйвера.

5. Прямой доступ к физической памяти и аппаратным регистрам считается опасным и нежелательным, так как он обходит стандартные механизмы безопасности Linux, предоставляемые драйверами устройств. Поэтому для такого доступа необходимы полномочия суперпользователя (root-доступ).
6. Для управления отдельными битами используются логические инструкции: sll (логический сдвиг влево), srl (логический сдвиг вправо), and (побитовое И), or (побитовое ИЛИ), xor (побитовое исключающее ИЛИ).
7. Для создания задержек в Linux используется системный вызов nanosleep.

Ответы на вопросы к главе 9

1. Для вызова функций, разработанных на языке C, из ассемблерного кода RISC-V, параметры передаются через регистры a0-a7.
2. Чтобы сделать ассемблерную функцию доступной для вызова из программы на C, ее необходимо объявить в коде C с использованием ключевого слова extern (например, extern int myfunction(int, int);). Также важно, чтобы имена подпрограмм были уникальными, чтобы избежать конфликтов с библиотеками, и чтобы ассемблерная функция была глобальной (с использованием директивы .global в ассемблерном коде).
3. Статическая библиотека — это архивный файл (в Linux с расширением .a), который содержит скомпилированный машинный код (объектные файлы) подпрограмм. При компоновке программы, использующей статическую библиотеку, код библиотеки копируется непосредственно в исполняемый файл. Это делает исполняемый файл самодостаточным, но увеличивает его размер. Для создания таких библиотек используется утилита ar.
4. Ассемблерная функция объявляется в коде на C с использованием ключевого слова extern, указывающего ее прототип (имя, параметры и тип возвращаемого значения), например: extern int myfunction(char *, char *);.
5. Для встроенных ассемблерных вставок (inline assembly) в коде C используется ключевое слово asm.
6. Результаты выполнения функции из ассемблерного кода передаются обратно в код на C через регистр a0.

Ответы на вопросы к главе 10

1. RISC-V поддерживает следующие инструкции умножения:
   - mul rd, rs1, rs2 – выполняет знаковое умножение rs1 на rs2 и сохраняет младшие 32 или 64 бита результата (в зависимости от разрядности архитектуры) в регистр rd.
   - mulh rd, rs1, rs2 – возвращает старшие биты результата знакового умножения.
   - mulhu rd, rs1, rs2 – возвращает старшие биты результата беззнакового умножения.

- mulhsu rd, rs1, rs2 – выполняет умножение знакового числа из rs1 на беззнаковое из rs2, сохраняя старшие биты результата.
- mulw rd, rs1, rs2 – 32-битное умножение для 64-битных процессоров, которое выдает только 32-битное произведение.

Различия заключаются в том, что mul и mulw дают младшую часть произведения, в то время как mulh, mulhu и mulhsu предоставляют старшую часть. Для получения полного результата умножения чисел, превышающих размер одного регистра, необходимо использовать комбинацию инструкций (например, mul и mulh для знаковых чисел).

2. Различия между div и divu:
   - div rd, rs1, rs2 – выполняет знаковое деление rs1 на rs2, записывая частное в rd.
   - divu rd, rs1, rs2 – выполняет беззнаковое деление. Результаты отличаются в зависимости от того, как обрабатываются знаковые и беззнаковые числа, особенно при работе с отрицательными значениями.
3. В архитектуре RISC-V деление на ноль не приводит к исключению. Вместо этого процессор возвращает фиксированный результат (например, -1 при знаковом делении) и продолжает работу. Программист должен самостоятельно проверять, является ли делитель нулем, если требуется особая обработка ошибки.
4. Переполнение при делении в RISC-V возникает в одном специфическом случае: когда наиболее отрицательное знаковое целое число делится на -1. Подобно делению на ноль, это не вызывает исключения, а возвращает фиксированный результат.
5. При знаковом делении на ноль инструкция div вернет результат -1.
6. Инструкция mulh (multiply high) необходима для получения старших битов результата знакового умножения. Поскольку произведение двух N-битных чисел может быть до 2N бит, mulh позволяет получить верхнюю часть этого произведения, в то время как mul возвращает нижнюю часть, что необходимо для полного результата, когда он превышает размер одного регистра.
7. Полный результат умножения чисел, превышающих размер регистра, можно получить, комбинируя инструкцию mul для младших битов с mulh (для знаковых чисел) или mulhu (для беззнаковых чисел) для старших битов. Спецификация RISC-V позволяет оптимизировать этот процесс: если mulh (или mulhu) идет непосредственно перед mul с теми же исходными регистрами, выполняется только одно умножение.

Ответы на вопросы к главе 11

1. Число с плавающей точкой в формате IEEE-754 (double, 64 бита) состоит из трех частей:
   - 1 бит знака определяет положительное (0) или отрицательное (1) число.
   - 11 бит экспоненты задают, насколько нужно сдвинуть мантиссу.
   - 52 бита мантиссы (дробной части) описывают значимые цифры числа. Такой формат позволяет представлять очень большие или очень маленькие числа, а также специальные значения, такие как бесконечность или "не число" (NaN).
2. Числа с плавающей точкой нельзя сравнивать на точное равенство из-за ошибок округления, которые могут возникать в процессе вычислений из-за ограниченной точности их представления. Правильный способ сравнения – проверять, насколько они

близки друг к другу, то есть вычислять их абсолютную разность и сравнивать ее с небольшим заранее заданным значением, называемым эпсилоном (epsilon). Например, abs(S1 - S2) < e.

3. Для загрузки чисел с плавающей точкой из памяти в регистры используются команды flw (для одинарной точности, 32 бита) и fld (для двойной точности, 64 бита). Для сохранения чисел из регистров обратно в память используются команды fsw (для одинарной точности) и fsd (для двойной точности).
4. Регистр fcsr (Floating-Point Control and Status Register) – это регистр управления и статуса для операций с плавающей точкой. Он хранит информацию о накопленных исключениях (таких как недопустимая операция, деление на ноль, переполнение, неточность) и управляет режимом округления чисел во время вычислений.
5. Различия fcvt.d.w и fmv.d.x?
   - fcvt.d.w – это инструкция конвертации типа, которая преобразует целочисленное значение из целочисленного регистра в число с плавающей точкой двойной точности в регистре для чисел с плавающей точкой. То есть, она меняет числовое представление.
   - fmv.d.x – это инструкция перемещения битов, которая копирует сырой битовый шаблон из целочисленного регистра в регистр для чисел с плавающей точкой (для двойной точности) без выполнения каких-либо числовых преобразований.
6. В формате double (64 бита) используется 11 бит для экспоненты и 52 бита для мантиссы (дробной части).
7. Для проверки условия «меньше или равно» для чисел с плавающей точкой (двойной точности) используется инструкция fle.d.
8. Для проверки погрешности при сравнении чисел с плавающей точкой обычно используют небольшое заранее заданное значение, называемое эпсилоном (epsilon).

Ответы на вопросы к главе 12

1. Наиболее эффективные методы оптимизации:
   - Минимизация количества инструкций – устранение избыточных операций и выбор наиболее подходящих команд.
   - Эффективное использование регистров сокращает время доступа к данным, так как доступ к регистрам значительно быстрее, чем к оперативной памяти.
   - Оптимизация циклов (например, разворачивание циклов) уменьшает накладные расходы на управление циклом и увеличивает пропускную способность программы.
   - Минимизация количества ветвлений. Ветвления могут вызывать задержки из-за неправильного предсказания переходов в конвейере процессора.
   - Применение специализированных инструкций RISC-V (например, расширений M, F, D, Zbb). Эти инструкции используются для ускорения специфических задач, так как они аппаратно реализованы. Важно их применять для повышения производительности кода и улучшения эффективности программы.

2. Влияние на производительность:

   – Использование регистров значительно повышает производительность, так как регистры являются частью схемотехники процессора и обеспечивают мгновенный доступ к данным, в то время как доступ к оперативной памяти требует времени на передачу данных.

   – Эффективное использование регистров сокращает потребность в обращении к памяти, снижая нагрузку на шину памяти.

   Ограничения:

   Процессоры RISC-V имеют ограниченное количество регистров общего назначения (31). Это означает, что не все переменные могут постоянно храниться в регистрах, часть данных приходится перемещать в стек или основную память, что замедляет работу.

3. Преимущества:

   – Уменьшение накладных расходов на управление циклом (инкремент счетчика, проверка условия, ветвление).

   – Увеличение пропускной способности программы за счет выполнения большего количества работы за одну итерацию и лучшего заполнения конвейера инструкций.

   Недостатки:

   – Увеличение размера итогового кода. Дублирование тела цикла приводит к большему исполняемому файлу, что может негативно сказаться на кэше инструкций.

   – Сложность реализации циклов с переменным количеством итераций.

4. Способы минимизации количества инструкций:

   – Устранение избыточных операций и выбор наиболее подходящих команд.

   – Перемещение кода, не зависящего от цикла, за его пределы.

   – Замена дорогостоящих инструкций (умножение, деление) на более дешевые (сдвиги, сложения), если возможно.

   – Использование битовых операций вместо условных ветвлений.

   – Применение макрофункций вместо вызовов функций для устранения накладных расходов.

   Необходимо учитывать, что минимизация инструкций часто приводит к ухудшению читаемости и поддерживаемости кода, а также может увеличить размер кода (например, при разворачивании циклов или использовании макросов).

5. Специализированные инструкции RISC-V, являющиеся частью расширений, могут быть использованы для оптимизации:

   – Расширение "M" (умножение и деление целых чисел): инструкции mul, div, rem. Применимы для выполнения арифметических операций над целыми числами, когда требуется высокая производительность аппаратной реализации.

- Расширения "F", "D" (операции с плавающей точкой): инструкции fadd.d, fmul.d, fsqrt.d и другие. Применимы для научных вычислений, 3D-графики, машинного обучения, где необходима работа с дробными числами.
- Расширение "Zbb" (битовые манипуляции): например, инструкция orc.b. Применимо для эффективной обработки битов, например, быстрой проверки наличия нулевых байтов в слове. Эти инструкции быстрее обрабатываются процессором по сравнению с программными реализациями, что приводит к значительному приросту производительности для конкретных задач.

6. Разворачивание циклов увеличивает размер итогового кода, поскольку тело цикла повторяется несколько раз, вместо того чтобы выполняться многократно в рамках одной небольшой циклической конструкции.
7. Минимизация ветвлений важна, потому что большое количество ветвлений может снижать производительность процессора. Процессоры RISC-V используют конвейерную обработку инструкций, поэтому при встрече с ветвлением процессор может отбросить уже выполненную работу по предзагрузке и декодированию следующих инструкций, если предсказание перехода оказалось неверным. Это приводит к перезапуску конвейера и значительной задержке.
8. Доступ к регистрам значительно быстрее, чем доступ к оперативной памяти. Это объясняется тем, что регистры являются частью схемотехники центрального процессора и обеспечивают мгновенный доступ к данным. Оперативная память же является отдельным компонентом, и для доступа к ней процессору требуется время для передачи данных через шину памяти.
9. Суть оптимизации через устранение избыточных операций заключается в удалении любых операций, которые не являются необходимыми для выполнения задачи, или которые можно вычислить лишь один раз вне цикла (например, инвариантный код цикла). Это включает устранение дублирующихся вычислений, излишних операций загрузки/сохранения данных или выбор более эффективных последовательностей инструкций.

Ответы на вопросы к главе 13

1. Сжатые инструкции RISC-V, такие как c.addi или c.mv, сокращают размер кода, заменяя стандартные 32-битные инструкции на 16-битные, что улучшает плотность кода и производительность. В Ghidra они отображаются в ассемблерном представлении, но декомпилированный код скрывает их, показывая эквивалентные операции на C. Это упрощает понимание логики, но может затруднить точный анализ, если требуется подсчет циклов или оценка размера кода. Например, c.addi a0, 4 увеличивает регистр на 4, но в декомпилированном виде это просто a0 += 4, без указания на сжатие.
2. Анализ точки входа _start (или app_main для микроконтроллеров) важен, поскольку это начальный адрес программы, с которого начинается выполнение кода. В этой точке часто происходят инициализационные действия (например, настройка стека, глобальных указателей, вызов среды выполнения C перед функцией main для Linux-программ). Понимание _start позволяет проследить начальный поток управления и общую структуру программы.
3. Динамическая линковка (использование общих библиотек) подразумевает, что код библиотеки находится в отдельном файле и загружается операционной системой динамически по мере необходимости.

Преимущества (для системы/использования):

- Уменьшение размера исполняемого файла.
- Экономия памяти: код загружается в память один раз и может использоваться несколькими приложениями.
- Упрощение обновлений: достаточно обновить библиотеку, не компилируя заново использующие ее приложения.

Недостатки (для анализа в Ghidra):

- Усложнение анализа: Ghidra должна учитывать, как динамически загружаются библиотеки и взаимодействуют с основным исполняемым файлом.
- Использование PIE/ASLR: общие библиотеки часто используют позиционно-независимый исполняемый код (PIE) и рандомизацию адресного пространства (ASLR), что делает адреса непредсказуемыми и усложняет статический анализ конкретных адресов.

4. Инструмент Ghidra позволяет:
   - Декомпилировать машинный код обратно в читаемый исходный код на языке C (по крайней мере, приближенно).
   - Анализировать скомпилированные программы.
   - Отображать графы вызовов функций.
   - Делать аннотации по мере анализа структуры программы. Это помогает понять логику программы, даже если исходный код недоступен, что полезно для реверс-инжиниринга, поиска уязвимостей или скрытых "пасхальных яиц".
5. Важно сопоставлять декомпилированный код с его поведением при выполнении (например, в эмуляторе QEMU), потому что это помогает на разных этапах разработки, например:
   - Верификация: инструменты, такие как Ghidra, генерируют приближенный исходный код на C, который не всегда полностью совпадает с оригинальным или оптимально скомпилированным кодом.
   - Понимание оптимизаций: компиляторы (например, GCC) значительно изменяют исходный код при компиляции в ассемблер. Выполнение помогает понять, как эти оптимизации влияют на фактическое поведение программы и как архитектура RISC-V влияет на исполнение.
   - Отладка и анализ уязвимостей: для понимания реального поведения программы, особенно при анализе сложных взаимодействий или уязвимостей, статический анализ декомпилированного кода может быть недостаточен; необходимо наблюдать за поведением в динамике.
6. При анализе кода, сгенерированного компилятором (например, GCC), можно обнаружить следующие виды инструкций:
   - Базовые целочисленные операции: сложение (add, addi), вычитание (sub), загрузка константы (li), перемещение регистров (mv).
   - Инструкции работы с памятью: загрузка (lw, ld, lb), сохранение (sw, sd, sb).

- Инструкции управления потоком: условные и безусловные ветвления (bgt, blt, bne, j, jal), возврат из функции (ret).
- Псевдоинструкции, которые компилятор расширяет из высокоуровневых конструкций (например, zext.b расширяется в andi для коррекции типов).
- Оптимизированные последовательности, например, упрощенные сравнения диапазонов или использование битовых операций вместо ветвлений.
- Специфические инструкции расширений, например, инструкции из расширения Zbb для битовых манипуляций, если компилятор их использует.

Ответы на вопросы к главе 14

1. Стек наиболее уязвим, потому что на нем хранятся локальные переменные, а также адреса возврата (ra) функций при их вложенных вызовах. Механизм перезаписи адреса возврата заключается в следующем: если в буфер, расположенный в стеке, записывается данных больше, чем он может вместить, происходит переполнение буфера. Это переполнение может перезаписать соседние области памяти в стеке, включая сохраненный адрес возврата (ra). Когда функция завершает свою работу, она пытается вернуться по адресу, записанному в ra. Если этот адрес перезаписан злоумышленником, функция переходит по указанному хакером адресу, что позволяет выполнить произвольный вредоносный код.
2. Shellcode – это внедряемый вредоносный код. При эксплуатации переполнения стека он внедряется в память программы (часто в сам стек), а затем адрес возврата функции перезаписывается таким образом, чтобы он указывал на начало этого вредоносного кода. Таким образом, когда функция пытается вернуться, она фактически запускает код злоумышленника.
3. «Канарейка» (stack canary) – это секретное случайное значение, размещаемое в стеке. Она размещается рядом с адресом возврата функции (обычно перед ним). Принцип работы: перед возвратом из функции программа проверяет, не было ли изменено значение канарейки. Если значение повреждено, это указывает на переполнение буфера (попытку перезаписи стека), и программа аварийно завершается, предотвращая выполнение вредоносного кода.
4. ASLR (Address Space Layout Randomization) — это метод защиты, при котором операционная система случайным образом размещает сегменты памяти (исполняемый файл, библиотеки, стек, кучу) в адресном пространстве процесса. Это усложняет эксплуатацию уязвимостей, так как хакеру становится трудно предсказать точные адреса, по которым будут расположены важные данные (например, адреса возврата или внедренный код). Поскольку адреса меняются при каждом запуске программы, жесткое кодирование адресов в эксплойте становится неэффективным.
5. Различия защит DEP (NX), ASLR и PIE
   - DEP (Data Execution Prevention) / NX (No-Execute bit) – это аппаратная защита, которая запрещает выполнение кода из областей памяти, предназначенных для данных (например, стека или кучи). Это предотвращает выполнение внедренного вредоносного кода, если он помещен в эти области.
   - ASLR (Address Space Layout Randomization) рандомизирует базовые адреса загрузки исполняемых файлов, библиотек, стека и кучи в памяти.

- PIE (Position Independent Executables) – это конкретный случай ASLR, применяемый к самому исполняемому файлу, делая его позиционно-независимым и загружающимся по случайному адресу.
- Взаимодействие ASLR и PIE усложняют хакеру предсказание того, куда совершать переход в памяти, а DEP/NX усложняет выполнение кода, даже если хакеру удалось внедрить его в сегмент данных. Вместе эти механизмы создают многослойную защиту, значительно усложняя атаку.

6. Обойти защиту можно следующим образом:
    - fstack-protector (канарейки): можно обойти, если хакеру удастся узнать значение канарейки (например, через утечку информации) или если реализация обработки ошибки не прерывает программу немедленно.
    - PIE: если в сообщениях об ошибках или других местах происходит утечка реальных адресов памяти, позволяющая хакеру определить смещение, защита может быть пройдена. Также, если любая общая библиотека скомпилирована без PIE, это может отключить защиту для всего приложения.
    - DEP/NX: можно преодолеть с помощью техник, таких как Return-Oriented Programming (ROP), когда атакующий использует существующие фрагменты кода в памяти для построения своей логики.
    - Ручной анализ остается важным методом, так как он:
        - Помогает найти тонкие ошибки и неочевидные уязвимости, которые могут быть упущены автоматизированными средствами.
        - Позволяет понять сложные взаимодействия между компонентами и выявить цепочки атак.
        - Компиляторы могут оптимизировать код таким образом, что его поведение становится непредсказуемым или содержит неочевидные побочные эффекты, требующие ручной проверки.
7. Важно минимизировать привилегии при разработке программ (принцип наименьших привилегий), потому что, если программа запускается с минимально необходимыми правами доступа, потенциальный ущерб от успешной атаки (например, переполнения буфера) будет значительно ограничен. Даже если злоумышленник получит контроль над программой, он не сможет выполнить действия, на которые у самой программы нет разрешений, например, получить root-доступ или доступ к конфиденциальным системным ресурсам.