Архитектура вычислительных систем

Содержание семинарских занятий

А. И. Легалов С. А. Виденин К. И. Пашигорев

31 октября 2023 г.

Оглавление

Введение

Назначение документа

Приводится содержание семинарских занятий, проводимых по дисциплине "Архитектура вычислительных систем в 2023-2024 учебном году. Он является основой для проведения занятий семинаристами. Для каждого семинара представлены:

- тема семинарского занятия;
- цель занятия;
- общий план занятия;
- содержание занятия (решаемые задачи, рассматриваемые вопросы) в соответствии с представленным планом, включающее последовательность действий, а также дополнительные комментарии и примечания к процессу проведения;
- описание домашнего задания;
- пример (или примеры) выполнения домашнего задания (зачастую это один из возможных вариантов);
- список основной и дополнительной литературы, рекомендуемой по теме семинара и необходимой для выполнения задания;
- список используемого аппаратного и программного обеспечения.

Данный документ не является догмой и жестким предписанием. Он определяет общий план и включает ряд пояснений, сформированных на основе предшествующего опыта. Поэтому он может уточняться и изменяться всеми участниками учебного процесса. Основной его целью является помощь в более быстром вхождении в процесс проведения семинаров тем кто впервые приступает к занятиям по данной дисциплине.

Изменять, расширять, дополнять не только можно, но и нужно. Задача — сформировать общие алгоритмы, которые в дальнейшем можно использовать как паттерны, адаптируя их к текущим ситуациям. Изменение эти алгоритмов возможно еще и потому, что каждый год, опираясь на предыдущий опыт, идет изменение учебного процесса. Этот процесс изменения является постоянным, так как нельзя дважды войти в одну и ту же реку...

Краткое содержание семинаров

> **Примечание**: По ходу написания этот список будет уточняться и корректироваться.

В соответствии с учебным планом предполагается проведение пятнадцати семинаров. Предполагается, что это будут следующие темы:

1. Эмулятор RARS. Первоначальное знакомство.
2. Организация памяти. Методы адресации. Ветвления и переходы. Системные вызовы.
3. Целочисленная арифметика. Одномерные и многомерные массивы. Простые алгоритмы.
4. Подпрограммы. Стек. Кадр стека. Параметры. Локальные переменные.
5. Немного об ассемблере. Многофайловые программы. Директивы. Макросы.
6. Математический сопроцессор. Арифметика с плавающей точкой.
7. Строки символов. Обработка символьных данных. Файлы.
8. Обработка исключений.
9. Ввод–вывод данных. Поллинг. Программирование ввода–вывода.
10. Таймер. Прерывания по таймеру.
11. Программирование ввода–вывода. Использование прерываний.
12. Микроархитектура. Предсказание переходов. Кеширование.
13. Микроархитектура. Конвейеризация.
14. Поддержка многозадачности. Виртуализация. Многоядерность.
15. Мультипроцессоры. Специализированные процессоры.

> **Примечание**: Если во введении необходимы дополнительные краткие пояснения по темам, то можно добавить.

1 Семинар 1. Эмулятор RARS. Первоначальное знакомство

1.1 Цель и задачи

> **Примечание**: На мой взгляд, изучение среды и особенностей организации системы команд имеет смысл начинать с непосредственного погружения по принципу: делай как я. Естественно, что первоначально не все будет понятно, но в ходе демонстрации можно уже начинать объяснять, что почему и как устроено, как это работает. Давать изначально голую теорию с последующей демонстрацией практически бесполезно. Особенно когда имеется такое наглядное пособие, как эмулятор. Исходя из этого можно при изучении среды сразу же совместить такие вопросы как: форматы команд, ассемблер, эмулятор RARS процессора RISC-V, особенности IDE, имитаторы системных вызово, используя для демонстрации простой пример. Часть информации может быть взята по ссылке: `http://uneex.org/LecturesCMC/ArchitectureAssembler2022/01_AboutRiscV`

Целью семинара является изучение основных особенностей эмулятора RARS, используемого в ряде семинаров для изучения особенностей архитектуры 32-разрядного процессора RISC-V. В ходе его проведения предполагается рассмотреть следующие вопросы.

1. Получение общих сведений о работе эмулятора и мотивах его использования:
   - Пояснение, почему используется эмулятор, а не реальная вычислительная система.
   - Характеристики эмулятора и краткая история его создания.
   - Что из себя представляет эмулятор RARS. Что он может.
2. Изучение процесса установки эмулятора на компьютеры с различной архитектурой и разными операционными системами.
   - Где можно взять эмулятор RARS. Сайт эмулятора. Софт на Гитхаб.
   - Установка эмулятора в различных операционных системах.
3. Демонстрация работы эмулятора на примере простых программ.
   - Запуск эмулятора в режиме среды разработки.
   - Основные компоненты эмулятора и их назначение.

- Особенности распределения адресного пространства памяти под программу и данные.
- Создание простейших программ на ассемблере RISC-V, их компиляция и запуск из среды эмулятора.
- Прогон готовых простых программ в автоматическом режиме и в режиме отладчика.
- Запуск эмулятора в режиме командной строки.

1.2 Общие соображения

> **AL**: Так как обычно лекции предшествуют семинарам, я постараюсь, особенно на первых лекциях, делать упреждающий рассказ об архитектуре процессора и его системе команд. Думаю, что половину этих лекций буду уделять общим вопросам, а другую часть конкретно архитектуре RISC-V и сопутствующим инструментальным средствам, чтобы студенты на занятиях имели начальное представление, и не было на семинарах соответствующего теоретического материала. Скореее всего это будут отдельные презентации, которые будут пополняться отдельно. Каждая из них будет выкладываться в соответствующую тему.

Для проведения занятий предполагается использование компьютерного класса с установленной на компьютерах ОС Linux или Windows. В этом как раз и заключается особенность эмулятора, что можно его использовать в любой из ОС. Но имеет смысл разобраться: есть ли под установленной версией Linux пакет, содержащий RARS. Если да, то можно будет установить его из дистрибутива. Помимо этого для любой из ОС его можно непосредственно скачать из репозитория: `https://github.com/TheThirdOne/rars/releases`
Сам пакет можно будет скачать заранее и выложить в LMS.

> **AL**: Еще возможной проблемой (хотя вряд ли, так как java ориентирована на Юникод) может быть локализация. Нужно это тоже предварительно выяснить. Нужно предварительно поиграться с примерами программ на Windows, используя русскоязычные комментарии. В Linux обычно проблем нет. По крайней мере у меня. Но можно просто комментарии делать на английском или транслитом, если такие проблемы возникают.

Традиционно на семинаре нужно будет использовать проектор. Но также думаю, что изначально можно будет запускать эмулятор RARS, проверив его работоспособность как под Linux, так и под Windows. Сценарий первого семинара в данном случае предполагает действия по принципу: делай как я. Начиная с простейших программ, состоящих из пары команд или псевдокоманд, можно перейти к системным вызовам обеспечивающим ввод-вывдо. То есть, общий алгоритм семинара можно выставить по схеме лекций Георгия Курячего. Но с непосредственным выполнением шагов студентами. По ходу демонстрации можно неоднократно открывать систему помощи и показывать информацию, которая там имеется, при

комментарии выполняемых шагов.

1.3 Сценарий семинара

1.3.1 Основные сведения об эмуляторе

С мотивами использования эмулятора все просто. Это преодоление разнотипности различного железа и операционных систем. Эмулятор, реализованный на java, может быть запущен практически везде. С другой стороны для изучения конкретной архитектуры на уровне системы команд использование реального железа не является необходимым. Предлагаемый эмулятор, помимо системы команд позволяет изучить и много другое, включая программирование ввода-вывода, прерывания, кеш память, а также прогнозирование переходов. Также стоит отметить перспективу архитектуры и ее достаточную простоту по сравнению с CISC системами.

По поводу использования других систем эмуляции. Можно отметить, что их в настоящее время много. Зачастую они запускаются в виртуальных машинах, что позволяет использовать операционные системы и средства разработки на различных языках программирования. Имеются также системы кросс компиляции. Однако в рамках дисциплины эти вопросы не являются актуальными и поэтому не рассматриваются.

1.3.2 Установка эмулятора

Так как базовую информацию по основам архитектуры и системе команд я предполагаю дать на лекции, то можно сразу стартовать с мотивов использования эмулятора и его местоположения в сети. Также (особенно при проблемах доступа к эмулятору в сети, его можно будет установить (если еще не установлен) в домашний каталог из LMS. Думаю, что туда он предварительно ляжет. Установить можно в любой домашний каталог и запускать используя консоль, как это делает Курячий. Если, конечно, он не запускается по умолчанию. Запуск в ОС Linux осуществляется в этом случае просто:

```
java -jar rars1_6.jar
```

> **AL**: У меня запуск эмулятора, установленного в домашний каталог, осуществляется по умолчанию как пакета java. Помимо этого в ряде дистрибутивов эмулятор может быть установлен из системного репозитория. Тогда он будет присутствовать в меню запуска. Это, например, реализовано в Manjaro Linux, а также в Simply Linux. Как запускаются подобные пакеты в Windows – посмотрите, кто этой ОС пользуется. Думаю, что на Маке проблем тоже не возникнет.

1.3.3 Демонстрация работы эмулятора

После запуска необходимо охарактеризовать основные окна, меню, инструментальные панели. Все как обычно. Обратить внимание на систему помощи и имеющиеся в ней разделы.

Продемонстрировать процесс использования простой программы расположенной в учебном каталоге: загрузку программы, ее компиляцию, запуск на выполнение. Выполнение в пошаговом режиме, в режиме точек останова. Можно это все делать по принципу: делай как я. При этом можно сразу не акцентировать внимание на распределении памяти эмулятора под программу, данные, ядро, адресное пространство ввода-вывода, так как это будет далее рассматриваться в ходе изучения архитектуры.

Простейшая программа нахождения суммы двух чисел

В качестве первого примера можно использовать программу, которая складывает пару чисел.

```
li      a7 5        # Системный вызов №5 - ввести десятичное число
ecall               # Результат - в регистре a0
mv      t0 a0       # Сохраняем результат в t0
ecall               # Регистр a7 не менялся, тот же системный вызов
add     a0 t0 a0    # Прибавляем ко второму число первое
li      a7 1        # Системный вызов №1 - вывести десятичное число
ecall
li      a7 10       # Системный вызов №10 - останов программы
ecall
```

Обратить внимание на то, что возможна компиляция как одного файла за один раз, так и всех файлов, расположенных в одном каталоге. Последнее позволяет осуществлять для эмулятора разработку многофайловых проектов. Для текущей работы предложить убрать флаг у параметра: «SettingsAssemle all file iin directory», если он установлен. В этом случае будет компилироваться только один файл, расположенный в открытом окне редактора. Также можно поизучать и обсудить (но поверхностно) и другие флаги.

В ходе пояснения обсудить и пояснить отличие псевдокоманд от команд. Показать где и как это отличие отображается. Также рассмотреть на примере простейших программ изменение данных и регистров. Обсудить псевдонимы (алиасы) регистров и специализацию регистров в соответствии с соглашением по их использованию для данной архитектуры процессора. Пояснить использование регистра zero, а также концепцию разработчиков системы команд, направленную на устранение избыточных команд.

Подчеркнуть специфику системы команд RISC процессоров, связанную с тем, что форматы команд предназначены для организации эффективной работы процессора, а для написания программ человеком используются ассемблеры.

Это во многом отличает ассемблеры RISC систем от ассемблеров CISC процесооров, в которых обычно каждая машинная команда однозначно соответствует инструкции на ассемблере, а формат машинной команды имеет логически выстроенную структуру полей. В RISC процессора поля в формате команды распределены так, чтобы можно было обеспечить эффективную обработку в устройстве управления.

> **AL**: На данном этапе изучения можно (если не возникнут соответствующие вопросы) не останавливаться на специфике форматов команд. Правда можно сказать, что формат команд ориентирован на эффективность их обработки устройством управления процессора, а не на программиста. Пояснить, что на программиста ориентирован ассемблер, в котором есть удобные мнемоники как для команд, так и псевдокоманд. Все эти вопросы также будут многократно повторяться в ходе изложения лекционного материала.

После этого можно приступить к обсуждению и демонстрации эмуляции простых системных вызовов (ecall), обеспечивающих ввод-вывод целочисленных данных. Пояснить, что это не вызовы ОС, а обращение к библиотекам java, которые и занимаются соответствующей эмуляцией. Но во многом эти вызовы похожи на вызовы, осуществляемые в операционных системах. Показать, где эти вызовы описаны в системе помощи эмулятора.

Первая программа из серии «Hello»

На примере этой программы можно рассмотреть использование системного вызова, осуществляющего вывод строки, представленной в формате языка Си (завершающейся нулем).

```
.text
    la a0, string       # buffer
    li a7, 4            # syscall write (4)
    ecall
    li a0, 0            # exit code
    li a7, 10           # syscall exit
    ecall
.data
    string:  .asciz "Hello! It works!!!\n"
```

Можно пояснить, что еще есть строки которые могут не завершаться нулем. Но этом вариант достаточно удобен для организации вызова.

Помимо этого обратить внимание (в поверхностном варианте) на распределением памяти в эмуляторе. Пояснить, откуда начинается адресное пространстов данных. Задание адресных пространств данных и кода директивами `.data` и `.text`.

Вторая программа из серии «Hello»

На примере этой программы показать, что непринципиально, в какой последовательности следуют описания данных и кода.

```
    .data
hello:
    .asciz "Hello, world!"
    .text
main:
    li a7, 4
    la a0, hello
    ecall
```

Третья программа из серии «Hello»

В принципе можно чередовать секции кода и данных. Они в конце концов соберутся правильно.

```
.text
    la a0, string       # buffer
    li a7, 4            # syscall write (4)
.data
    string:  .asciz "Hello! It works!!!\n"
.text
    ecall
    li a0, 0            # exit code
    li a7, 10           # syscall exit
    ecall
```

Четвертая программа из серии «Hello»

Можно также посмотреть, как будут представлена кириллица при выводе данных. В принципе в Java (на чем написан эмулятор) используется Юникод. Но посмотреть стоит, что будет под разными ОС.

```
.text
    la a0, string       # buffer
    li a7, 4            # syscall write (4)
    ecall
    li a0, 0            # exit code
    li a7, 10           # syscall exit
    ecall
.data
    string:  .asciz "Привет. Русский язык выглядит так!!!\n"
```

В целом при выводе все должно быть нормально. Но отображение не `ascii` символов в памяти может не получитья. Только в шестнадцатиричном формате. Для демонстрации можно поиграться флажками, которые определяют формат вывода дампа памяти.

Еще раз сложение двух чисел

При наличии времени (или в быстром режиме) можно сделать обзор программы сложения двух числе, котора выводить дополнительные сообщения перед вводом или выводом данных.

```
.data
    arg01:  .asciz "Input 1st number: "
    arg02:  .asciz "Input 2nd number: "
    result: .asciz "Result = "
    ln:     .asciz "\n"
.text
    la  a0, arg01   # Подсказка для ввода первого числа
    li  a7, 4       # Системный вызов №4
    ecall
    li      a7 5    # Системный вызов №5 - ввести десятичное число
    ecall           # Результат - в регистре a0
    mv      t0 a0   # Сохраняем результат в t0

    la  a0, arg02   # Подсказка для ввода второго числа
    li  a7, 4       # Системный вызов №4
    ecall
    li      a7 5    # Системный вызов №5 - ввести десятичное число
    ecall           # Результат - в регистре a0
    mv      t1 a0   # Сохраняем результат в t1

    la a0, result   # Подсказка для выводимого результата
    li a7, 4        # Системный вызов №4
    ecall
    add   a0 t0 t1  # Складываем два числа
    li      a7 1    # Системный вызов №1 - вывести десятичное число
    ecall

    la a0, ln       # Перевод строки
    li a7, 4        # Системный вызов №4
    ecall

    li      a7 10   # Системный вызов №10 - останов программы
    ecall
```

1.3.4 Использование эмулятора в режиме командной строки

На данном этапе изучения достаточно продемонстрировать простейшие действия с представленными выше программами. То есть, показать, каким образом можно

осуществить ассемблирование программы, а также ее запуск на выполнение. Предполагается, что более детальное рассмотрение (если это будет необходимо) будет сделано в ходе последующих семинаров.

1.4 Разное

При демонстрации программ основной упор делать на особенности организации среды. Можно часто обращаться к системе помощи программы для демонстрации различных особенносте в моменты рассмотрения тех или иных вопросов.

Привести ссылки на используемые источники, указав, какая тема затронута. Сами источники в конце методы. Они могут быть общими для всех документов.

1.5 Домашнее задание

1.5.1 Оценка до 8 баллов

Установить RARS и запустить в нём программы, размещенные в LMS (в теме первого семинара). Сформировать отчет об использовании эмулятора, в котором привести скриншоты, демонстрирующие работу эмулятора с этими программами.

В отчете описать, какие команды являются псевдокомандами, проанализировав для этого результаты компиляции одной из программ.

Описать типы форматов команд для одной из представленных программ.

Описать какие системные вызовы используются в изученных программах. Для этого достаточно обратиться к системе помощи эмулятора.

1.5.2 Опционально +2 балла

Записать на видео процесс работы эмулятора с этими программами в режимах компиляции и автоматического выполнения. Одну из программ выполнить в пошаговом режиме с комментариями.

2 Семинар 02. Регистры. Организация памяти. Директивы. Описание данных.

2.1 Цель и задачи

Целью семинара является знакомство с организацией регистров и памяти эмулятора процессора RISC-V, Методов представления данных в памяти, распределения пространства памяти с использованием различных директив. Помимо этого предполагается изучение директив, разделяющих данные и код, а также вариантов их использования.

1. Регистры.
    - Особенности организации целочисленных регистров процессора, их имена и алиасы.
    - Специализация ряда регистров.
    - Соглашения по использованию регистров.
2. Соглашения об использовании модели памяти в RARS
    - Общее понятие о моделях памяти. Контекст программы (процесса).
    - Области (секции) памяти и распределение адресного пространства.
    - Отображение областей памяти в ассемблере.
    - Отображение областей памяти в RARS.
3. Описание данных различного типа на ассемблере в области данных.
    - Мнемонические обозначения для данных различного типа и их задание соответсвующими константами.
    - Резервирование областей памяти заданного размера.
    - Выравнивание в памяти.

2.2 Сценарий семинара

> **Примечание**: По сути в рамках семинара предполагается рассмотреть, каким образом описываются данные и выполняются команды, связанные с адресацией памяти, используемой в процессоре. Также необходимо затронуть, на какие форматы команд накладываются те или иные методы адресации. Информация во многом соответствует содержанию страницы на сайте Георгия Курячего: http://uneex.org/LecturesCMC/ArchitectureAssembler2022/02_MemoryRegisters

Имеет смысл лишний раз подчеркнуть специфику системы команд RISC процессоров, связанную с тем, что форматы команд предназначены для организации эффективной работы процессора, а для написания программ человеком используются ассемблеры.

Также стоит отметить, что принятые специальные соглашения (конвенции) об использовании регистров (их виртуальной специализации) помогают писать совместимые и переносимые программы. Помимо этого разработчики процессоров могут использовать эти соглашения (включая и соглашения по замене псевдокоманд конкретными командами) для оптимизации аппаратных решений, что обеспечивает повышение производительности систем.

2.2.1 Регистры процессора

Основная идея в представлении регистров заключается в пояснении их количества, а также вариантов использования в соответствии с изначально принятыми соглашения. Рассказать, для чего предназначены соглашения по использованию регистров. Также пояснить почему ряд регистров реализованы с соответствующей специализацией. При этом можно более подробно остановиться на специализации отдельных регистров. Можно также вкратце отметить, что делают, когда количество параметров при вызове некоторой функции превышает число выделенных для этого регистров. Обычно при нехватке регистров оставшиеся параметры выкладываются на стек. Но на данном этапе заострять внимание на этом нет смысла. Также сказать. Почему не все параметры размещают на стеке, что было раньше. Пояснить это повышением производительности и увеличением числа регистров.

> **Примечание**: Размещение на стеке для RISC-V я пока это не смотрел. Но в дальнейшем при изучение процедур к этому стоит вернуться.

Таблица 2.1 описывает особенности распределения и использования регистров.

2.2.2 Распределение адресного пространства памяти

Дать пояснения что практически во всех системах осуществляется распределение области памяти между различными программами в соответствии с принятыми соглашениями, что позволяет упростить процесс распределения соответствующего ресурса. В эмуляторы RARS тоже приняты соответствующие соглашения и поддерживаются соответствующие модели памяти, упрощающие разработку программ.

Таблица 2.1 — Обозначение и использование регистров процессора

Регистры	**Псевдонимы**	**Соглашения по использованию**
x0	zero	Всегда равен 0 (hard-wired zero)
x1	ra	Адрес возврата (return address)
x2	sp	Указатель стека (stack pointer)
x3	gp	Адрес области глобальных данных (global pointer)
x4	tp	Указатель потока (thread pointer)
x5–x7	t0–t2	Временные регистры 0–2 (temporaries 0–2)
x8	s0, fp	Сохраняемый регистр 0 или указатель фрейма (saved register 0, frame pointer)
x9	s1	Сохраняемый регистр 1 (saved register 1)
x10–x17	a0–a7	Регистры параметров подпрограмы и возврата значений из них (function arguments 0 to 7)
x18–x27	s2–s11	Сохраняемые регистры 2–11 (saved registers 2–11)
x28–x31	t3–t6	Временные регистры 3–6 (temporaries 3–6)
pc	pc	Указатель команд (program counter)

Рассказать о принятых типовых соглашениях. Для этого можно сослаться на то, как в существующих ОС формируется контекст программы (процесса) на примере программы, написанной на Си.

> **Примечание**: Соответствующий материал будет представлен в методе для самостоятельной работы. Он уже сформирован по предшествующим годам. Возможно он также будет представлен и в лекционных материалах.

Таблица 2.2 описывает соглашения, принятые для распределения адресного пространства памяти в эмуляторе RARS.

2.2.3 Директивы для размещения данных в памяти

В рамках данной темы предлагается разместить данные в памяти и рассмотреть с использованием средств эмулятора их размещение после ассемблирования. Рассмотреть, каким образом осуществляется размещение в памяти для используемых в эмуляторе типов данных:

- `.word` число — одно или несколько 4-байтовых чисел;
- `.dword` число — одно или несколько 8-байтовых чисел;
- `.half` число — одно или несколько 2-байтовых чисел;
- `.byte` число — одно или несколько однобайтовых чисел;
- `.ascii` строка— последовательность символов в кодировке ASCII;

- `.asciz` строка— последовательность символов в кодировке ASCII в конце которой добавляется нулевой байт (строка в стиле языка программирования Си).

Изначально можно создать только одну секцию данных. После этого можно добавить код, который выводит данные с использованием системных вызовов.

Пример (из материалов Курячего):

```
.data
    .word   0xdeadbeef
    .dword  0xacebad0feeded
    .half   0x1234, 0x5678
    .byte   12, 13, 14, 15
    .half   0x3344
    .byte   0x66, 0x77
```

Предложить набрать (или скопировать с LMS) данный код и откомпилировать его. Рассмотреть, каким образом оно ляжет в памяти эмулятора. Сравнить с демонстрацией на проекторе. Основная задача — закрепление навыков работы с эмулятором. Результат трансляции пословно:

```
10010000: deafbeef d0feeded 000aceba 56781234 0f0e0d0c 77663344
```

Обратить внимание, что секция `.data` начинается по умолчанию с адреса `0x10010000`), а данные расположены в формате little endian (младший байт в слове имеет меньший адрес). Можно также поиграться с разными форматами отображения памяти, придоставляемыми эмулятором.

В рамках следующего задания рассмотреть, каким образом можно зарезервировать адресное пространство под данные, которые будут формироваться во время вычислений (`.space`). В частности, под массивы данных. Обсудить соответсвтующие директивы. Помимо этого затронуть необходимость выравнивания данных и соответствующие директивы, демонстрирующие это (`.align`). Все эти вопросы, как и в случае предыдущего примера, можно рассмотреть на соответствующем коде:

```
#
# Example: Data and its alignment.
#
.data
    .space 3
    word1:
    .word  0x12345678
    half1:
    .half  0x1234
    byte1:
    .byte  0x12
    .align 4
```

```
word2:
.word 0x12345678
.align 3
half2:
.half  0x1234
.align 3
byte2:
.byte  0x12
.align 0
word3:
.word 0x12345678
```

Откомпилировать данный код и посмотреть полученное распределение в памяти.

Таблица 2.2 — Соглашения по распределению памяти эмулятора RARS

Адреса	Назначение	Целевое выделение памяти
0xffffffff	**Память устройств**	Последний адрес, доступный ядру (highest address in kernel)
		Конец памяти устройств (memory map limit address)
0xffff0000		Начало памяти устройств (MMIO base address)
0x80000000	**Область памяти ядра**	Начало памяти ядра (Начало памяти ядра)
0x7fffffff	**Область данных**	Последняя ячейка, доступная пользователю (highest address in user space)
0x7fffffff		Последняя ячейка области данных (data segment limit address)
0x7ffffffc		Адрес исчерпания стека (↓ stack base address)
0x7fffeffc		Сюда указывает регистр стека. Растёт вниз (↓ stack pointer sp)
0x7fffeffc		Сюда указывает регистр стека. Растёт вниз. (↓ stack pointer sp)
0x10040000		Нижняя граница роста стека (stack limit address)
		Начало кучи. Растёт вверх (↑ heap base address)
0x10010000		Начало статических данных (.data base Address)
0x10008000		Сюда указывает регистр глобальных данных (global Pointer gp)
0x10000000		Область глобальных данных (.extern base address)
		Начало области данных (DATA Segment base address)
0x0ffffffc	**Область кода**	Последняя ячейка области программного кода (text limit address)
0x00400000		Начало программы (.text base address)
		Начало области программного кода (TEXT segment base address)
0x00000000	**Зарезервировано**	

3 Семинар 03. Система команд. Методы адресации. Ветвления и переходы. Системные вызовы

Целью семинара является знакомство с организацией регистров и памяти эмулятора процессора RISC-V, использованием методов адресации для работы с памятью. Помимо этого предполагается более детальное изучение работы отладчика, а также мониторинга регистров и памяти. Основны вопросы, которые предполагается рассмотреть на семинаре.

1. Организация команд. Система команд.
   - Форматы команд. Общая идея предложенных форматов.
   - Мнемоника команд.
   - Псевдокоманды.
2. Область кода.
   - Методы адресации данных, используемые в ассемблере RISC-V.
   - Моделирование на ассемблере различных операторов управления.
3. Имитация в эмуляторе системных вызовов. Аналогии системных вызовов в операционных системах.
4. Примеры простых целочисленных алгоритмов.

3.1 Сценарий семинара

3.1.1 Организация команд. Система команд

Кратко (по сути как повтор общей классификации) определить специфику системы команд RISC процессоров и ее отличие от системы команд CISC процессоров. Также отметить, что для семейства RISC-V существуют различные системы команд в зависимости от разрядности процессоров и их дополнительного обвеса. Но имеется общая специфика, связаннай с построением форматов и размером команд (32 разряда)

Форматы команд. Общая идея предложенных форматов

Привести и пояснить (достаточно кратко) основные форматы команд 32-разрядной версии процессора (RV32I). Рассмотреть четыре базовых типа команд (рисунок 3.1):

- **R** — типа «регистр-регистр-регистр» (Register)
- **I** — типа «непосредственное значение-регистр-регистр» (Immediate)
- **S** — типа «регистр-регистр-непосредственное значение» (Store)
- **U** — типа «непосредственное значение-регистр» (Upper)

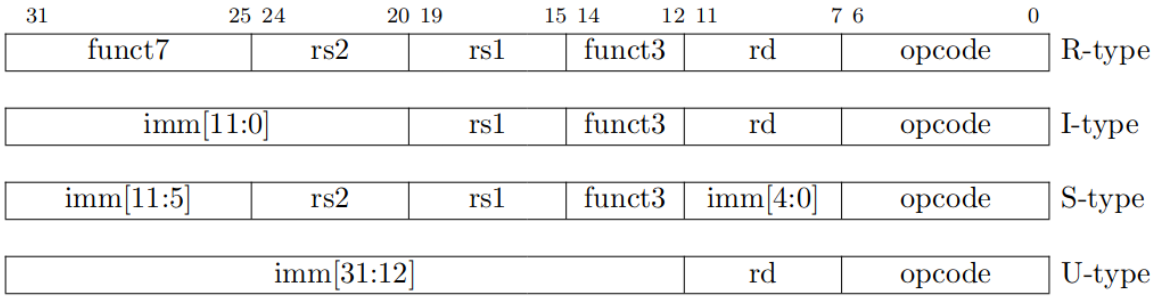


Рисунок 3.1 — Основные форматы команд 32-разрядного процессора RISC-V

Обозначения на рисунке:

- **opcode** — код операции (6 битов)
- **rs1** — № регистра-источника (5 битов)
- **rs2** — № регистра-операнда (5 битов)
- **rd** — № регистра-приёмника (5 битов)
- **imm[11:0]** — непосредственный операнд размером в 12 битов (В случае, когда непосредственное значение определяет «приёмник» (смещен адреса для «близкого» перехода или записи результата в память), 12 битов целиком в поле rd не помещаются, и его приходится «распиливать» (инструкция типа S). Непосредственный операнд всегда знаковый, и его знак всегда приходится на 31-й бит)
- **imm[31:12]** — непосредственный операнд размером в 20 битов. Используется в инструкциях типа U для заполнения старших двадцати битов регистра (в операциях «далёкого» перехода и как дополнительная инструкция при записи в регистр полного 32-разрядного непосредственного операнда)

- **funct** — поле функции (6 битов), используется для разных инструкций, у которых код операции одинаковый. Например, все арифметические инструкции типа I имеют одинаковый opcode OP-IMM (чему он равен?), а различаются полем funct. По-видимому, для эффективной реализации R-команд в конвейере удобнее не декодировать опкод, а по-быстрому сравнить его с нулём, и получать значения регистров, параллельно декодируя функцию, чтобы потом её применить.

Мнемоника команд

Изначально мнемонику команд можно рассмотреть в системе помощи RARS (раздел Basic Inctructions), где в алфавитном порядке представлены практически все команды. Однако этот формат не вполне удобен для постепенного изучения. Поэтому можно сослаться на пару таблиц, в которых команды систематизирован, но представлены без описания. Можно предложить совместное использование этой справочной информации по мере необходимости. В принципе в методе по самостоятельной работе данный материал стоит систематизировать и выложить в LMS. Начало положено. Там можно сделать все в соответствии с порядком изучения. Обе таблицы с кратким представлением набора команд (`RARS_reference_card.pdf` и `riscv-reference-card.pdf`) я положил в каталог `info` на `gitflic`.

Псевдокоманды

Рассмотрение мнемоники псевдокоманд в целом можно осуществить по той же схеме, что и изучение команд. Их достаточно обширный список представлен в системе помощи. Однако, в связи с тем, что речь идет о первоначальном знакомстве, вряд ли имеет смысл на этой мнемонике долго останавливаться. Также можно обратиться к отображениям в дополнительной информации. Или к методическим материалам для самостоятельного изучения (надеюсь, что соответствующие материалы и презентации разработаем).

3.2 Область кода

Можно еще раз подчеркнуть адресное пространство, в котором располагается программный код. Также отметить, что при описании на ассемблере соответствующими директивами (`.data` и `.text`) можно чередовать код и данные, а ассемблер разместит их в своих областях.

3.2.1 Методы адресации данных, используемые в ассемблере RISC-V

Пояснить основные методы адресации, которые используются в процессоре на уровне обмена с памятью. Показать необходимость использования адресации в па-

мяти как для обращения к данным и их пересылки, так и для организации управления вычислениями.

Ветвления и переходы

Можно рассмотреть базовые инструкции вида «сравнить и перейти» (`beq/bne /blt/bge/btlu/bgeu`. Все остальные инструкции вида `b*`, включая безусловный переход `b` — это псевдоинструкции, потому что получаются простой перестановкой регистров–операндов или подстановкой регистра `zero` в нужное место. Непосредственный 12-битный операнд используется для хранения смещения относительно текущего адреса. Вычислением этого смещения из метки занимается ассемблер, оно бывает положительное (вперёд) и отрицательное (назад, как в примере). 12 битов хватает на то, чтобы сделать переход на 4 килобайта кода (4, а не 2, потому что адрес всегда кратен 2 и самый младший бит просто не хранится).

Пример перехода по постусловию (из материалов Курячего):

```
# Переходы по постусловию
li      s2 10           # Граница счётчика
li      s1 1            # Счётчик
loop:   li      a7 1            # Вывод счётчика
mv      a0 s1
ecall
addi    s1 s1 1         # Увеличение
blt     s1 s2 loop      # Сравнение счётчика и границы и переход
li      a7 10           # Останов
ecall
```

Можно рассмотреть, что получится после компиляции и выделить непосредственные 12–разрядные операнды

Пример перехода по предусловию с длинным переходом (из материалов Курячего):

```
li      s2 10
li      s1 1            # Инициализация
loop:   bge     s1 s2 final     # Проерка условия
li      a7 1            # Тело
mv      a0 s1
ecall                   # Вывод целого
li      a7 11
li      a0 10
ecall                   # Вывод перевода строки
addi    s1 s1 1         # Изменение
j       loop            # Дополнительный переход
final:  li      a7 10
ecall
```

Безусловный переход может называться b, а может j — это одна и та же псевдоинструкция. Это инструкция «длинного» перехода типа J (20 битов, т. е. по мегабайту в обе стороны с учётом ещё одного младшего бита). Есть более подробное описание у Курячего. Но думаю, что сильно на данном этапе заморачиваться не стоит, так как пока программы простые. А в методических указаниях можно более подробно описать. Возможно, что данный пример можно пока тоже не рассматривать...

Косвенная адресация и массивы

Как вариант доступа к данным — записать полный абсолютный адрес в регистр, и воспользоваться инструкцией, которая работает с памятью, находящейся по этому адресу. Пример использования псевдоинструкции `lw метка` (из Курячего):

```
.data
.word   0x1223344
var:    .word   0xdeadbeef
addr:   .word   var
.text
lw      t1 var
lw      t2 addr
lw      t3 (t2)
lw      t4 4(t2)
lw      t5 -4(t2)
```

Результаты компиляции:

```
Address    Code        Basic                       Source

0x00400000  0x0fc10317  auipc x6,0x0000fc10   6   lw      t1 var
0x00400004  0x00432303  lw x6,0x00000004(x6)
0x00400008  0x0fc10397  auipc x7,0x0000fc10   7   lw      t2 addr
0x0040000c  0x0003a383  lw x7,0x00000000(x7)
0x00400010  0x0003ae03  lw x28,0x00000000(x7) 8   lw      t3 (t2)
0x00400014  0x0043ae83  lw x29,0x00000004(x7) 9   lw      t4 4(t2)
0x00400018  0xffc3af03  lw x30,0xfffffffc(x7) 10  lw      t5 -4(t2)
```

- Инструкция `auipc` формирует в регистре `t1` (он же `x6`) адрес, по которому лежит интересующее нас значение.
- далее `lw` выбирает это значение из памяти, попутно скорректировав его смещением 4, и кладёт в тот же регистр `t1`.
- По метке `addr` размещается метак `var`, то есть адрес `0x10010004`.
- Он оказывается в регистре `t2` тем же способом, каким `0xdeadbeef` оказалось в `t1`.

- После этого с помощью явно указанного смещения в инструкции `lw` получаем в разных регистрах содержимое памяти по адресам `0x10010004`, `0x10010008` и `0x10010000` сооответственно.

Косвенная адресация — единственный способ обработки массива. Массив — это адрес в памяти и длина (количество элементов * размер одного элемента). В примере массив слов расписывается последовательными значениями:

```
.data
array:  .space  64
arrend:
.text
la      t0 array
la      t1 arrend
li      t2 1
loop:   sw      t2 (t0)
addi    t2 t2 1
addi    t0 t0 4
bltu    t0 t1 loop
li      a7 10           # Останов
ecall
```

Пояснить каким образом формируется адрес текущего элемента массива. Также акцентировать внимание, что остановка осуществляет не по числу элементов в массиве (что тоже возможно), а по достижению адреса `arrend`, фиксирующего ячейку после завершения массива.

3.2.2 Моделирование на ассемблере различных операторов управления

Учитывая, что основные элементы и команды управления уже показаны, можно кратко остановиться на примерах имитации базовых операторов управления языков высокого уровня. Думаю, что для понимания достаточно их прогона. Тоже будут в LMS.

Оператор if

```
# if (t0 == 0) {
    #     t1 = 1;
    # } else if (t0 < 0) {
    #     t1 = 2;
    # } else if (t0 >= 10) {
    #     t1 = 3;
    # } else {
    #     t1 = 4;
```

```
    # }
main:
li   a7, 5
ecall
mv   t0, a0
if_0:
bnez t0, if_less_0
li   t1, 1
j    end_if
if_less_0:
bgtz t0, if_greater_10
li   t1, 2
j    end_if
if_greater_10:
li   t3, 10
ble  t0, t3, else
li   t1, 3
j    end_if
else:
li   t1, 4
end_if:
li   a7, 1
mv   a0, t1
ecall
```

Оператор while

```
# while((t0 = read_int()) != 0) {
    #    print_int(t0)
    #    print_char('\n')
    # }
while:
li   a7, 5
ecall
mv   t0, a0
beqz a0, end_while
li   a7, 1
ecall
li   a7, 11
li   a0, '\n'
ecall
j    while
end_while:
```

Оператор for

```
# for (t0 = 0; t0 < t1; ++t0) {
    #     print_int(t0)
    #     print_char('\n')
    # }
for:
li   a7, 5
ecall
mv   t1, a0
mv   t0, zero
next:
beq  t0, t1, end_for
mv   a0, t0
li   a7, 1
ecall
li   a7, 11
li   a0, '\n'
ecall
addi t0, t0, 1
j    next
end_for:
```

3.3 Имитация в эмуляторе системных вызовов. Аналогии системных вызовов в операционных системах

Разговор о том, что системные вызовы эмулятора отличаются от системных вызовов ОС идет постоянно, начиная с первого семинара. Поэтому в ходе этого и последующих семинаров дополнительно стоит пояснять только новые вызовы и отсылать к системе помощи эмулятора, в которой хорошо описаны принципы их работы. Поэтому здесь особо акцентироваться не на чем. Только повторять ранее сказанное, избегая при этом занудства... Может быть не имеет смысла держать этот пункт в теме.

3.4 Примеры простых целочисленных алгоритмов

После рассмотрения команд управления можно рассмотреть пару простых алгоритмов. Ниже представлены в качестве примеров нахождение числа Фибоначчи и вычисление наибольшего общего делителя двух чисел (алгоритм Евклида). В целом заострять внимание на них сильно не стоит в связи с очевидностью решений. Но прогнать их студентам имеет смысл. Возможно, с разными исходным данным,

чтобы показать отсутствие обработки некорректных входных данных. Программы также будут находиться в LMS.

3.4.1 Вычисление числа Фибоначчи

```
#
# Example that calculates the Fibonacci sequence.
#
main:
mv   t0, zero
li   t1, 1

li   a7, 5
ecall
mv   t3, a0
fib:
beqz t3, finish
add  t2, t1, t0
mv   t0, t1
mv   t1, t2
addi t3, t3, -1
j    fib
finish:
li   a7, 1
mv   a0, t0
ecall
```

3.4.2 Алгоритм Евклида

```
# Calculates the greatest common divisor of
# two values using the Euclidean algorithm.
# int gcd(int a, int b) {
    #    while (a != b)
    #        if a > b
    #            a = a - b;
    #        else
    #            b := b - a;
    #    return a;
    # }

.data
arg01:  .asciz "Input 1st number: "
arg02:  .asciz "Input 2nd number: "
result: .asciz "Result = "
```

```
ln:     .asciz "\n"

.text
main:
la  a0, arg01   # Подсказка для ввода первого числа
li  a7, 4       # Системный вызов №4. Ввод строки в стиле Си
ecall

li  a7, 5       # Системный вызов №5. Ввод целого
ecall
mv  t1, a0

la  a0, arg02   # Подсказка для ввода второго числа
li  a7, 4       # Системный вызов №4. Ввод строки в стиле Си
ecall

li  a7, 5
ecall
mv  t2, a0

loop:
beq t1, t2, finish

slt t0, t1, t2
bne t0, zero, if_less

sub t1, t1, t2
b   loop

if_less:
sub t2, t2, t1
b   loop

finish:
la a0, result   # Подсказка для выводимого результата
li a7, 4        # Системный вызов №4
ecall

li  a7, 1  # Системный вызов №1 - вывести десятичное число
mv  a0, t1
ecall

la  a0, ln      # Перевод строки
li  a7, 4       # Системный вызов №4
```

```
ecall

li      a7 10   # Системный вызов №10 - останов программы
ecall
```

3.5 Домашнее задание

3.5.1 До 10 баллов

Разработать на ассемблере RARS программу, осуществляющую целочисленное деление для 32-разрядных целых чисел со знаком, используя операции вычитания, ветвления и циклы. Исходные делимое и делитель вводятся с клавиатуры в десятичной системе счисления. Полученные в результате деления частное и остаток необходимо вывести на консоль эмулятора. Остаток от деления вычисляется по правилам, используемых при выполнения операции вычисления остатка (`%`) в языках программирования `C/C++`. При делении учитывать знаки операндов и результатов, а также возможность ошибок при делении на ноль.

В отчете привести примеры скриншотов консоли, демонстрирующие все возможные комбинации тестового покрытия.

Рекомендации. Предварительно данную программу можно отработать на языках более высокого уровня (рекомендуется использовать C/C++).

4 Семинар 04. Целочисленная арифметика. Одномерные и многомерные массивы. Простые алгоритмы

Целью семинара изучение команд, обеспечивающих обработку целочисленных данных со знаком и без знака, организацию управления в программах, использующих целочисленную арифметику. Распределение памяти под целочисленные одномерные и многомерные массивы, организации доступа к элементам массивов различной размерности.

Проведение семинара предполагается по следующему плану:

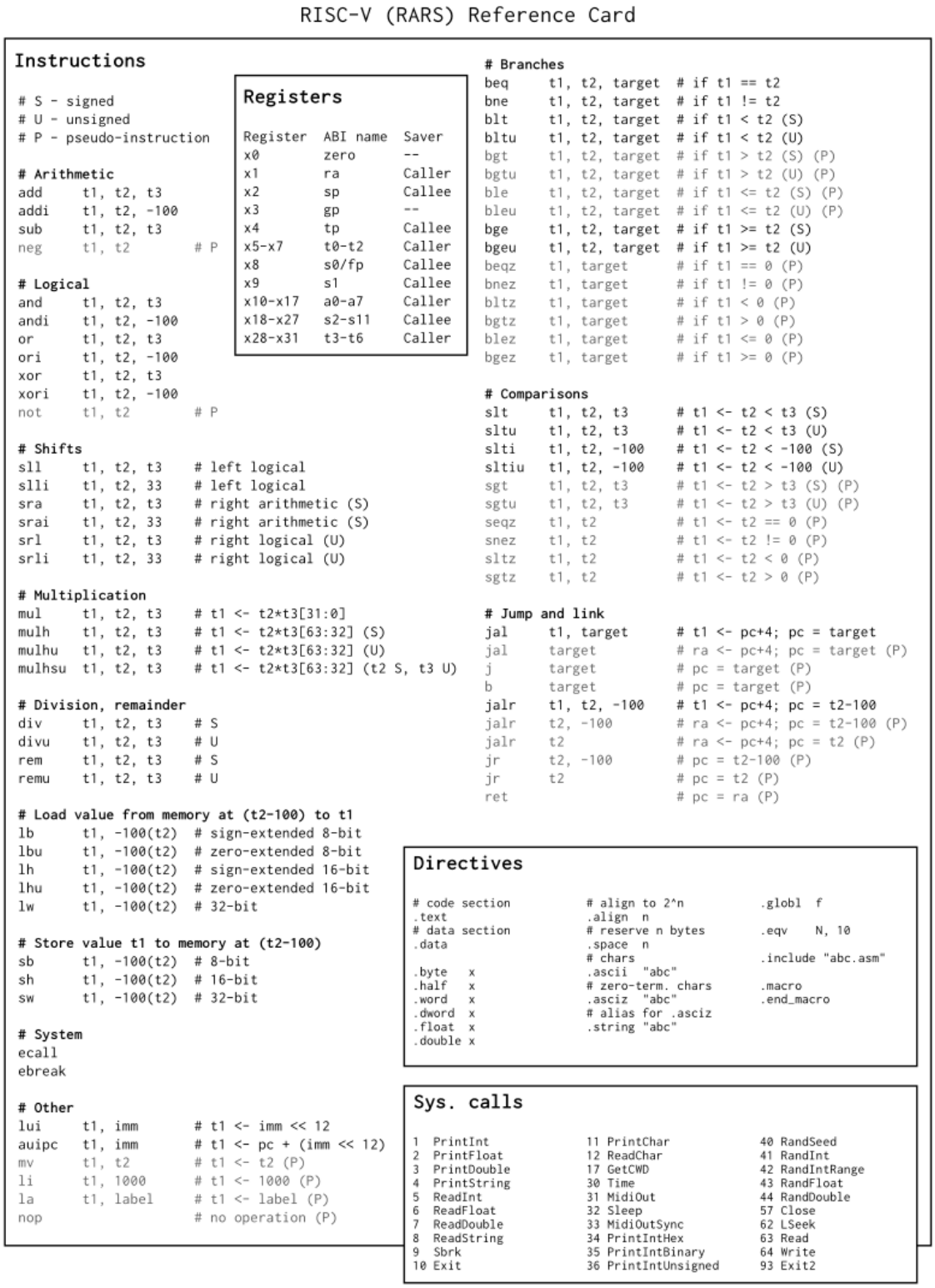
1. Обзор команд, обеспечивающих поддержку целочисленной арифметики.
2. Организация в памяти одномерных массивов и методы доступа к этим массивам.
3. Примеры работы с одномерными массивами.

4.1 Сценарий семинара

4.1.1 Обзор команд, обеспечивающих поддержку целочисленной арифметики

Предлагается кратко охарактеризовать список основных и дополнительных команд процессора, реализующих базовую арифметику и логику для 32-х разрядной архитектуры. В конспекте лекций я их разбил на отдельные небольшие таблицы. Думаю, что после завершения перевода они перенесутся в методу для самостоятельной работы (сами списки взяты из системы помощи RARS).

Но в целом понятно, что на все эти команды разом обращать внимание бесполезно поэтому проще продемонстрировать те схемы, которые в сокращенном виде представляют команды процессора. Одна из них — это список команд в эмуляторе RARS (рисунок 4.1). При необходимости их описание можно прочитать в системе помощи эмулятора.



RISC-V (RARS) Reference Card

Instructions

```
# S - signed
# U - unsigned
# P - pseudo-instruction

# Arithmetic
add     t1, t2, t3
addi    t1, t2, -100
sub     t1, t2, t3
neg     t1, t2          # P

# Logical
and     t1, t2, t3
andi    t1, t2, -100
or      t1, t2, t3
ori     t1, t2, -100
xor     t1, t2, t3
xori    t1, t2, -100
not     t1, t2          # P

# Shifts
sll     t1, t2, t3      # left logical
slli    t1, t2, 33      # left logical
sra     t1, t2, t3      # right arithmetic (S)
srai    t1, t2, 33      # right arithmetic (S)
srl     t1, t2, t3      # right logical (U)
srli    t1, t2, 33      # right logical (U)

# Multiplication
mul     t1, t2, t3      # t1 <- t2*t3[31:0]
mulh    t1, t2, t3      # t1 <- t2*t3[63:32] (S)
mulhu   t1, t2, t3      # t1 <- t2*t3[63:32] (U)
mulhsu  t1, t2, t3      # t1 <- t2*t3[63:32] (t2 S, t3 U)

# Division, remainder
div     t1, t2, t3      # S
divu    t1, t2, t3      # U
rem     t1, t2, t3      # S
remu    t1, t2, t3      # U

# Load value from memory at (t2-100) to t1
lb      t1, -100(t2)  # sign-extended 8-bit
lbu     t1, -100(t2)  # zero-extended 8-bit
lh      t1, -100(t2)  # sign-extended 16-bit
lhu     t1, -100(t2)  # zero-extended 16-bit
lw      t1, -100(t2)  # 32-bit

# Store value t1 to memory at (t2-100)
sb      t1, -100(t2)  # 8-bit
sh      t1, -100(t2)  # 16-bit
sw      t1, -100(t2)  # 32-bit

# System
ecall
ebreak

# Other
lui     t1, imm         # t1 <- imm << 12
auipc   t1, imm         # t1 <- pc + (imm << 12)
mv      t1, t2          # t1 <- t2 (P)
li      t1, 1000        # t1 <- 1000 (P)
la      t1, label       # t1 <- label (P)
nop                     # no operation (P)
```

Registers

Register	ABI name	Saver
x0	zero	--
x1	ra	Caller
x2	sp	Callee
x3	gp	--
x4	tp	Callee
x5-x7	t0-t2	Caller
x8	s0/fp	Callee
x9	s1	Callee
x10-x17	a0-a7	Caller
x18-x27	s2-s11	Callee
x28-x31	t3-t6	Caller

```
# Branches
beq     t1, t2, target  # if t1 == t2
bne     t1, t2, target  # if t1 != t2
blt     t1, t2, target  # if t1 < t2 (S)
bltu    t1, t2, target  # if t1 < t2 (U)
bgt     t1, t2, target  # if t1 > t2 (S) (P)
bgtu    t1, t2, target  # if t1 > t2 (U) (P)
ble     t1, t2, target  # if t1 <= t2 (S) (P)
bleu    t1, t2, target  # if t1 <= t2 (U) (P)
bge     t1, t2, target  # if t1 >= t2 (S)
bgeu    t1, t2, target  # if t1 >= t2 (U)
beqz    t1, target      # if t1 == 0 (P)
bnez    t1, target      # if t1 != 0 (P)
bltz    t1, target      # if t1 < 0 (P)
bgtz    t1, target      # if t1 > 0 (P)
blez    t1, target      # if t1 <= 0 (P)
bgez    t1, target      # if t1 >= 0 (P)

# Comparisons
slt     t1, t2, t3      # t1 <- t2 < t3 (S)
sltu    t1, t2, t3      # t1 <- t2 < t3 (U)
slti    t1, t2, -100    # t1 <- t2 < -100 (S)
sltiu   t1, t2, -100    # t1 <- t2 < -100 (U)
sgt     t1, t2, t3      # t1 <- t2 > t3 (S) (P)
sgtu    t1, t2, t3      # t1 <- t2 > t3 (U) (P)
seqz    t1, t2          # t1 <- t2 == 0 (P)
snez    t1, t2          # t1 <- t2 != 0 (P)
sltz    t1, t2          # t1 <- t2 < 0 (P)
sgtz    t1, t2          # t1 <- t2 > 0 (P)

# Jump and link
jal     t1, target      # t1 <- pc+4; pc = target
jal     target          # ra <- pc+4; pc = target (P)
j       target          # pc = target (P)
b       target          # pc = target (P)
jalr    t1, t2, -100    # t1 <- pc+4; pc = t2-100
jalr    t2, -100        # ra <- pc+4; pc = t2-100 (P)
jalr    t2              # ra <- pc+4; pc = t2 (P)
jr      t2, -100        # pc = t2-100 (P)
jr      t2              # pc = t2 (P)
ret                     # pc = ra (P)
```

Directives

```
# code section          # align to 2^n          .globl  f
.text                   .align  n
# data section          # reserve n bytes       .eqv    N, 10
.data                   .space  n
                        # chars                 .include "abc.asm"
.byte   x               .ascii  "abc"
.half   x               # zero-term. chars      .macro
.word   x               .asciz  "abc"           .end_macro
.dword  x               # alias for .asciz
.float  x               .string "abc"
.double x
```

Sys. calls

```
1  PrintInt             11 PrintChar            40 RandSeed
2  PrintFloat           12 ReadChar             41 RandInt
3  PrintDouble          17 GetCWD               42 RandIntRange
4  PrintString          30 Time                 43 RandFloat
5  ReadInt              31 MidiOut              44 RandDouble
6  ReadFloat            32 Sleep                57 Close
7  ReadDouble           33 MidiOutSync          62 LSeek
8  ReadString           34 PrintIntHex          63 Read
9  Sbrk                 35 PrintIntBinary       64 Write
10 Exit                 36 PrintIntUnsigned     93 Exit2
```

Рисунок 4.1 — Команды 32-разрядного процессора RISC-V, реализованные в эмуляторе RARS

Другая схема дает более полное описание уже не только RV32I, но и других вариантов. Но там легко выделить изучаемый процессор. И возможно, что она является более наглядной. Занимает, правда две страницы (рисунок 4.2 и рисунок 4.3). Основное достоинство — более четко расписаны операнды и приводятся типы форматов команд. Но в целом на больше будет интересовать первый документ.

Обе схемы выложены в LMS. Поэтому при необходимости их можно будет скачать. В принципе на второй схем можно пояснить, что команды целочисленного умножения и деления являются для 32-х разрядной архитектуры опциональными (не знаю, насколько это важно).

Основная идея — сказать, чтобы не пугались, что их много. Не все будут нужны и не сразу, а постепенно. Главное - ориентироваться в базовом списке в основных группах и при необходимости читать о них в системе помощи.

Лалее по ходу объяснения можно обращаться к различным группам команд, когда потребуются различия в представлении:

- знаковых и беззнаковых чисел;
- байтов, слов, двойных слов;
- и т.д.

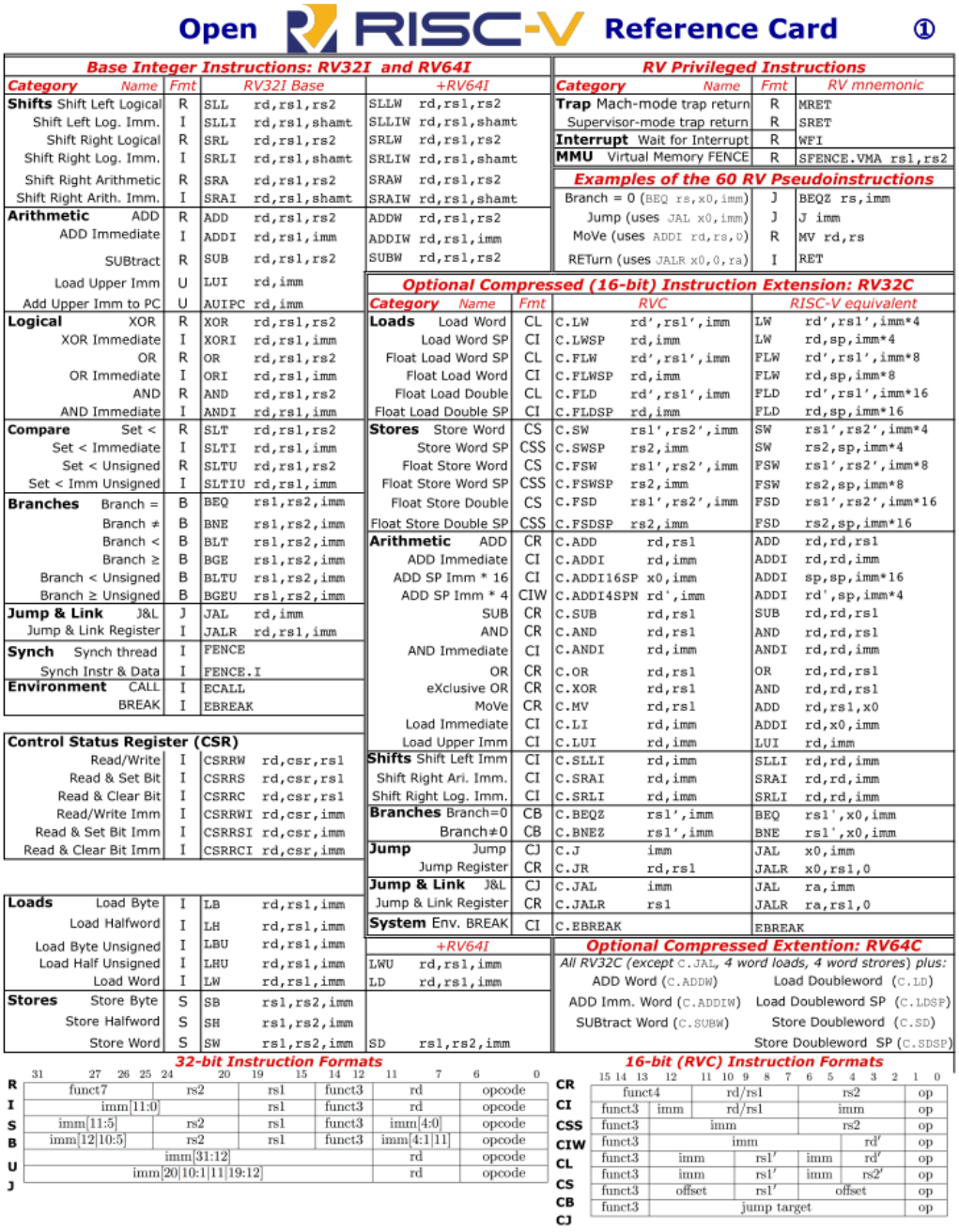
4.2 Организация в памяти одномерных массивов и методы доступа к этим массивам

Изначально можно представить простую программу на Си, в которой, как и в последующей ассемблерной решается задача формирования и вывода элементов целочисленного массива:

```
#include <stdio.h>

int array[16];

int main()
{
    fill:
    for(int i = 0; i < 16; ++i) {
        array[i] = i+1;
    }
    printf("--------\n");
    out:
    for(int i = 0; i < 16; ++i) {
        printf("%d\n", array[i]);
    }
    return 0;
}
```



RISC-V Integer Base (RV32I/64I), privileged, and optional RV32/64C. Registers `x1`-`x31` and the `PC` are 32 bits wide in RV32I and 64 in RV64I (`x0`=0). RV64I adds 12 instructions for the wider data. Every 16-bit RVC instruction maps to an existing 32-bit RISC-V instruction.

Рисунок 4.2 — Команды процессора RISC-V (начало)

Open RISC-V Reference Card ②

Optional Multiply-Divide Instruction Extension: RVM

Category	Name	Fmt	RV32M (Multiply-Divide)		+RV64M	
Multiply	MULtiply	R	MUL	rd,rs1,rs2	MULW	rd,rs1,rs2
	MULtiply High	R	MULH	rd,rs1,rs2		
	MULtiply High Sign/Uns	R	MULHSU	rd,rs1,rs2		
	MULtiply High Uns	R	MULHU	rd,rs1,rs2		
Divide	DIVide	R	DIV	rd,rs1,rs2	DIVW	rd,rs1,rs2
	DIVide Unsigned	R	DIVU	rd,rs1,rs2		
Remainder	REMainder	R	REM	rd,rs1,rs2	REMW	rd,rs1,rs2
	REMainder Unsigned	R	REMU	rd,rs1,rs2	REMUW	rd,rs1,rs2

Optional Atomic Instruction Extension: RVA

Category	Name	Fmt	RV32A (Atomic)		+RV64A	
Load	Load Reserved	R	LR.W	rd,rs1	LR.D	rd,rs1
Store	Store Conditional	R	SC.W	rd,rs1,rs2	SC.D	rd,rs1,rs2
Swap	SWAP	R	AMOSWAP.W	rd,rs1,rs2	AMOSWAP.D	rd,rs1,rs2
Add	ADD	R	AMOADD.W	rd,rs1,rs2	AMOADD.D	rd,rs1,rs2
Logical	XOR	R	AMOXOR.W	rd,rs1,rs2	AMOXOR.D	rd,rs1,rs2
	AND	R	AMOAND.W	rd,rs1,rs2	AMOAND.D	rd,rs1,rs2
	OR	R	AMOOR.W	rd,rs1,rs2	AMOOR.D	rd,rs1,rs2
Min/Max	MINimum	R	AMOMIN.W	rd,rs1,rs2	AMOMIN.D	rd,rs1,rs2
	MAXimum	R	AMOMAX.W	rd,rs1,rs2	AMOMAX.D	rd,rs1,rs2
	MINimum Unsigned	R	AMOMINU.W	rd,rs1,rs2	AMOMINU.D	rd,rs1,rs2
	MAXimum Unsigned	R	AMOMAXU.W	rd,rs1,rs2	AMOMAXU.D	rd,rs1,rs2

Two Optional Floating-Point Instruction Extensions: RVF & RVD

Category	Name	Fmt	RV32{F|D} (SP,DP Fl. Pt.)		+RV64{F|D}	
Move	Move from Integer	R	FMV.W.X	rd,rs1	FMV.D.X	rd,rs1
	Move to Integer	R	FMV.X.W	rd,rs1	FMV.X.D	rd,rs1
Convert	ConVerT from Int	R	FCVT.{S|D}.W	rd,rs1	FCVT.{S|D}.L	rd,rs1
	ConVerT from Int Unsigned	R	FCVT.{S|D}.WU	rd,rs1	FCVT.{S|D}.LU	rd,rs1
	ConVerT to Int	R	FCVT.W.{S|D}	rd,rs1	FCVT.L.{S|D}	rd,rs1
	ConVerT to Int Unsigned	R	FCVT.WU.{S|D}	rd,rs1	FCVT.LU.{S|D}	rd,rs1
Load	Load	I	FL{W,D}	rd,rs1,imm		
Store	Store	S	FS{W,D}	rs1,rs2,imm		
Arithmetic	ADD	R	FADD.{S|D}	rd,rs1,rs2		
	SUBtract	R	FSUB.{S|D}	rd,rs1,rs2		
	MULtiply	R	FMUL.{S|D}	rd,rs1,rs2		
	DIVide	R	FDIV.{S|D}	rd,rs1,rs2		
	SQuare RooT	R	FSQRT.{S|D}	rd,rs1		
Mul-Add	Multiply-ADD	R	FMADD.{S|D}	rd,rs1,rs2,rs3		
	Multiply-SUBtract	R	FMSUB.{S|D}	rd,rs1,rs2,rs3		
	Negative Multiply-SUBtract	R	FNMSUB.{S|D}	rd,rs1,rs2,rs3		
	Negative Multiply-ADD	R	FNMADD.{S|D}	rd,rs1,rs2,rs3		
Sign Inject	SiGN source	R	FSGNJ.{S|D}	rd,rs1,rs2		
	Negative SiGN source	R	FSGNJN.{S|D}	rd,rs1,rs2		
	Xor SiGN source	R	FSGNJX.{S|D}	rd,rs1,rs2		
Min/Max	MINimum	R	FMIN.{S|D}	rd,rs1,rs2		
	MAXimum	R	FMAX.{S|D}	rd,rs1,rs2		
Compare	compare Float =	R	FEQ.{S|D}	rd,rs1,rs2		
	compare Float <	R	FLT.{S|D}	rd,rs1,rs2		
	compare Float ≤	R	FLE.{S|D}	rd,rs1,rs2		
Categorize	CLASSify type	R	FCLASS.{S|D}	rd,rs1		
Configure	Read Status	R	FRCSR	rd		
	Read Rounding Mode	R	FRRM	rd		
	Read Flags	R	FRFLAGS	rd		
	Swap Status Reg	R	FSCSR	rd,rs1		
	Swap Rounding Mode	R	FSRM	rd,rs1		
	Swap Flags	R	FSFLAGS	rd,rs1		
	Swap Rounding Mode Imm	I	FSRMI	rd,imm		
	Swap Flags Imm	I	FSFLAGSI	rd,imm		

Calling Convention

Register	ABI Name	Saver
x0	zero	---
x1	ra	Caller
x2	sp	Callee
x3	gp	---
x4	tp	---
x5-7	t0-2	Caller
x8	s0/fp	Callee
x9	s1	Callee
x10-11	a0-1	Caller
x12-17	a2-7	Caller
x18-27	s2-11	Callee
x28-31	t3-t6	Caller
f0-7	ft0-7	Caller
f8-9	fs0-1	Callee
f10-11	fa0-1	Caller
f12-17	fa2-7	Caller
f18-27	fs2-11	Callee
f28-31	ft8-11	Caller
zero	Hardwired zero	
ra	Return address	
sp	Stack pointer	
gp	Global pointer	
tp	Thread pointer	
t0-6,ft0-11	Temporaries	
s0-11,fs0-11	Saved registers	
a0-7,fa0-7	Function args	

Optional Vector Extension: RVV

Name	Fmt	RV32V/R64V	
SET Vector Len.	R	SETVL	rd,rs1
MULtiply High	R	VMULH	rd,rs1,rs2
REMainder	R	VREM	rd,rs1,rs2
Shift Left Log.	R	VSLL	rd,rs1,rs2
Shift Right Log.	R	VSRL	rd,rs1,rs2
Shift R. Arith.	R	VSRA	rd,rs1,rs2
LoaD	I	VLD	rd,rs1,imm
LoaD Strided	R	VLDS	rd,rs1,rs2
LoaD indeXed	R	VLDX	rd,rs1,rs2
STore	S	VST	rd,rs1,imm
STore Strided	R	VSTS	rd,rs1,rs2
STore indeXed	R	VSTX	rd,rs1,rs2
AMO SWAP	R	AMOSWAP	rd,rs1,rs2
AMO ADD	R	AMOADD	rd,rs1,rs2
AMO XOR	R	AMOXOR	rd,rs1,rs2
AMO AND	R	AMOAND	rd,rs1,rs2
AMO OR	R	AMOOR	rd,rs1,rs2
AMO MINimum	R	AMOMIN	rd,rs1,rs2
AMO MAXimum	R	AMOMAX	rd,rs1,rs2
Predicate =	R	VPEQ	rd,rs1,rs2
Predicate ≠	R	VPNE	rd,rs1,rs2
Predicate <	R	VPLT	rd,rs1,rs2
Predicate ≥	R	VPGE	rd,rs1,rs2
Predicate AND	R	VPAND	rd,rs1,rs2
Pred. AND NOT	R	VPANDN	rd,rs1,rs2
Predicate OR	R	VPOR	rd,rs1,rs2
Predicate XOR	R	VPXOR	rd,rs1,rs2
Predicate NOT	R	VPNOT	rd,rs1
Pred. SWAP	R	VPSWAP	rd,rs1
MOVe	R	VMOV	rd,rs1
ConVerT	R	VCVT	rd,rs1
ADD	R	VADD	rd,rs1,rs2
SUBtract	R	VSUB	rd,rs1,rs2
MULtiply	R	VMUL	rd,rs1,rs2
DIVide	R	VDIV	rd,rs1,rs2
SQuare RooT	R	VSQRT	rd,rs1,rs2
Multiply-ADD	R	VFMADD	rd,rs1,rs2,rs3
Multiply-SUB	R	VFMSUB	rd,rs1,rs2,rs3
Neg. Mul.-SUB	R	VFNMSUB	rd,rs1,rs2,rs3
Neg. Mul.-ADD	R	VFNMADD	rd,rs1,rs2,rs3
SiGN inJect	R	VSGNJ	rd,rs1,rs2
Neg SiGN inJect	R	VSGNJN	rd,rs1,rs2
Xor SiGN inJect	R	VSGNJX	rd,rs1,rs2
MINimum	R	VMIN	rd,rs1,rs2
MAXimum	R	VMAX	rd,rs1,rs2
XOR	R	VXOR	rd,rs1,rs2
OR	R	VOR	rd,rs1,rs2
AND	R	VAND	rd,rs1,rs2
CLASS	R	VCLASS	rd,rs1
SET Data Conf.	R	VSETDCFG	rd,rs1
EXTRACT	R	VEXTRACT	rd,rs1,rs2
MERGE	R	VMERGE	rd,rs1,rs2
SELECT	R	VSELECT	rd,rs1,rs2

RISC-V calling convention and five optional extensions: 8 RV32M; 11 RV32A; 34 floating-point instructions each for 32- and 64-bit data (RV32F, RV32D); and 53 RV32V. Using regex notation, {} means set, so `FADD.{F|D}` is both `FADD.F` and `FADD.D`. RV32{F|D} adds registers `f0-f31`, whose width matches the widest precision, and a floating-point control and status register `fcsr`. RV32V adds vector registers `v0-v31`, vector predicate registers `vp0-vp7`, and vector length register `vl`. RV64 adds a few instructions: RVM gets 4, RVA 11, RVF 6, RVD 6, and RVV 0.

Рисунок 4.3 — Команды процессора RISC-V (окончание)

4.2.1 Работа с одномерными массивами

Можно начать с пояснения того, что формирование индекса элемента массива как отдельно рассматриваемого целого числа, увеличивающегося на 1 в принципе возможно. Однако в ассемблере предпочтительнее манипулировать непосредственно с адресным пространством нужной величины, а не умножать каждый раз на величину слова, так как операция умножения затратна по времени. Она даже не входит в базовый набор. Хотя можно ислпользовать сдвиги вместо умножения. Но все равно это лишние команды (но при желании пусть используют). Основной режим работы - использование косвенной адресации с индексацией относительно начала массива.

Показать, что для выделения массива достаточно задать память, выделяемую под него различными способами:

- можно выделить пространство требуемого размера;
- при наличии заранее предопределенных значений элементов можно их перечислить;
- при неизвестном числе элементов можно зарезервировать некоторый большой кусок памяти, а количество элементов задавать числом, меньшим выделенного размера, которое (как и впрограмме на Си) может служить окраничителем цикла при формировании массива;
- можно выделить память под массив на куче после получения числа элементов в массиве.

В качестве стартового примера можно опять обратиться к лекциям Г. Курячего:

```
.data
sep:    .asciz  "--------\n"    # Строка-разделитель (с \n и нулём в конце)
.align  2                       # Выравнивание на границу слова
array:  .space  64              # 64 байта
arrend:                         # Граница массива
.text
la      t0 array                # Счётчик
la      s1 arrend
li      t2 1            # Число, которое мы будем записывать в массив
fill:   sw      t2 (t0)         # Запись числа по адресу в t0
addi    t2 t2 1         # Изменим число
addi    t0 t0 4         # Увеличим адрес на размер слова в байтах
bltu    t0 s1 fill      # Если не вышли за границу массива
la      a0 sep          # Выведем строку-разделитель
li      a7 4
ecall
la      t0 array
out:    li      a7 1
```

```
lw      a0 (t0)         # Выведем очередной элемент массива
ecall
li      a7 11           # Выведем перевод строки
li      a0 10
ecall
addi    t0 t0 4
blt     t0 s1 out
li      a7 10           # Останов
ecall
```

На основе этого примера можно пояснить, что при выделении под массив, который заполняется полностью, можно не использовать его размерность, а указывать метку, задающую ограничительный адрес.

На примере этой программы можно предложить модификации на занятиях, в которых заполнение осуществляется побайтно и по 16 разрядным словам (эти примеры лежать не будут, студенты должны сделать самостоятельно). При этом следует откорректировать не только команду загрузки в память, но и чтения из нее (при выводе).

Пример для генерации байтового массива:

```
.data
sep:    .asciz  "--------\n"    # Строка-разделитель (с \n и нулём в конце)
.align  2                       # Выравнивание на границу слова
array:  .space  64              # 64 байта
arrend:                         # Граница массива
.text
la      t0 array                # Счётчик
la      s1 arrend
li      t2 1            # Число, которое мы будем записывать в массив
fill:   sb      t2 (t0)         # Запись числа по адресу в t0
addi    t2 t2 1         # Изменим число
addi    t0 t0 1         # Увеличим адрес на размер слова в байтах
bltu    t0 s1 fill      # Если не вышли за границу массива
la      a0 sep          # Выведем строку-разделитель
li      a7 4
ecall
la      t0 array
out:    li      a7 1
lb      a0 (t0)         # Выведем очередной элемент массива
ecall
li      a7 11           # Выведем перевод строки
li      a0 10
ecall
addi    t0 t0 1
blt     t0 s1 out
```

```
    li      a7 10           # Останов
    ecall
```

Аналогично можно быстро сделать модификацию и для 16-разрядных слов, изменяя только команды чтения и записи в память.

Помимо этого можно предложиеть модифицировать первую программу так, чтобы вводить в память ограниченное число элементов, заданных максимально допустимым числом. По примеру следующей программы на Си:

```
#include <stdio.h>

int array[16];
int n;

int main()
{
    in:
    printf("n = ? ");
    scanf("%d", &n);
    if(n > 16 || n < 1) {
        printf("n = %d is incorrect!\n", n);
        return 1;
    }
    fill:
    for(int i = 0; i < n; ++i) {
        array[i] = i;
    }
    printf("--------\n");
    out:
    for(int i = 0; i < n; ++i) {
        printf("%d\n", array[i]);
    }
    return 0;
}
```

В этом случае может получиться примерно такое (но необязательно одинаково, так как возможны варианты):

```
    .data
    sep:    .asciz  "--------\n"    # Строка-разделитель (с \n и нулём в конце)
    prompt: .asciz  "n = ? "         # Подсказка для ввода числа
    error:  .asciz  "incorrect n!\n"  # Сообщение о некорректном вводе
    .align  2                       # Выравнивание на границу слова
    n: .word 0 # Число введенных элементов массива
    array:  .space  64              # 64 байта
```

```
.text
in:
la  a0, prompt      # Подсказка для ввода числа элементов массива
li  a7, 4           # Системный вызов №4
ecall
li      a7 5            # Системный вызов №5 - ввести десятичное число
ecall
mv      t3 a0           # Сохраняем результат в t3 (это n)
ble     t3 zero fail    # На ошибку, если меньше 0
li      t4 16           # Размер массива
bgt     t3 t4 fail      # На ошибку, если больше 16
la t4 n # Адрес n в t4
sw t3 (t4) # Загрузка n в память на хранение

la      t0 array        # Указатель элемента массива
li      t2 1            # Число, которое мы будем записывать в массив
fill:   sw      t2 (t0)         # Запись числа по адресу в t0
addi    t2 t2 1         # Изменим число
addi    t0 t0 4         # Увеличим адрес на размер слова в байтах
addi    t3 t3 -1        # Уменьшим количество оставшихся элементов на 1
bnez    t3 fill         # Если осталось больш 0
# bltu    t0 s1 fill      # Если не вышли за границу массива
la      a0 sep          # Выведем строку-разделитель
li      a7 4
ecall

lw t3 n # Число элементов массива
la      t0 array
out:    li      a7 1
lw      a0 (t0)         # Выведем очередной элемент массива
ecall
li      a0 '\n'         # Выведем перевод строки
li      a7 11
ecall
addi    t0 t0 4
addi    t3 t3 -1        # Уменьшим количество оставшихся элементов на 1
bnez    t3 out          # Если осталось больше 0
# blt     t0 s1 out
li      a7 10           # Останов
ecall
fail:
la  a0, error       # Сообщение об ошибке ввода числа элементов массива
li  a7, 4           # Системный вызов №4
ecall
```

```
li      a7 10           # Останов
ecall
```

Примечание.: На самом деле у меня на отработку этой простой программы ушло немало времени, включая отладку. Возможно стоит, после небольшой паузы, просто продемонстрировать ее решение. Нужно прикинуть хронометраж. Также решил убрать работу с многомерными массивами. Их не будет в задании. Вряд ли на стоит заострять внимание.

4.3 Домашнее задание

4.3.1 До 8 баллов

Написать программу, осуществляющую суммирование целочисленных элементов одномерного массива. Количество элементов в массиве может варьироваться от 1 до 10. Сами элементы вводятся с клавиатуры. Значение суммы также выводится в консоль эмулятора RARS. Необходимо контролировать, чтобы число вводимых элементов не превышало максимально допустимое. В случае, когда возникает переполнение, необходимо вывести последнее корректное значение суммы и число просуммированных при этом элементов. Суммирование осуществлять после размещения массива в памяти.

4.3.2 Опционально до +2 баллов

Подсчитать количество четных и нечетных элементов в обрабатываемом массиве. Подсчет осуществлять в массиве, который уже расположен в памяти.

5 Семинар 05. Подпрограммы. Передача параметров

Целью семинара является изучение использования подпрограмм на уровне систмы команд. На занятии предполагается рассмотреть следующие темы:

1. Особенности вызова подпрограмм в системе команд RISC-V и возврата из подпрограмм.
2. Использование подпрограмм без параметров и с параметрами. Достоинства и недостатки. Примеры.
3. Соглашения о передаче фактических параметров и возврате результатов в архитектуре RISC-V.

5.1 Особенности вызова подпрограмм и возврата из подпрограмм

> **Примечание**: Данную тему в первой итерации предлагается заимствовать из Г. Курячего, что в общем-то уже и сделано. В дальнейшем, думаю, стоит уточнить примеры. Помимо этого вставить дополнительные более сложные примеры, раскрывающие особенности различных моментов использования подпрограмм. Все необходимые материалы уже сгруппированы в соответствующей презентации.

В целом изложение выглядит следующим образом:

- определение подпрограммы;
- отличие ассемблерных подпрограммы от подпрограмм, процедур, функций языков высокого уровня;
- описание подпрограмм в ассемблере процессора RISC-V (эмулятора RARS);
- команды и псевдокоманды вызова подпрограмм;
- команды и псевдокоманды возврата из подпрограмм

5.2 Использование подпрограмм и без параметров. Достоинства и недостатки. Примеры

Основной акцент сделать на примерах (изложены в презентациях). Показать на эмуляторе в пошаговом режиме, что происходит:

- пример использования вызовов `jal` и `jalr`;
- пример с отрезками, составляющими стороны треугольника (на нем можно показать изощренность возврата, что не всегда удобно в практическом решении из-за появления избыточных зависимостей);

5.3 Соглашения о передаче фактических параметров и возврате результатов

Рассказать об основных соглашениях, моделирующих передачу параметров внутрь подпрограмм, аналогичную использованию параметров в процедурах и функциях. Рассказать о соглашении использования регистров `a*` как для передачи параметров, так и возврата результатов (`a0`, `a1`). Лишний раз напомнить, для чего используются соглашения.

5.4 Выдача задания № 1

Судя по дыре в расписании учебных занятий, минисессия будет проходить с 30 октября. Думаю, что на выполнение первого задания можно выделить 3 полных недели, поставив дедлайн 15 октября 23:59. График заданий сформирован и выложен в ЛМС. Также уже выложены варианты и требования к выполнению заданий. Все это ляжет и в репозиторий. Генератор вариантов тоже лежит. В принципе всю информацию по вариантам можно выложить в чаты и зафиксировать в текущих ведомостях. Основное — объявить дефакто, что задание роздано. Я об этом также объявлю на лекции.

Для групп, которые в пятницу, распределить задания можно раньше. Хоть в понедельник, разослав им список номеров по группам. По почте каждой группе отдельно предпочтительнее. Я наверное тоже своей группе дополнительно вышлю, так как тогда будет документ, против которого возразить будет сложно.

5.5 Домашнее задание

Домашнее задание завязано на второй семинар по данной теме, который по сути является продолжением. Тема разбита и немного увеличена по содержанию из-за добавления семинарских занятий в этом году.

6 Семинар 06. Подпрограммы. Стек. Кадр стека. Параметры. Локальные переменные

Целью семинара является продолжение изучения подпрограмм на уровне систмы команд. Основной упор при этом делается на роль стека в современном программировании не только на ассемблере, но и в языках высокого уровня.

На занятии предполагается рассмотреть следующие темы:

1. Пояснение специфики использования стека и его организации в целом. Привязаки стека к памяти.
2. Особенности работы со стеком в RISC-V. Основные команды. Стек как хранилище, альтернативное общей памяти.
3. Использование стека для дополнительных фактических параметров.
4. Использование стека для локальных переменных подпрограммы и дополнительных фактических параметров.
5. Разработка рекурсивных подпрограмм.
6. Соглашения о вызовах подпрограмм, использовании регистров. Кадр стека.
7. Выдача индивидуального задания № 1. Установка сроков и регламента сдачи индивидуальных заданий.

6.1 Общая специфика использования стека

В принципе уже прописано в презентации. Скорее всего он также изучен в дисциплине "Алгоритмы и структуры данных". Поэтому сильно акцентироваться не стоит. Достаточно дать основные определения. Можно подчеркнуть, что стек на уровне архитектур ВС обычно представляет собой линейное пространство памяти. Оно может быть выделено в любом месте. Но чаще всего в верхних адресах. Стек может расти с нижних адресов вверх, хотя чаще реализуется таким образом, что с верхних адресов растет вниз навстречу куче.

Остановиться на специфике размещения стека в памяти эмулятора RARS.

6.2 Особенности работы со стеком в RISC-V

Пояснить, что в принципе для работы со стеком нет специальных команд и любой регистр может являться указателем на вершину стека. Вместе с тем, под указатель стека соглашением выделен регистр `x2`, который имеет псевдоним `sp`.

Далее на примере (уже есть в презентации) рассмотреть использования стека в пошаговом режиме. Посмотреть в эмуляторе, как изменяются регистры процессора и память, отведенная под стек.

Здесь же можно отметить, что на стеке обычно сохраняется и адрес возврата из подпрограммы, что позволяет организовать иерархические вложенные вызовы подпрограмм, а также обеспечить выполнение рекурсивных подпрограмм.

6.3 Использование стека для дополнительных фактических параметров

Рассказать, что если передаваемых в подпрограмму параметров много, и они не размещаются в регистрах, то обычно используется стек, в который эти параметры и ложатся.

6.4 Использование стека для локальных переменных подпрограммы

Аналогичным образом пояснить использовани и хранение на стеке локальных переменных. Можно также сказать (возможно еще раз), что это сохранение бывает более эффективным, чем в глобальной памяти за счет того, что поддерживает одной командой загрузки в стек или чтения из него по сравнению с обращением по абсолютному адресу.

6.5 Разработка рекурсивных подпрограмм

Показато на примерах работу с рекурсивными подпрограммами.

6.6 Соглашения о вызовах подпрограмм, использовании регистров. Кадр стека

Суммировать материал об основных соглашениях, связанных с организацией подпрограмм, ролевым распределением регистров, выделением регистров `s*` в качеств сохраняемых на стеке и т.д.

6.7 Домашнее задание

6.7.1 До 8 баллов

Разработать программу, определяющую максимальное значение аргумента, при котором результат вычисления факториала размещается в 32-х разрядном машинном слове. Вычисление факториала организовать как подпрограмму с циклом, которая возвращает найденный аргумент в регистре a0. Вывод результатов должна осуществлять главная функция.

6.7.2 Опционально до +2 баллов

Дополнительно реализовать решение предыдущей задачи с использованием рекурсивной подпрограммы вычисления максимального значения аргумента, при котором результат вычисления факториала размещается в 32-х разрядном машинном слове.

7 Семинар 07. Немного об ассемблере. Директивы. Макросы. Многофайловые программы

Целью семинара является более глубое погружение в разработку программа на ассемблере RARS для повышения эффективности выполнения последующих домашних и индивидуальных заданий.

> **AL**: Принцип тот же самый: не успеем рассмотреть все - переносим на следующий семинар. На самом деле погружение не такое уж глубокое, так как реальные ассемблеры позволяют больше и имеют более развитые средства отладки. Но в курсе по архитектурам ВС это не столь важно. Тем более, что имеющиеся дополнительные средства вполне очевидны и реально позволяют повысить эффективность программирования без изучения каких-либо новых инструментов. Что приходится делать в реальны условиях.

На занятии предполагается рассмотреть следующие темы:

1. Дополнительные директивы, повышающие эффективность написания кода.
2. Использование макросов.
3. Создание макробиблиотек.
4. Сочетание макросов и подпрограмм.
5. Создание многомодульных программ.

7.1 Дополнительные директивы, повышающие эффективность написания кода

К дополнительным директивам, облегчающим написание кода относятся директивы управления, псевдонимы (алиасы), директивы работы с макроопределениями.

7.1.1 Директивы управления

Директивы управления в основном предназначены для более эффективного управления памятью при написании программ. В большей степени они уже рассматривались. Ниже приведен их список с краткими пояснениями.

Таблица 7.1 описывает особенности распределения и использования регистров.

Таблица 7.1 — Директивы для управления памятью

Обозначение	Соглашения по использованию
`.align`	Align next data item on specified byte boundary (0=byte, 1=half, 2=word, 3=double)
`.ascii`	Store the string in the Data segment but do not add null terminator
`.asciz`	Store the string in the Data segment and add null terminator
`.byte`	Store the listed value(s) as 8 bit bytes
`.data`	Subsequent items stored in Data segment at next available address
`.double`	Store the listed value(s) as double precision floating point
`.dword`	Store the listed value(s) as 64 bit double-word on word boundary
s1	Сохраняемый регистр 1 (saved register 1)
`.eqv`	Substitute second operand for first. First operand is symbol, second operand is expression (like #define)
`.float`	Store the listed value(s) as single precision floating point
`.half`	Store the listed value(s) as 16 bit halfwords on halfword boundary
`.section`	Allows specifying sections without .text or .data directives. Included for gcc comparability
`.space`	Reserve the next specified number of bytes in Data segment
`.string`	Alias for .asciz
`.text`	Subsequent items (instructions) stored in Text segment at next available address
`.word`	Store the listed value(s) as 32 bit words on word boundary

Примечание.: Пока комментарии не переведены...

7.1.2 Псевдонимы (алиасы)

Псевдонимы обычно предназначены для подмены одного текста другим. Чаще всего заменяются константы на идентификаторы этих констант. В RARS для этого используется примитивный макрос, являющийся директивой `.eqv`. Она имеет следующий формат:

```
.eqv имя_псевдонима строка_заменяющая имя
```

В результате препроцессорной обработки (обработки текста перед трансляцией) происходит замена этого имени на строку:

7.1.3 Директивы для работы с макроопределениями

Макроподстановка — механизм поиска шаблона в тексте и замены его другим текстом. Полученный текст также может содержать шаблоны, так что процесс макроподстановки обычно рекурсивен.

Таблица 7.2 описывает директивы, используемые при создании макроопределений.

Таблица 7.2 — Директивы для создания макроопределений

Обозначение	Соглашения по использованию
`.end_macro`	End macro definition. See `.macro`
`.macro`	Begin macro definition. See `.end_macro`

7.1.4 Директивы для работы с многофайловыми программами

В реальных системах программирования программы собираются из множества модулей, которые хранятся в отдельных файлах, образуя проект. В целом это довольно сложные инструменты. В RARS создание многофайловых проектов решается намного проще.

Таблица 7.3 описывает директивы, используемые при создании многофайловых проектов.

Таблица 7.3 — Директивы для создания многофайловых проектов

Обозначение	Соглашения по использованию
`.extern`	Declare the listed label and byte length to be a global data field
`.global`	Declare the listed label(s) as globl to enable referencing from other files
`.globl`	Declare the listed label(s) as global to enable referencing from other files|
`.include`	Insert the contents of the specified file. Put filename in quotes.

7.2 Использование макросов

Для демонстрации использования макросов я реализовал сквозной пример на основе алгоритма Евклида. В обычном текстовом представлении он рассматривался и раньше. Его первая версия демонстрирует непосредственно добавление макросов ввода и вывода целых чисел, обертывающих системные вызовы, в файл с кодом основной программы. Пример показывает, как можно просто вставить изначально

макросы в программу и как в ней задаются описания параметров и осуществляется вызов макроса.

Текст программы находится в каталоге `euclid/euclid`.

7.3 Создание макробиблиотек

Следующий пример с той же самой программой, но макросы собраны в виде некоторой библиотеки. Библиотека на данном этапе заимствована у Г. Курячего. Помимо ввода данных добавлено использование макросов для вспомогательных сообщений. Соответствующее макроопределение демонстрирует локальное использование данных.

Текст программы находится в каталоге `euclid/euclid1`.

7.3.1 Макросы с локальными метками

Многократный вызов макросов требует разрешения конфликтов имен, которые могут повторяться при повторном к ним обращении. Это и метки внутри кода и метки внутри данных. В различны ассемблерах существуют различные подходы решению проблемы дублирования имен. В RARS сделано все просто. Уникальность имен обеспечивается добавлением суффикса `_Mi`. Не всегда это ведет к однозначности. Но в целом для нас достаточно. Другой вариант: можно имя метки задавать через параметр. Тоже не радикальное решение проблемы. Но кратко обсудить можно, используя примеры Курячего, если недостаточно примера с вычислением НОД.

7.4 Сочетание макросов и подпрограмм

Следующий пример демонстрирует использование макросов совместно с подпрограммами. Вычисление НОД выделяется в подпрограмму. В данном примере результат становится ошибочным, так как макросы используют те же регистры, что и подпрограмма.

Текст программы находится в каталоге `euclid/euclid2x`.

Предотвратить подобные конфликты в общем случае сложно, так как часто библиотеки макроопределений пишутся независимо от конкретных программ и, как в нашем случае, могут повторно использоваться. Следовательно их нужно изучать для организации правильного использования. Подобное изучение позволяет предотвратить конфликты по общим ресурсам (регистрам).

Если же регистры, занимаемые макросами, должны использоваться, то можно поступать как и с подпрограммами: сохранять их на стеке. В следующем примере как раз и вводится подобный прием. Перед вызовом конфликтующих макросов осуществляется сохранение в стеке. Для удобства дополнительно разработаны два макроса `push` и `pop`.

Текст программы находится в каталоге `euclid/euclid3`.

Однако не всегда известно, какие макросы и какие регистры используют. Поэтому целесообразно уже при их разработке предусмотреть (как и для подпрограмм) сохранение используемых регистров. В следующем примере сохранение на стеке в ряде макросов, используемых при вычислении НОД, непосредственно используется сохранение. Здесь тоже возможны ситуации, когда это ведет к кофликтам. Например при использовании регистра `a0` Старое значение из стека затирает введенное значение.

Текст программы находится в каталоге `euclid/euclid3`.

Поэтому в примере для работы с этим регистром в следующем примере выделен отдельный макрос. а ввод в любой другой регистр осуществляется с сохранением на стеке значения `a0`, которое может быть в дальнейшем полезно (как в данном примере).

Текст программы находится в каталоге `euclid/euclid4`.

И эта программа работает уже правильно

7.4.1 Проблемы макровзрыва и обертывание подпрограмм

Здесь, думаю, можно сослаться на Курячего, сказав о том, что макроподстановки вместо вызовов процедур удлиняют текст программы. Скорее всего здесь нужно использовать более наглядный пример взрыва, чем НОД, так как в этом примере в основном идет обертки над системными вызовами. Можно подсмотреть у Курячего.

Ну и философски оформить мысль о том, что часто макросами целесообразно обертывать вызовы подпрограмм для сокращения общего размера программы. Несмотря на их удобства. На практике обычно нужно искать баланс, аналогично использованию обычных и `inline` функций в C++.

7.5 Создание многомодульных программ

Многомодульные программы состоят из нескольких единиц компиляции, в каждой из которых сформирован некоторый код реализующий часть решаемой задачи. Для уже существующей программы, вычисляющий наибольший общий делитель, выделим в отдельные файлы подпрограмму нахождения НОД (файл euclid.s) и главную программу, осуществляющую ввод данных, вызов подпрограммы и вывод результата. Для того, чтобы подпрограммы могли быть видны из вне, необходимы их имена отметить директивой `.global` (или с использованием `.globl`, что является дубликатом предыдущей директивы).

Текст программы находится в каталоге `euclid/euclid05mod`.

Для компиляции и выполнения многомодульных программ необходимо в меню эмулятора выставить соответствующие опции (рисунок 7.1)

> **Примечание.**: Кстати, можно многофайловый проект ассемблировать и запустить, используя опции командной строки «`p sm`». Они являются аналогами опций, устанавливаемых через меню.

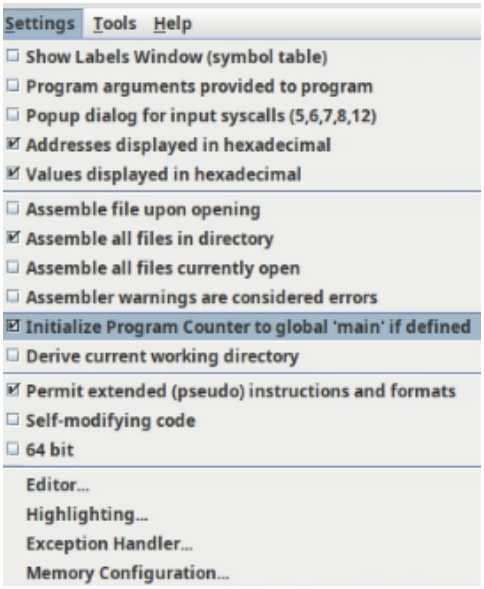


Рисунок 7.1 — Установка опций «Assemble all files in directory» и «Initialize Program Counter to global 'main' if defined» для компиляции и выполнения многофайловых программ

7.6 Домашнее задание

До 8 баллов

Написать программу, осуществляющую суммирование целочисленных элементов одномерного массива. Количество элементов в массиве может варьироваться от 1 до 10. Сами элементы вводятся с клавиатуры. Значение суммы также выводится в консоль эмулятора RARS. Необходимо контролировать, чтобы число вводимых элементов не превышало максимально допустимое. В случае, когда возникает переполнение, необходимо вывести последнее корректное значение суммы и число просуммированных при этом элементов. Суммирование осуществлять после размещения массива в памяти.

При выполнении задания для ввода и вывода массивов, вычисления суммы использовать подпрограммы, размещенные в отдельных файлах общего каталога. Для ввода и вывода отдельных элементов массива использовать макроопределения из библиотеки, рассмотренной на семинарах.

Опционально до +2 баллов

Обернуть подпрограммы ввода массива, вычисления суммы, вывода массива соответствующими макросами, используя эти макросы в основной программе вместо вызова подпрограмм. Эти макросы оформить в отдельном файле, подключаемом к основной программе вместо описания в ней глобальных точек. Подпрограммы должны также оставаться в отдельных файлах.

Срок сдачи задания: до начала восьмого семинарского занятия в каждой из групп.

8 Семинар 08. Математический сопроцессор. Арифметика с плавающей точкой

Целью семинара является изучение подходов к реализации арифметики с плавающей точкой за счет применения математического сопроцессора.

На занятии предполагается рассмотреть следующие темы:

1. Представление чисел с плавающей точкой.
2. Организация математического сопроцессора в архитектуре RISC-V.
3. Псевдоинструкции.
4. Управление вычислениями в арифметическом сопроцессоре.
5. Примеры вычислений с плавающей точкой.

8.1 Представление чисел с плавающей точкой

> **Примечание**: Следует отметить, что формат чисел с плавающей точкой уже был рассказан на лекции. Поэтому данный материал можно было бы вообще не упоминать. Поэтому я его как бы и не упоминаю. Но можно задать вопрос о том помнят ли студенты, как представляются числа с плавающей точкой в стандарте IEEE-754.Естественно, что больше половины ничего не знают, так как игнорируют лекции. В этой ситуации я планирую просто открыть лекцию и быстро пройтись по ее содержанию одновременно с демонстрацией различных вариантов чисел (включая и крайние) на эмуляторе.

То есть, как и Г. Курячий, обратиться к RARS и открыть окно эмулятора сопроцессора с плавающей точкой через меню (Tools->Floating Point Representaion). И в этом диалоговом окне имеется визуализация числа с плавающей точной. Можно рассмотреть, что из себя различные значения числа с плавающей точкой представляют. И как они отражаются на двоичное представление. Следует при этом учесть, что это диалоговое окно используется для отображение 32-х разрядных чисел с плавающей точкой. Поэтому, в соответствии с размерностью, ненормализованное представление начинается где-то после 10^{-38}.

Следует подчеркнуть, что манипулировать можно не только десятичным числом, но и двоичным, выставляя знак, мантиссу и порядок. Таким образом можно показать все варианты (они отображаются в десятичном окне), включая прямое задание ненормализованных чисел, неопределенные значения, плюс/минус бесконечность.

Помимо этого во время выполнения программы данный девайс можно связать с программой и в отладочном режиме просматривать содержимое регистров с плавающей точкой (32-х разрядных значений) в десятичной системе счисления (рисунок 8.1).

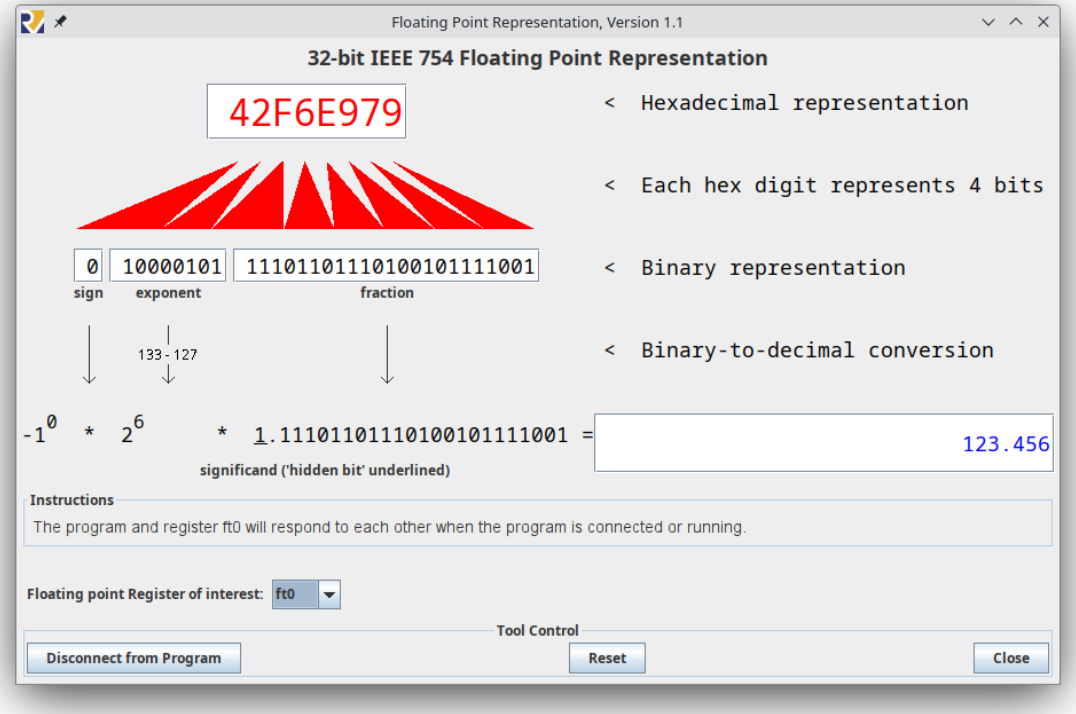


Рисунок 8.1 — Диалоговое окно для просмотра 32-х разрядных чисел с плавающей точкой

Примечание: Курячий показал только манипуляции в десятичном окне. У нас есть время показать больше. Помимо этого можно даже провести небольшое тестирование по этому поводу. Например, дать задание показать, каким образом в двоичном виде будут представлены Пи, Е, и другие числа. Включая не входящие в диапазон 32-х разрядного числа. Можно также поступить наоборот (из двоичной в десятичную), но не знаю, зачем это нужно. Хотя, я им рассказывал, что в LUA использовались ненормализованные числа для представления целочисленной арифметики (52 разряда мантиссы и знак 64-разрядного числа). Можно задать вопрос: какой диапазон целочисленной арифметики может быть представлен в 32-х разрядном числе с плавающей точкой за счет мантиссы и знака.

8.2 Организация математического сопроцессора в архитектуре RISC-V

Можно остановится на определении сопроцессора (как и у Курячего) как дополнительного устройства, предназначенного для реализации дополнительной функциональности. Сказать, что оно может быть реализовано для различных целей и быть выполнено как на том же кристалле, что и универсальный процессор, или разработано в виде отдельного кристалла (я рассказывал об отдельном Intel 8087), или же быть отдельным устройством (например, видеокарта).

> **Можно немного поговорить**: Возможный вопрос: почему не использовать программное решение для арифметики с плавающей точкой на основе целочисленных регистров и моделирования на них любых форматов? Ответ: слишком медленно, а стандартные форматы с плавающей точкой обеспечивают переносимость между различными архитектурами.

8.2.1 Варианты реализации математического сопроцессора

После этого можно перейти к особенностям математического сопроцессора в архитектуре RISC-V. Сказать, что существуют различные стандарты расширения:

- **F (float)** – поддерживает числа с плавающей точкой одинарной точности (32 разряда);
- **D (double)** – поддерживает числа с плавающей точкой двойной точности (64 разряда);
- **Q (quadruple)** – поддерживает числа с плавающей точкой четверной точности (128 разрядов);
- **ZfH (half)** – поддерживает числа с плавающей точкой половинной точности (16 разрядов).

Отметить, что RARS поддерживает стандарты **F** и **D**.

8.2.2 Регистры математического сопроцессора

Здесь содержание практически стандартно. Архитектура RISC-V предусматривает наличие тридцати двух регистров для чисел с плавающей точкой «f0–f31». В математическом сопроцессоре это свои регистры, которые не связаны с регистрами процессора. Разрядность этих регистров определяется поддерживаемым стандартом расширения. В RARS это 64 разряда (double). При использовании чисел меньшей разрядности (в RARS 32 разряда) старшие части таких регистров заполняются значением **NaN** старшего числа. Тогда при попытке прочесть 64-разрядное число прочтется **NaN**.

> **Примечание**: Имеет смысл специально проверить при выполнении примеров.

В соответствие с соглашением о вызове подпрограмм (`https://github.com/riscv-non-isa/riscv-elf-psabi-doc/blob/master/riscv-cc.adoc`), у этих регистров разное назначение (рисунок 8.2).

Общее название	Мнемоника ABI	Назначение	Сохранение при вызове подпрограммы
f0 - f7	ft0 - ft7	Временные	Нет
f8 - f9	fs0 - fs1	Сохраняемые при вызове	Да
f10 - f17	fa0 - fa7	Параметры и возвращаемые значения	Нет
f18 - f27	fs2 - fs11	Сохраняемые при вызове	Да
f28 - f31	ft8 - ft11	Временные	Нет

Рисунок 8.2 — Соглашения по использованию регистров с плавающей точкой

Можно отметить, что в целом за счет отсутствия специализации, количество используемых регистров больше. Но при этом также заметить, что это расширение процессора. То есть, дополнительное устройство внутри кристалла. Следовательно, оно создается в случае более мощных процессоров и не присутствует в микроконтроллерах, которым оно не нужно из-за специфики их применения.

8.2.3 Система команд математического сопроцессора

> **Примечание**: Я разбил команды по группам на будущее. Возможно, что не совсем корректно и без перевода. Но думаю, что при формировании более сложной методы это можно будет повторно использовать.

Команды обработки данных арифметического сопроцессора

Команды обработки данных арифметического сопроцессора представлены в таблице 8.1

Команды пересылки данных арифметического сопроцессора

Команды пересылки данных арифметического сопроцессора представлены в таблице 8.2

Команды конвертации данных арифметического сопроцессора

Команды конвертации данных арифметического сопроцессора представлены в таблице 8.3

Форматы команд математического сопроцессора совпадают с форматами **S** (запись в память, fs*), **I** (чтение из памяти, fl*) и **R** (вычисления).

Для хранения и передачи данных можно использовать слова, двойные слова, так как на уровне системы команд используется операционная однозначность (однозначность выполнения операций рассматривалась на лекциях). Поэтому мнемоника комнд пересылки использует обозначения «word» / «double word»: flw, fsw, fld, fsd,

Таблица 8.1 — Команды обработки данных арифметического сопроцессора

Команда	**Описание**
`fadd.d f1, f2, f3, dyn`	Floating ADD (64 bit): assigns f1 to f2 + f3
`fadd.s f1, f2, f3, dyn`	Floating ADD: assigns f1 to f2 + f3
`fdiv.d f1, f2, f3, dyn`	Floating DIVide (64 bit): assigns f1 to f2 / f3
`fdiv.s f1, f2, f3, dyn`	Floating DIVide: assigns f1 to f2 / f3
`fmul.d f1, f2, f3, dyn`	Floating MULtiply (64 bit): assigns f1 to f2 * f3
`fmul.s f1, f2, f3, dyn`	Floating MULtiply: assigns f1 to f2 * f3
`fsgnj.d f1, f2, f3`	Floating point sign injection (64 bit): replace the sign bit of f2 with the sign bit of f3 and assign it to f1
`fsgnj.s f1, f2, f3`	Floating point sign injection: replace the sign bit of f2 with the sign bit of f3 and assign it to f1
`fsgnjn.d f1, f2, f3`	Floating point sign injection (inverted 64 bit): replace the sign bit of f2 with the opposite of sign bit of f3 and assign it to f1
`fsgnjn.s f1, f2, f3`	Floating point sign injection (inverted): replace the sign bit of f2 with the opposite of sign bit of f3 and assign it to f1
`fsgnjx.d f1, f2, f3`	Floating point sign injection (xor 64 bit): xor the sign bit of f2 with the sign bit of f3 and assign it to f1
`fsgnjx.s f1, f2, f3`	Floating point sign injection (xor): xor the sign bit of f2 with the sign bit of f3 and assign it to f1
`fsub.d f1, f2, f3, dyn`	Floating SUBtract (64 bit): assigns f1 to f2 - f3
`fsub.s f1, f2, f3, dyn`	Floating SUBtract: assigns f1 to f2 - f3

(а также *q и *h, если они реализованы в соответствующих стандартах расширений архитектуры).

Точность арифметических команд сопроцессора указывается суффиксом инструкции (s, d, q, h) который в ассемблере добавляется к коду операций, отделяясь от него точкой: `fCMD.P`, где `CMD` — мнемоника инструкции, `P` — точность.

Примеры команд `CMD: add, sub, mul, div, sqrt, min, max`

Вычисление полусуммы двух чисел

Пример от Г.Курячего, демонстрирующий основные команды, пояснения есть в видео.

```
    .data
a:      .float  123.456
b:      .float  654.321
_2:     .float  2
    .text
    flw     ft0 a t0
```

Таблица 8.2 — Команды пересылки данных арифметического сопроцессора

Команда	**Описание**
`fld f1, -100(t1)`	Load a double from memory
`flw f1, -100(t1)`	Load a float from memory
`fmv.s.x f1, t1`	Move float: move bits representing a float from an integer register
`fmv.x.s t1, f1`	Move float: move bits representing a float to an integer register
`fsd f1, -100(t1)`	Store a double to memory
`fsw f1, -100(t1)`	Store a float to memory

Таблица 8.3 — Команды конвертации данных арифметического сопроцессора

Команда	**Описание**
`fcvt.d.s f1, f2, dyn`	Convert a float to a double: Assigned the value of f2 to f1
`fcvt.d.w f1, t1, dyn`	Convert double from integer: Assigns the value of t1 to f1
`fcvt.d.wu f1, t1, dyn`	Convert double from unsigned integer: Assigns the value of t1 to f1
`fcvt.s.d f1, f2, dyn`	Convert a double to a float: Assigned the value of f2 to f1
`fcvt.s.w f1, t1, dyn`	Convert float from integer: Assigns the value of t1 to f1
`fcvt.s.wu f1, t1, dyn`	Convert float from unsigned integer: Assigns the value of t1 to f1
`fcvt.w.d t1, f1, dyn`	Convert integer from double: Assigns the value of f1 (rounded) to t1
`fcvt.w.s t1, f1, dyn`	Convert integer from float: Assigns the value of f1 (rounded) to t1
`fcvt.wu.d t1, f1, dyn`	Convert unsinged integer from double: Assigns the value of f1 (rounded) to t1
`fcvt.wu.s t1, f1, dyn`	Convert unsinged integer from float: Assigns the value of f1 (rounded) to t1

```
flw     ft1 b t0
flw     ft2 _2 t0
fadd.s  ft3 ft2 ft1
fdiv.s  fa0 ft3 ft2
li      a7 2
ecall
```

При этом какие регистры и как используются определяется кодом операции. Информация об этих командах также имеется в системе помощи RARS. Переключение на сопроцессор осуществляет устройство управления по коду операции команды. Оно также определяе использование регистров с плавающей точкой совместно с целочисленными регистрами.

Пример раскрытия команды `flw` (рисунок 8.3). Представлен также в видео.

Code	Basic			
0x0fc10297	auipc x5,0x0000fc10	6:	flw	ft0 a t0
0x0002a007	flw f0,0(x5)			
0x0fc10297	auipc x5,0x0000fc10	7:	flw	ft1 b t0
0xffc2a087	flw f1,0xfffffffc(x5)			
0x0fc10297	auipc x5,0x0000fc10	8:	flw	ft2 _2 t0
0xff82a107	flw f2,0xfffffff8(x5)			

Рисунок 8.3 — Раскрытие команды flw при ассемблировании

Также с данной программой можно пройтись пошагово. Тогда можно увидеть как меняются различные регистры. Обратить внимание на t0, используемый в данном случае для формирования базы адреса в памяти, чтобы обеспечить доступ для загрузки числа. А также отметить использование уже рассмотренной команды `auipc` для формирования позиционно независимого кода. При этом можно обратить внимание, что 32-разрядные слова фиксируются в регистрах с плавающей точкой без указания старшей части, которая должна быть **NaN**.

Команды арифметического сопроцессора, осуществляющие обработку сложных выражений

Наряду с традиционными форматами команд в арифметический процессор добавлен новый тип команд **R4** (рисунок 8.4), поддерживающих четыре операнда.

Общее название	Мнемоника ABI	Назначение	Сохранение при вызове подпрограммы
f0 - f7	ft0 - ft7	Временные	Нет
f8 - f9	fs0 - fs1	Сохраняемые при вызове	Да
f10 - f17	fa0 - fa7	Параметры и возвращаемые значения	Нет
f18 - f27	fs2 - fs11	Сохраняемые при вызове	Да
f28 - f31	ft8 - ft11	Временные	Нет

Рисунок 8.4 — Формат **R4**, используемый для поддержки составных операций

Данные команды выполняют не элементарные действия а вычисления по более сложным выражениям с чередованием операций умножения, сложения, вычитания, смены знака. Эти команды представлены в таблице 8.4

8.2.4 Обмен между регистрами с плавающей точкой и целочисленными регистрами

Можно обмениваться не с памятью, а с регистрами общего назначения. В мнемонике используются два суффикса:

Перемещать машинное слово из одного регистра в другой умеет центральный процессор: `fmv.s.x` и `fmv.x.s` (для двойной точности такой инструкции нет, так

Таблица 8.4 — Команды арифметического сопроцессора, осуществляющие обработку сложных выражений

Команда	**Описание**
`fmadd.d f1, f2, f3, f4, dyn`	Fused Multiply Add (64 bit): Assigns f2*f3+f4 to f1
`fmadd.s f1, f2, f3, f4, dyn`	Fused Multiply Add: Assigns f2*f3+f4 to f1
`fmsub.d f1, f2, f3, f4, dyn`	Fused Multiply Subatract: Assigns f2*f3-f4 to f1
`fmsub.s f1, f2, f3, f4, dyn`	Fused Multiply Subatract: Assigns f2*f3-f4 to f1
`fnmadd.d f1, f2, f3, f4, dyn`	Fused Negate Multiply Add (64 bit): Assigns -(f2*f3+f4) to f1
`fnmadd.s f1, f2, f3, f4, dyn`	Fused Negate Multiply Add: Assigns -(f2*f3+f4) to f1
`fnmsub.d f1, f2, f3, f4, dyn`	Fused Negated Multiply Subatract: Assigns -(f2*f3-f4) to f1
`fnmsub.s f1, f2, f3, f4, dyn`	Fused Negated Multiply Subatract: Assigns -(f2*f3-f4) to f1
`fmax.d f1, f2, f3`	Floating MAXimum (64 bit): assigns f1 to the larger of f1 and f3
`fmax.s f1, f2, f3`	Floating MAXimum: assigns f1 to the larger of f1 and f3
`fmin.d f1, f2, f3`	Floating MINimum (64 bit): assigns f1 to the smaller of f1 and f3
`fmin.s f1, f2, f3`	Floating MINimum: assigns f1 to the smaller of f1 and f3
`fsqrt.d f1, f2, dyn`	Floating SQuare RooT (64 bit): Assigns f1 to the square root of f2
`fsqrt.s f1, f2, dyn`	Floating SQuare RooT: Assigns f1 to the square root of f2

как процессор манипулирует только 32-х разрядными словами). Целое число при этом не преобразуется.

Но преобразовывать из вещественного формата в целый и обратно (а также из двойного в одинарный и обратно) может только FPU: `fcvt.d.s` `fcvt.s.d`, `fcvt.P.w[u]` и `fcvt.w[u].P`.

Пример вычисления функции $(x-1)^2$. Расчет с использованием приведения формулы к виду: $x^2 - 2x + 1$.

```
    .data
x:      .double 12.34
    .text
li              t2 2
fcvt.d.w        ft2 t2                  # ft2 = 2.0
li              t1 1
```

```
fcvt.d.w        ft1 t1                  # ft1 = 1.0
fld             ft0 x t0                # ft0 = x
fnmsub.d        ft3 ft2 ft0 ft1         # ft3 = -(2 * x) + 1
fmadd.d         fa0 ft0 ft0 ft3         # fa0 = x * x + ft3
li              a7 3                    # Вывод числа двойной точности
ecall
```

Перемещение между f-регистрами: `fmv f1 f2` — в действительности это псевдоинструкция на базе инструкции расширения знака `fsgnj.P`.

Во многих архитектурах (в частности, MIPS) недооценили важность и частоту операции «смены знака по образцу», а вычислительно эта операция непростая, особенно для вещественных чисел. Поэтому в RISC-V есть специальная операция: взять знак из `fA`, а мантиссу и порядок из `fB`, и всё это положить в `fC`.

Тогда fmv расгрывается так:

```
0x00400028  0x222100d3  fsgnj.d f1,f2,f2     13     fmv.d   ft1 ft2
```

8.2.5 Команды сравнения

feq|lt|le/P x1 f1 f2 — записывает 0 или 1 в x1

Остальные сравнения — псевдоинструкции

> **TODO**: При дальнейшем оформлении сюда стоит перетащить команды из таблицы.

8.3 Псевдоинструкции

> **Примечание**: Вопрос: где о них говорить? В ходе пояснения основных групп команд или отдельно? На выбор, наверное.

Как и в случае с командами основного процессора, при программировании арифметического процессора также используются псевдоинструкции (псевдокоманды), повышающие гибкость в разработке программ. Они затрагивают практически весь спектр команд с плавающей точкой и ниже представлены разнесенными в различные подгруппы.

8.3.1 Псевдоинструкции для обработки данных

Как и соответствующие команды, команды обработки данных осуществляют различные функциональные преобразования. Они представлены в таблице 8.5.

8.3.2 Псевдоинструкции пересылки, загрузки и сохранения данных

Псевдокоманды, обеспечивающие пересылку данных между различными регистрами и областями памяти представлены в таблице 8.6. Они расширяют возмож-

ности основных команд пересылки данных, обеспечивая программиста более компактными псевдонимами.

8.3.3 Псевдоинструкции сравнения

Псевдокоманды сравнения представлены в таблице 8.7. Через уже существующие команды реализованы псевдокоманды, задающие отношения больше, а также больше или равно.

8.3.4 Псевдоинструкции конвертации данных

Данный вид псевдоинструкций осуществляет трансформацию между различными форматами данных. Соответствующие псевдоинструкции представлены в таблице 8.8.

Таблица 8.5 — Псевдокоманды обработки данных

Команда	**Описание**
`fabs.d f1, f2`	Set f1 to the absolute value of f2 (64 bit)
`fabs.s f1, f2`	Set f1 to the absolute value of f2
`fadd.d f1, f2, f3`	Floating ADD (64 bit): assigns f1 to f2 + f3
`fadd.s f1, f2, f3`	Floating ADD: assigns f1 to f2 + f3
`fdiv.d f1, f2, f3`	Floating DIVide (64 bit): assigns f1 to f2 / f3
`fdiv.s f1, f2, f3`	Floating DIVide: assigns f1 to f2 / f3
`fmadd.d f1, f2, f3, f4`	Fused Multiply Add (64 bit): Assigns f2*f3+f4 to f1
`fmadd.s f1, f2, f3, f4`	Fused Multiply Add: Assigns f2*f3+f4 to f1
`fmsub.d f1, f2, f3, f4`	Fused Multiply Subatract (64 bit): Assigns f2*f3-f4 to f1
`fmsub.s f1, f2, f3, f4`	Fused Multiply Subatract: Assigns f2*f3-f4 to f1
`fmul.d f1, f2, f3`	Floating MULtiply (64 bit): assigns f1 to f2 * f3
`fmul.s f1, f2, f3`	Floating MULtiply: assigns f1 to f2 * f3
`fneg.d f1, f2`	Set f1 to the negation of f2 (64 bit)
`fneg.s f1, f2`	Set f1 to the negation of f2
`fnmadd.d f1, f2, f3, f4`	Fused Negate Multiply Add (64 bit): Assigns -(f2*f3+f4) to f1
`fnmadd.s f1, f2, f3, f4`	Fused Negate Multiply Add: Assigns -(f2*f3+f4) to f1
`fnmsub.d f1, f2, f3, f4`	
	Fused Negated Multiply Subatract (64 bit): Assigns -(f2*f3-f4) to f1
`fnmsub.s f1, f2, f3, f4`	Fused Negated Multiply Subatract: Assigns -(f2*f3-f4) to f1
`fsqrt.d f1, f2`	Floating SQuare RooT (64 bit): Assigns f1 to the square root of f2
`fsqrt.s f1, f2`	Floating SQuare RooT: Assigns f1 to the square root of f2
`fsub.d f1, f2, f3`	Floating SUBtract (64 bit): assigns f1 to f2 - f3
`fsub.s f1, f2, f3`	Floating SUBtract: assigns f1 to f2 - f3

Таблица 8.6 — Псевдокоманды пересылки, загрузки, сохранения

Команда	**Описание**
`fld f1,%lo(label)(t2)`	Load from Address
`fld f1,(t2)`	Load Word: Set f1 to 64-bit value from effective memory word address
`fld f1,-100`	Load Word: Set f1 to 64-bit value from effective memory word address
`fld f1,10000000,t3`	Load Word: Set f1 to 64-bit value from effective memory word address using t3 as a temporary
`fld f1,label, t3`	Load Word: Set f1 to 64-bit value from effective memory word address using t3 as a temporary
`flw f1,%lo(label)(t2)`	Load from Address
`flw f1,(t2)`	Load Word Coprocessor 1 : Set f1 to 32-bit value from effective memory word address
`flw f1,-100`	Load Word Coprocessor 1 : Set f1 to 32-bit value from effective memory word address
`flw f1,10000000,t3`	Load Word Coprocessor 1 : Set f1 to 32-bit value from effective memory word address using t3 as a temporary
`flw f1,label, t3`	Load Word Coprocessor 1 : Set f1 to 32-bit value from effective memory word address using t3 as a temporary
`fmv.d f1, f2`	Move the value of f2 to f1 (64 bit)
`fmv.s f1, f2`	Move the value of f2 to f1
`fmv.w.x f1, t1`	Move float (New mnemonic): move bits representing a float from an integer register
`fmv.x.w t1, f1`	Move float (New mnemonic): move bits representing a float to an integer register
`fsd f1,(t2)`	Store Word: Store 64-bit value from f1 to effective memory word address
`fsd f1,-100`	Store Word: Store 64-bit value from f1 to effective memory word address
`fsd f1,10000000,t3`	Store Word: Store 64-bit value from f1 to effective memory word address using t3 as a temporary
`fsd f1,label, t3`	
	Store Word: Store 64-bit value from f1 to effective memory word address using t3 as a temporary
`fsw f1,(t2)`	Store Word Coprocessor 1 : Store 32-bit value from f1 to effective memory word address
`fsw f1,-100`	Store Word Coprocessor 1 : Store 32-bit value from f1 to effective memory word address
`fsw f1,10000000,t3`	Store Word Coprocessor 1 : Store 32-bit value from f1 to effective memory word address using t3 as a temporary
`fsw f1,label, t3`	Store Word Coprocessor 1 : Store 32-bit value from f1 to effective memory word address using t3 as a temporary

Таблица 8.7 — Псевдокоманды сравнения

Команда	**Описание**
`fge.d t1, f2, f3`	Floating Greater Than or Equal (64 bit): if f2 >= f3, set t1 to 1, else set t1 to 0
`fge.s t1, f2, f3`	Floating Greater Than or Equal: if f2 >= f3, set t1 to 1, else set t1 to 0
`fgt.d t1, f2, f3`	Floating Greater Than (64 bit): if f2 > f3, set t1 to 1, else set t1 to 0
`fgt.s t1, f2, f3`	Floating Greater Than: if f2 > f3, set t1 to 1, else set t1 to 0

Таблица 8.8 — Псевдокоманды преобразования данных

Команда	**Описание**
`fcvt.d.s f1, f2`	Convert float to double: Assigned the value of f2 to f1
`fcvt.d.w f1, t1`	Convert double from signed integer: Assigns the value of t1 to f1
`fcvt.d.wu f1, t1`	Convert double from unsigned integer: Assigns the value of t1 to f1
`fcvt.s.d f1, f2`	Convert double to float: Assigned the value of f2 to f1
`fcvt.s.w f1, t1`	Convert float from signed integer: Assigns the value of t1 to f1
`fcvt.s.wu f1, t1`	Convert float from unsigned integer: Assigns the value of t1 to f1
`fcvt.w.d t1, f1`	Convert signed integer from double: Assigns the value of f1 (rounded) to t1
`fcvt.w.s t1, f1`	Convert signed integer from float: Assigns the value of f1 (rounded) to t1
`fcvt.wu.d t1, f1`	Convert unsigned integer from double: Assigns the value of f1 (rounded) to t1
`fcvt.wu.s t1, f1`	Convert unsigned integer from float: Assigns the value of f1 (rounded) to t1

9 Семинар 09. Математический сопроцессор. Арифметика с плавающей точкой. Продолжение

Продолжение темы по арифметическому сопроцессору. В данном случае искусственное разделение темы по занятиям. Часть вопросов рассмотрено в продолжении рассматрвается:

На занятии предполагается рассмотреть следующие темы, связанные с управлением вычислениями в арифметическом сопроцессоре:

1. Сравнение данных с плавающей точкой.
2. Команды классификации
3. Блок управляющих регистров
4. Условные операторы и fcsr
5. Проверка на потерю точности.

9.1 Сравнение данных с плавающей точкой

Команды сравнения и анализа

Команды сравнения и анализа представлены в таблице 9.1

9.2 Команды классификации

Инструкция `fclass.(s/d) t1, f1` проверяет значение в регистре чисел с плавающей точкой `f1` и записывает в целочисленный регистр `t1` 10-битную маску, указывающую класс числа с плавающей запятой. Формат маски описан в таблице 9.2. Соответствующий бит в t1 будет установлен, если свойство истинно, и очищен в противном случае. Все остальные биты в `t1` очищаются. При этом в `t1` **будет установлен только один бит**. `fclass` не устанавливает флаги исключений с плавающей запятой.

Таблица 9.1 — Команды сравнения и анализа

Команда	**Описание**
`fclass.d t1, f1`	Classify a floating point number (64 bit)
`fclass.s t1, f1`	Classify a floating point number
`feq.d t1, f1, f2`	Floating EQuals (64 bit): if f1 = f2, set t1 to 1, else set t1 to 0
`feq.s t1, f1, f2`	Floating EQuals: if f1 = f2, set t1 to 1, else set t1 to 0
`fle.d t1, f1, f2`	Floating Less than or Equals (64 bit): if f1 <= f2, set t1 to 1, else set t1 to 0
`fle.s t1, f1, f2`	Floating Less than or Equals: if f1 <= f2, set t1 to 1, else set t1 to 0
`flt.d t1, f1, f2`	Floating Less Than (64 bit): if f1 < f2, set t1 to 1, else set t1 to 0
`flt.s t1, f1, f2`	Floating Less Than: if f1 < f2, set t1 to 1, else set t1 to 0

Таблица 9.2 — Биты, устанавливаемые командами классификации

Установленный бит	**Свойства числа с плавающей точкой**
`0 (0x001)`	f1 — это -∞
`1 (0x002)`	f1 — это отрицительное нормализованное число
`2 (0x004)`	f1 — это отрицительное денормализованное число
`3 (0x008)`	f1 — это -0
`4 (0x010)`	f1 — это +0
`5 (0x020)`	f1 — это положительное денормализованное число
`6 (0x040)`	f1 — это положительное нормализованное число
`7 (0x080)`	f1 — это +∞
`8 (0x100)`	f1 — это сигнализирующий NaN
`9 (0x200)`	f1 — это тихий NaN

Можно отметить, что в отличие от целочисленной арифметики проверка результатов вычислений с плавающей точкой может играть важное значение для получения достоверных решений. Например, возможно перемножение чисел, сильно отличающихся друг от друга, что ведет к накоплению ошибки вплоть до переполнения или исчезновения порядка. Поэтому ввод в спецпроцессор дополнительных команд контроля и классификации, обеспечивающих подобные проверки для принятия решений и управления, оправдан.

Для демонстрации я создал макрос `macro-is-class.mac`, который выводит информацию об этих проверках (лежит в каталоге `is-class` вместе с тестовой программой). Тестовый код можно расширить, добавив туда различные варианты чисел с плавающей точкой. Для получения крайних значений можно использовать либо пересылки данных посредством их подготовки в целочисленных регистрах, либо готовить их с использованием диалогового окна (для 32-х разрядных чисел).

9.3 Блок управляющих регистров (общее представление)

Блок управляющих регистров (БУР) — Control and Status Register (CSR). Более подробно будет рассматриваться при изучении новых свойств. Пока кратко применительно к арифметическому сопроцессору.

RISC-V определяет отдельное адресное пространство из 4096 регистров управления и состояния, связанных с каждым портом. Регистры образуют 4K (12 битов) пространство управляющих счётчиков, флагов, масок и прочего. Имеется набор инструкций CSR, которые работают с блоком. Все инструкции CSR атомарно читают-изменяют-записывают один регистр блока, спецификатор регистра которого закодирован в 12-битном поле csr инструкции, хранящемся в битах 31–20. Это атомарные R/W инструкции типа I.

В RARS часть используемых регистров отображается во вкладке «**Control and Status**».

Команды для работы с блоком управляющих регистров представлены в тиблице 9.3. Все инструкции CSR атомарно читают–изменяют–записывают один CSR, спецификатор CSR которого закодирован в 12-битном поле csr инструкции, хранящемся в битах 31–20. Инструкции с непосредственным операндом используют 5-битную расширенную нулем непосредственную информацию, закодированную в поле rs1.

Таблица 9.3 — Команды для работы с регистром блока управляющих регистров

Команда	**Описание**
`csrrc t0, fcsr, t1`	Atomic Read/Clear CSR: read from the CSR into t0 and clear bits of the CSR according to t1
`csrrci t0, fcsr, 10`	Atomic Read/Clear CSR Immediate: read from the CSR into t0 and clear bits of the CSR according to a constant
`csrrs t0, fcsr, t1`	Atomic Read/Set CSR: read from the CSR into t0 and logical or t1 into the CSR
`csrrsi t0, fcsr, 10`	Atomic Read/Set CSR Immediate: read from the CSR into t0 and logical or a constant into the CSR
`csrrw t0, fcsr, t1`	Atomic Read/Write CSR: read from the CSR into t0 and write t1 into the CSR
`csrrwi t0, fcsr, 10`	Atomic Read/Write CSR Immediate: read from the CSR into t0 and write a constant into the CSR

То есть, основная идея практически всех команд заключается в сохранении читаемого состояния и в различных вариантов установки нового состояния одного из регистров.

9.3.1 Регистр fcsr (0x003)

Регистр `fcsr` имеет следующие поля:

- 5 бит — флаги
  - NX потеря точности
  - UF сверхмалое число
  - OF переполнение
  - DZ деление на 0
  - NV недопустимая операция
- 3 бита — тип округления
  - RNE ближайшее, лучше чётное
  - RTZ ближайшее к нулю
  - RDN ближайшее к -∞
  - RUP ближайшее к +∞
  - RMM ближайшее, лучше с большим модулем
  - ...
  - ...
  - DYN не менять установленное по умолчанию (используется в инструкциях явного округления)

Для работы с регистром `fcsr` используются псевдоинструкции типа `f[sr]csr`. В RARS, видимо для удобства, в отдельные регистры выделены регистры `frm` и `fflags`, хотя это на самом деле поля `fcsr`. С `fflags` используется отдельная псевдоинструкция `frflags`, а `frm` используется инструкция `fsrm`. Соответствующие псевдоинструкции представлены в таблице 9.4.

В примере `round-up` приведено использование округления различными способами с использованием псевдокоманды `fsrm`:

- по умолчанию;
- в сторону 0 (явно);
- в сторону минус бесконечности.

```
.data
numb:   .float  7.5

.text
flw     ft0 numb t0
fcvt.w.s a0 ft0         # По умолчанию RNE (ближайшее, чётное) = 8
```

Таблица 9.4 — Псевдокоманды для работы с регистром `fcsr`

Команда	**Описание**
`frcsr t1`	Read FP control/status register
`frflags t1`	Read FP exception flags
`frrm t1`	Read FP rounding mode
`frsr t1`	Alias for frcsr t1
`fscsr t1, t2`	Swap FP control/status register
`fscsr t1`	Write FP control/status register
`fsflags t1`	Write FP exception flags
`fsflags t1, t2`	Swap FP exception flags
`fsrm t1`	Write FP rounding mode
`fsrm t1, t2`	Swap FP rounding mode
`fssr t1`	Alias for fscsr t1
`fssr t1, t2`	Alias for fscsr t1, t2

```
jal     outn
fcvt.w.s a0 ft0 rtz     # Ближайшее к 0 = 7
jal     outn
li      t0 2            # RDN - ближайшее к -бесконечности
fsrm    t0
fcvt.w.s a0 ft0         # теперь по умолчанию RDN = 7
jal     outn
fcvt.w.s a0 ft0 dyn     # Не менять установленное по умолчанию = 7
jal     outn
fcvt.w.s a0 ft0 rne     # RNE (ближайшее, чётное) = 8
jal     outn

li      a7 10
ecall

outn:   li      a7 1
ecall
li      a0 '\n'
li      a7 11
ecall
ret
```

Примечание: Я расширил пример. Здесь можно выдать задание потренироваться на подстановку различных чисел: отрицательных с дробной частью больше и меньше 0.5 и т.д.

9.4 Условные операторы и fcsr

Наряду с осуществление непосредственных сравнений, арифметический сопроцессор отображает своё состояние в управляющем регистре `fcsr`, который входит в блок управляющих регистров **CSR (Control and Status Register)**. Использование дополнительных регистров флагов и состояний в спецпроцессорах оправдано, так как существует необходимость в выполнении быстрых проверок сразу же после выполнения их инструкций.

Варианты организации условных переходов:

- использование инструкций `fCMP.P / fclas.P` с последующим переходом по содержимому целочисленного (X) регистра (0 / не 0);
- использование инструкций `fcsr*/frflags` с чтением всех флагов в целочисленный регистр, выделением конкретного флага инструкцией, например, `andi` и условный переход по содержимому целочисленного (X) регистра (0 / не 0).

В любом случае условный переход неатомарен.

Использование условных переходов на примере определения того, являются ли три отрезка сторонами треугольника (`is-float-triangle.s`). Используются команды непосредственного сравнения на неравенство с последующим объединением результатов сравнения по **И** (все неравенства должны быть истинными).

```
.data
yes:    .asciz  "It is a triangle\n"
no:     .asciz  "It is not a triangle\n"
.text
jal     input
fmv.d   fs2 fa0
jal     input
fmv.d   fs1 fa0
jal     input
fmv.d   fa1 fs1
fmv.d   fa2 fs2
jal     check
bnez    a0 true
la      a0 no
b       output
true:
la      a0 yes
output: li      a7 4
ecall
li      a7 10
ecall
```

```
.data
prompt: .ascii  "Enter triangle side: "
.text
input:
la      a0 prompt
li      a7 4
ecall
li      a7 7
ecall
ret

check:  fadd.d  ft0 fa1 fa2
flt.d   t0 fa0 ft0
fadd.d  ft1 fa2 fa0
flt.d   t1 fa1 ft1
fadd.d  ft2 fa1 fa0
flt.d   t2 fa2 ft2
and     a0 t0 t1
and     a0 a0 t2
ret
```

Следующий пример демонстрирует анализ на потерю точности, возможную в вычислительных задачах (`precision.s`). Вычисляется выражение `(a + b) / c`.

```
.include "macro-common.mac"
.data
Exact:  .asciz  " - это точное решение!\n"
Inex:   .asciz  " - результат неточный (с округлением).\n"

.text
print_str ("Считаем по формуле: (a + b) / c\n")
print_str ("a = ")
li      a7 6
ecall
fmv.s   fs0 fa0         # A
print_str ("b = ")
li      a7 6
ecall
fmv.s   fs1 fa0         # B
print_str ("c = ")
li      a7 6
ecall
fmv.s   fs2 fa0         # C
```

```
fadd.s  ft0 fs0 fs1
frflags t0              # Флаги FPU
andi    t0 t0 1         # Потеря точности?
fdiv.s  fa0 ft0 fs2
frflags t1              # Флаги FPU
andi    t1 t1 1         # Потеря точности?
or      s1 t0 t1        # ...хотя бы раз
print_str ("Result = ")
li      a7 2
ecall
la      a0 Inex
bnez    s1 out
la      a0 Exact

out:    li      a7 4
ecall
li      a7 10
ecall
```

Примечание: Для большей наглядности я подключил макробиблиотеку для вывода вспомогательных сообщений.
Примечание: Думаю, что по объему где-то примерно здесь (может чуть раньше или позже) будет конец второго семинара по теме. Но это можно сдвигать и обсуждать полученные реальные ситуации.

10 Семинар 10. Математический сопроцессор. Арифметика с плавающей точкой. Окончание

Завершение темы сопровождается рядом примеров. Возможно их не так много. Но пока в голову больше не лезет. В качестве дополнения можно рассмотреть дополнительное оформление в виде подпрограмм и макросов, обеспечивающих повышение юзабилити.

На занятии предполагается рассмотреть следующие примеры:

1. Предварительная проверка числа перед выполнением вычислений на примере вычисления квадратного корня.
2. Вычисление площади треугольника по формуле Герона.
3. Вычисление числа e с заданной точностью.

10.1 Примеры вычислений с плавающей точкой

Специфической особенностью арифметики с плавающей точкой является отсутствие точных проверок чисел на равенство. Помимо этого часто различные вычисления осуществляются приближенно с заданной точностью.

10.1.1 Предварительная проверка числа перед выполнением вычислений на примере вычисления квадратного корня

Ниже представлено (из Г. Курячего) вычисление квадратного корня из целого числа, с предварительной проверкой его на положительное значение уже в регистре арифметического сопроцессора:

```
li      a7 5
ecall                   # Ввод целого
fcvt.s.w ft1 a0         # Преобразование в вещественное
fmv.s.x ft0 zero        # 0 не надо преобразовывать!
flt.s   t0 ft1 ft0      # t0 = ft1 < 0 ?
fmv.s   fa0 ft0         # результат пока 0...
```

```
bnez    t0 negat        # ... и останется 0, если число отрицательное
fsqrt.s fa0 ft1         # вычислим корень
negat:  li      a7 2            # выведем результат
ecall
li      a7 10
ecall
```

Примечание: Код простой. Поэтому со вспомогательными сообщениями решил не мудрить.

10.1.2 Вычисление площади треугольника по формуле Герона

В примере рассматривается простое решение, связанное с вычислением площади треугольника по формуле: $S = \sqrt{p * (p-a) * (p-b) * (p-c)}$ где a, b, c - стороны треугольника, $p = (a+b+c)/2$.

Простая программа (triangle-area) выглядит следующим образом:

```
.data
yes:    .asciz  "It is a triangle\n"
no:     .asciz  "It is not a triangle\n"
area:   .asciz  "Area = \n"
.text
# Последовательный ввод трех сторон
jal     input
fmv.s   fs2 fa0
jal     input
fmv.s   fs1 fa0
jal     input
fmv.s   fa1 fs1
fmv.s   fa2 fs2

jal     check      # Вызов проверки на треугольник

# Это не треугольник
bnez    a0 true
la      a0 no
li      a7 4
ecall
b       exit

# Это треугольник.
true:   la      a0 yes
li      a7 4
ecall
```

```
jal     geron   # Вычисление площади
li a7 2 # Вывод площади
ecall
exit:
li      a7 10
ecall

# Ввод стороны треугольника
.data
prompt: .ascii  "Enter triangle side: "
.text
input:  la      a0 prompt
li      a7 4
ecall
li      a7 6
ecall
ret

# Проверка, что это стороны треугольника
check:  fadd.s  ft0 fa1 fa2
flt.s   t0 fa0 ft0
fadd.s  ft1 fa2 fa0
flt.s   t1 fa1 ft1
fadd.s  ft2 fa1 fa0
flt.s   t2 fa2 ft2
and     a0 t0 t1
and     a0 a0 t2
ret

# Вычисление площади по формуле Герона
# Стороны остались в регистрах - параметрах
.data
double_two: .float 2.0
.text
geron:
# Вычисления полупериметра
fadd.s  ft0 fa0 fa1
fadd.s  ft0 ft0 fa2
flw ft2 double_two t0
fdiv.s  ft0 ft0 ft2
# Вычисление разностей
fsub.s ft3 ft0 fa0
fsub.s ft4 ft0 fa1
fsub.s ft5 ft0 fa2
```

```
# Их перемножение
fmul.s ft1 ft3 ft4
fmul.s ft1 ft1 ft5
fmul.s ft1 ft1 ft0
# Получение площади
fsqrt.s fa0 ft1
ret
```

10.1.3 Вычисление числа e с заданной точностью

Пример (`exp.s`) такой же как и у Курячего. Только взят из другого источника. Поэтому комментарии на английском. Вычисляет число e по формуле:

$\exp = 1/n! + 1/(n-1)! + ...$

Точность задается формулой:

$\epsilon = 1/(10 * *k))$

Добавил подсказку для ввода данных с использованием библиотеки макроопределений.

```
#
# Example: calculates e as an infinite sum: 1/n! + 1/(n-1)! + ...
#          with the specified precision epsilon = 1/(10 ** k))
#
.include "common.mac"
.data
one:
.double 1
ten:
.double 10

.text
main:
fld     f2, one, t0   # 1
fsub.d  f4, f4, f4    # n = 0
fmv.d   f6, f2        # n!
fmv.d   f8, f2        # here will be e
fld     f10, ten, t0  # here will be epsilon
fmv.d   f0, f2        # decimal length k

print_str("Введите число точных разрядов: ")
read_int_a0
enext:
blez    a0, edone     # 10 ** (k+1)
fmul.d  f0, f0, f10
addi    a0, a0, -1
```

```
j       enext
edone:
fdiv.d  f10, f2, f0   # epsilon

loop:
fadd.d  f4, f4, f2    # n = n+1
fmul.d  f6, f6, f4    # n! = (n-1)! * n
fdiv.d  f0, f2, f6    # next summand
fadd.d  f8, f8, f0
flt.d   t0, f0, f10   # next summand < epsilon
beqz    t0, loop

li      a7, 3         # output a double
fmv.d   fa0, f8
ecall
```

10.2 Домашнее задание

> **Примечание**: *Домашнее задание на арифметику с плавающей точкой не выдаем. Текст оставляю на будущее (на всякий случай, чтобы не вспоминать). Компенсируется индивидуальным заданием по этой теме.*

До 8 баллов

Написать программу, осуществляющую вычисление корня квадратного из положительного числа с заданной точностью, используя для этого соответствующую итерационную формулу. В качестве исходных данных вводятся исходное действительное число и точность вычислений, задаваемая также действительным числом. Вычисление корня оформить как отдельную подпрограмму, принимающую значение аргумента и точность, которая возвращает в качестве результата полученную величину квадратного корня. Основная программа осуществляет ввод исходного числа и точности с консоли с проверкой на корректные значения, а также вывод результата.

Опционально до +2 баллов

Обернуть ввод исходных данных и их проверку, вывод результата вычислений, вызов подпрограммы вычисления квадратного корня в отдельные макроопределения. Эти макроопределения оформить в отдельном файле, подключаемом к основной программе вместо описания в ней глобальных точек. Подпрограмма вычисления квадратного корня также должна находиться в отдельном файле.

При выполнении задания разработать только одну программу с учетом целевой оценки.

> **TODO**: Я сформировал весь материал. Вопрос в его общем представлении. В принципе на псевдоинструкциях можно особо не заморачиваться. Возможно на три занятия может и не хватить. Думаю, что стоит подумать в этом случае, что добавить.

11 Семинар 11. Обработка строк символов. Тексты. Файлы

Цель семинара — изучение команд и методов, обеспечивающих обработку строк символов. В данном случае акцент предлагается сделать на использовании кодировки ASCII, так как все кодировки, превышающие крайнее значение 127 либо различны в разных ОС, либо, как Юникод, требуют дополнительных алгоритмических решений, что не является задачей данной дисциплины. То есть, речь идет о байтах, их обработке и передаче в обе стороны между регистрами и памятью.

Помимо этого имеет смысл затронуть вопросы, связанные с использованием файлов в эмуляторе RARS, так как работа с текстами — это работа не с парой символов. Также работа с файлами будет в третьем задании на обработку строк символов.

На занятии предполагается рассмотреть следующие вопросы, связанные с обработкой строк символов:

1. Особенности обработки символов в процессоре RISC-V. Используемые команды.
2. Примеры обработки строк символов на основе воспроизведения некоторых функций библиотки `string.h` языка программирования `C`.
3. Организация работы с файлами в RARS.
4. Использование файлового ввода вывода для загрузки и выгрузки текстов.
5. Примеры обработки текстов (с использование уже написанных подпрограмм).

11.1 Особенности обработки символов в процессоре RISC-V

Обработку строк символов предлагается показать на примере трех программ:

- вычисление длины строки, ограниченной нулевым символом;
- вычисление длины строки, ограниченной нулевым символом или числом просматриваемых символов;
- сравнение на равенство двух строк символов, ограниченных нулем.

Думаю, что этого для демонстрации будет более чем достаточно. Предлагаемые подпрограммы реализованы в стиле Си. Поэтому можно, используя `man`, посмотреть их описание. Или в Интернете. Также у меня есть версии этих функций написанных на Си. Поэтому их можно предварительно изучить и запустить.

11.1.1 Вычисление длины строки, ограниченной нулем

Данная программа весьма проста и понятна. На Си она по сути занимает одну строчку. Однако следует отметить недостаток, связанный с контролем выхода за границы памяти. Примеры программ, демонстрирующих эту подпрограмму и тесты ее представлены в каталоге `03-strlen`.

11.1.2 Вычисление длины строки, ограниченной нулем или числом анализируемых символов

Основная идея подобного ограничения является обычно в дополнительной проверке буфера на переполнение, которое может возникать по разным причинам. Например, при наложении данных друг на друга. Это возможно в небезопасных языках программирования, к которым относятся Си и Ассемблеры. Примеры на Си и Ассемблере расположены в каталоге `04-strnlen` и демонстрируют, ситуацию, когда строка может оказаться длиннее размера буфера. В этом случае возвращается ошибка в виде значения длины, равного -1.

11.1.3 Сравнение на равенство двух строк символов, ограниченных нулем

В целом пример аналогичен предыдущим. Основная его идея заключается в обработке и сопоставлении символьных данных. Пример расположен в каталоге `05-strcmp`. Нет ничего особенного, что хотелось бы отметить. Просто еще одна функция из библиотеки языка Си.

11.2 Примеры подпрограмм и макроосов обработки строк символов

Для завершения темы подпрограмм хотелось бы рассмотреть варианты объединения всего кода в подобие библиотек подпрограмм и макроопределений. Для этого подпрограммы из предыдущих примеров вынесены в отдельные файлы (можно было их все вынести в один общий файл, что непринципиально). Пример размещен в каталоге `06-string-macro`. Помимо этого над каждой из подпрограмм сформировано макроопределение, все макроопределения собраны в файл макробиблиотеки `macro-string.m`. Файл с подпрограммой `main` демонстрирует тесты предлагаемых

макросов. Дополнительно в примере подключается макробиблиотека системных вызовов `macro-syscalls.m`. По сути это ранее рассмотренная библиотека макросов. Но планируется в ней сформировать все обращения к системным вызовам и использовать с другими программами. Думаю, что это может войти в дальнейшую методу.

11.3 Организация работы с файлами в RARS

Взаимодействие с файлами осуществляется через имитацию системных вызовов. Перечень этих вызовов приведет в таблице 11.1.

Рассмотрим использование некоторых из этих вызовов в эмуляторе RARS.

TODO. Стоит затем перенести в текст: Для организации диалогового ввода и формирования более короткого пути к рабочим каталогам могут оказаться полезными ряд опций в меню `Settings`. Опция `Popup dialog for input syscalls (5,6,7,8,12)` позволяет установить для ввода данных отдельное диалоговое окно, в котором допускается ввод до 255 символов (байт). При этом введеннае в диалоге строка копируется в консоль, дублируя по сути ввод пользователя. Опция `Derive current working directory` позволяет указывать имена файлов относительно текущей рабочей директории. Это позволяет не пользоваться абсолютными путями и писать более короткие имена файлов, устанавливаемых в диалоге или в виде строк символов.
Примечание: На использование этих опций стоит обратить внимание на семинаре. Следует отметить, что относительные (точечные) имена файлов тоже прекрасно работают. Возможно также, что в примерах имеет смысл обратить внимание и на другие системные вызовы, формирующие диалоги. Также, думаю, что стоит сформировать файл макроопределений, задающих определения для системных вызовов. Будет удобно.

11.3.1 Запись данных в файл

Я немного модифицировал пример, который присутствует в системе помощи (при описании системных вызовов). Добавил ввод с консоли имени файла для записи данных. Сдела простую обработку, заменяющую ввод пустого имени на имя файла по умолчанию. В целом же пример (каталог`write-file`) остался без каких-либо других изменений. После ввода имени файла выводится в него строка из буфера. На основе данного примера можно разобрать системные вызовы, связанные с открытием и закрытием файлов, понятие дескриптора. Можно приостановить вычисления и посмотреть в регистре номер дескриптора. Также можно разобрать работу вызова записи в файл.

```
# Sample program that writes to a new file.
.eqv NAME_SIZE 256 # Размер буфера для имени файла

.data
prompt:  .asciz "Input file path: "     # Путь до читаемого файла
```

```
er_name_mes: .asciz "Incorrect file name\n"
default_name: .asciz "testout.txt"      # Имя файла по умолчанию
# Это выводимый текст
buffer: .asciz "The quick brown fox jumps over the lazy dog."
file_name: .space NAME_SIZE # Имя читаемого файла

.text
################################################################
# Ввод имени файла с консоли эмулятора
la a0 file_name
li      a1 NAME_SIZE
li      a7 8
ecall
# Убрать перевод строки
li  t4 '\n'
la  t5  file_name
mv  t3 t5 # Сохранение начала буфера для проверки на пустую строку
loop:
lb t6  (t5)
beq t4 t6 replace
addi t5 t5 1
b   loop
replace:
beq t3 t5 default # Установка имени введенного файла
sb  zero (t5)
mv   a0, t3  # Имя, введенное пользователем
b out
default:
la   a0, default_name # Имя файла по умолчани
################################################################
out:
# Open (for writing) a file that does not exist
li   a7, 1024     # system call for open file
li   a1, 1        # Open for writing (flags are 0: read, 1: write)
ecall             # open a file (file descriptor returned in a0)
mv   s6, a0       # save the file descriptor
################################################################
# Write to file just opened
li   a7, 64       # system call for write to file
mv   a0, s6       # file descriptor
la   a1, buffer   # address of buffer from which to write
li   a2, 44       # hardcoded buffer length
ecall             # write to file
################################################################
```

```
# Close the file
li   a7, 57       # system call for close file
mv   a0, s6       # file descriptor to close
ecall             # close file
################################################################
```

> **Примечание**: Отступы нарушены. Поэтому лучше использовать примеры непосредственно из файла кода.

11.3.2 Чтение данных из файла

Обратная задача чтения данных из файла связана с тем, что необходимо иметь буфер соответствующего размера (каталог `read-file`). Также необходимо проверять имя файла при открытии на наличие. Это делается просто по возврату -1 системным вызовом `Open`. В данном случае файл читается в буфер. Делается попытка заполнить весь буфер за один раз. Если файл короче, то возвращаемое количество прочитанных байт будет меньше длины буфера. Зная это число, в конце можно поставить нулевой ограничитель строки и вывести файл соответствующим системным вызовом. Если же файл имеет длину, большую, чем буфер, он заполнит буфер только своей считанной частью. Пример подводит к тому, что слишком большие файлы, которые не умещаются в буфер (а часто мы не знаем размер файла), должны читаться по частям. Возможно с расширением буфера с использованием, например, динамической памяти.

> **Примечание**: Это будет следующий пример по чтению-записи файлов. В текущем примере вложен файл `strcmp.c` для демонстрации ввода файла, размещаемого в буфере. Файлом, который не размещается в буфере, является код программы.

11.4 Примеры обработки текстов

Программа представленная в каталоге `09-load-text`, осуществляет ввод данных из большого файла, их отображение в консоли и вывод данных в другой файл. Имена входного и выходного файлов задаются в диалоге. Ее особенностью является ограниченный по размеру буфер, используемый для чтения из файла. Поэтому чтение данных осуществляется в цикле. В этом же цикле осуществляется динамическое расширение данных под читаемые данные в процессе их поступления. В принципе подход может быть использован для любых файлов, но в примере осуществляется вывод данных в консоль. Поэтому он ориентирован только на текстовые файлы.

Наряду с этим в программе расширена библиотека макросов, поддерживающая системные вызовы. Также для демонстрации используется внешняя подпрограмма, определяющая длину сформированной из файла строки. В целом это неплохая заготовка для выполнения задания по строкам, так как она полностью реализует чтение и запись больших файлов.

В подкаталог `data` я добавил примеры различных файлов, которые можно читать. Обычно в этот же подкаталог я сохраняю результаты записи. При этом настройки эмулятора, которые сейчас использую, описаны в начале этого раздела. То есть, установлен текущий каталог в качестве рабочего каталога. Также для ввода консольных данных использую всплывающее окно, так как в ряде случаев оно удобно (при отсутствии подсказки), а результат ввода все равно потом дублируется в консоли.

11.5 Домашнее задание

Написать подпрограмму, осуществляющую копирование строки символов аналогично функции strncpy языка программирования C. Протестировать функцию на различных комбинациях данных. Ознакомиться с функцией можно в системе справки по библиотеке языка C, которая имеется в различных источниках информации. Исходные данные для тестирования задавать как при вводе с консоли, так и с использованием строк символов в разрабатываемой программе (по аналогии с программами, рассмотренными на семинаре). Подпрограмму вынести в отдельный файл.

Опционально до +2 баллов

Дополнительно к подпрограмме разработать соответствующий макрос, расширив тем самым макробиблиотеку строк символов.

Срок сдачи задания: через неделю после его выдачи. К следующему семинарскому занятию своей группы.

Таблица 11.1 — Системные вызовы, обеспечивающие работу с файлами в симуляторе RARS

Вызов	**Номер**	**Описание**	**Вход**	**Выход**
`Close`	57	Close a file	a0 = the file descriptor to close	N/A
`LSeek`	62	Seek to a position in a file	a0 = the file descriptor a1 = the offset for the base a2 is the begining of the file (0), the current position (1), or the end of the file (2)	a0 = the selected position from the beginning of the file or -1 is an error occurred
`Read`	63	Read from a file descriptor into a buffer	a0 = the file descriptor a1 = address of the buffer a2 = maximum length to read	a0 = the length read or -1 if error
`Write`	64	Write to a filedescriptor from a buffer	a0 = the file descriptor a1 = the buffer address a2 = the length to write	a0 = the number of charcters written
`Open`	1024	Opens a file from a path Only supported flags (a1) are read-only (0), write-only (1) and write-append (9). write-only flag creates file if it does not exist, so it is technically write-create. write-append will start writing at end of existing file.	a0 = Null terminated string for the path a1 = flags	a0 = the file decriptor or -1 if an error occurred

12 Семинар 12.

13 Семинар 13.

14 Семинар 14.

15 Семинар 15. Обработка исключений

Целью семинара является разбора выполненного задания. Обсуждение основных недостатков и их учет при выполнении следующих заданий.

Примерный перечень вопросов:

1. Ручная проверка, это не проверка олимпиадных задач. Идет просмотре исходных и компилируемых текстов (после рефакторинга), запуск вручную. Поэтому необходимы элементарные юзабилити и устранение «запаха».
2. Проверки на корректность командной строки. Наличие аргументов. Наличие нужных файлов.
3. Проверка диапазонов вводимых массивов.
4. Хорошо бы при написании ассемблерного кода использовать символические обозначения смещений для локальных переменных.
5. Ответы на вопросы и претензии.

Все это в режиме импровизации.

16 Семинар 16. Ввод–вывод данных. Поллинг. Программирование ввода–вывода

Целью семинара является изучение взаимодействия процессора с устрокйствами ввода–вывода в режимет опроса состояния внешних устройств.

Изучение ведется на моделях внешних устройств, подключенных к эмулятору. Примерный перечень вопросов:

1.

2.

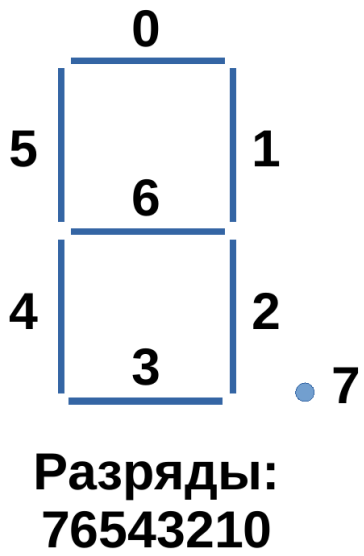


Рисунок 16.1 — Отображение сегментов индикатора в данных, передаваемых устройству

16.1 Домашнее задание

17 Семинар 17. Таймер. Прерывания по таймеру

Целью семинара является изучение базовых механизмов синхронизации POSIX Threads, обеспечивающих совместное безконфликтное использование ресурсов. Необходимо провести анализ примеров, демонстрирующих различные методы синхронизации и того, что может получиться при отсутствии синхронизации.

Примерный перечень вопросов:

1. Пример 01. Перемножение матриц. Лучше использовать пример 07 (07. Перемножение матриц. Добавление мьютексов для синхронизации очереди вывода данных), так как там имеется больше информации для вывода данных. Первый вариант связан с тем, что можно после запуска корректной программы закомментировать мьютексы и сопоставить результаты. Возможно, из-за того, что вычислительная нагрузка небольшая и потоки отрабатывают быстро, весь вывод данных и формирование матрицы с результатами на выводе произойдут корректно. Можно запустить программу без мьютексов нескольно раз. Тогда, раскомментировав мьютексы, можно увеличить матрицы, например до 10*10. Создать десяток потоков и повторить еще раз. После этого раскомментировать и посмотреть, произойдет ли изменение вывода. То есть, разобраться с тем, что не всегда очевидно сразу, что синхронизация может понадобиться.

2. Пример 02 (08. Задача о кольцевом буфере. Использование семафоров для синхронизации потоков). Здесь время не играет определенной роли. Поэтому после демонстрации протокола работающей программы можно посмотреть, что будет если закомментировать по очереди семафоры и мьютексы. Помимо этого можно изменить поведени системы. Например, ускорить писателей и читателей. Или резко увеличить тех или других без изменения временных интервалов. В этих ситуациях интересно посмотреть ожидание, когда буфер будет либо полон, либо пуст.

3. Пример 03 (09. Задача о кольцевом буфере. Использование условных переменных для синхронизации потоков). Но с другими синхропримитивами. Просто акцентировать на специфике и отличии условных переменных от семафоров. Но так же можно поиграться.

4. Пример 04 (10. Читатели-писатели с общим одномерным массивом. Использование блокировок).Если успеваем, то можно зацепить этот и следующий. Или перенести на следующий семинар, увязав на нем с OpenMP. Здесь, наверное,

можно поиграться только с разными интенсивностями работы читателей и писателей. Тем более, что читатели ничего не меняют. Можно поиграться числом разных потоков или интенсивностью их обращения.

5. Пример 05 (11. Использование барьеров для синхронизации данных). Пример интересен именно тем, где и как убирать или ставить барьеры. Именно они влияют на синхронизацию двух разделенных массивов. Вариант одного барьера закомментирован. С двумя другими можно тоже поиграться, посмотрев разные варианты. Вплоть до отсутствия барьеров. Мьютексы отвечают за вывод. Поэтому их трогать смысла особого нет. Но тоже можно посмотреть, если будет время.

Во всех альтернативных программах можно использовать код как на Си, так и на плюсах. Они работают одинаково.

Пусть студенты сами сделают прогоны и сформируют результаты.

17.1 Домашнее задание

Перенести задание с предыдущего занятия. Пусть проведут дополнительный анализ на необходимость использования синхронизации.

Что было на предыдущем:

Сделать выслать письменный реферат, в котором оценить возможность параллельного выполнения задания №1 на обработку целочисленных одномерных массивов и использованием методов синхронизации. По своему сделанному варианту. Программу писать не нужно.

18 Семинар 18. Программирование ввода–вывода. Использование прерываний

Целью семинара является продолжение изучения базовых механизмов синхронизации POSIX Threads, обеспечивающих совместное безконфликтное использование ресурсов. Необходимо провести анализ примеров, демонстрирующих различные методы синхронизации и того, что может получиться при отсутствии синхронизации.

Изучение примеров с OpenMP, прилагающихся к лекциям. Предыдущие и эти примеры достаточно короткие. Поэтому можно просто пробежаться. Вполне возможно, что OpenMP придется установить. Обычно пакет openmp. На Убунте не пробовал. Попробую — сообщу. Все эти примеры я планирую показать на лекции. Но для раздолбаев, которые на лекции не ходят, может оказаться полезным. Заодно все узнают, как установить библиотеку.

После этих примеров обсудить результаты выполнения заданий.

Примерный перечень вопросов (часть перенесена из предыдущего семинара, так как не рассматривалась на лекции):

1. Пример 04 (10. Читатели-писатели с общим одномерным массивом. Использование блокировок). Здесь, наверное, можно поиграться только с разными интенсивностями работы читателей и писателей. Тем более, что читатели ничего не меняют. Можно поиграться числом разных потоков или интенсивностью их обращения. В целом основной акцент на самом механизме организации блокировок и поведении участников. В целом ничего хитрого в пояснении и использовании я не вижу.

2. Пример 05 (11. Использование барьеров для синхронизации данных). Пример интересен именно тем, где и как убирать или ставить барьеры. Именно они влияют на синхронизацию двух разделенных массивов. Вариант одного барьера закомментирован. С двумя другими можно тоже поиграться, посмотрев разные варианты. Вплоть до отсутствия барьеров. Мьютексы отвечают за вывод. Поэтому их трогать смысла особого нет. Но тоже можно посмотреть, если будет время. Ключевая идея связана с независимостью данных и использования барьеров для согласованного изменения своих массивов.

3. Пример 06 (01. Многопоточный вывод "Hello World"). Данный пример прост. На нем можно увидеть, как работает прагма `#pragma omp parallel`. Можно ее отключить и посмотреть, что получится. Основная идея — рассмотреть две функции, которые определяют число возможных потоков по ядрам системы и номер текущего потока. Каждый поток обрабатывается независимо и выводит свои данные.

4. Пример 08 (02. Использование критической секции. Шаг 1). Пример показывает использование критической секции `#pragma omp critical`. В данном случае общая переменная изменяется произвольно, а синхронизация осуществляется по выводу. Как раз можно оценить, что происходит с этой переменной при неправильной установке критической секции.

5. Пример 09 (03. Использование критической секции. Шаг 2). В этом примере критическая секция установлена на переменной. Но не на выводе данных. Тоже полезно посмотреть рассогласование вывода. Как и в предыдущем случае можно запустить несколько раз, чтобы отследить недетерминированность.

6. Пример 10 (04. Использование критической секции. Шаг 3). Окончательный пример по критической секции. Имеется выбор конкретного числа потоков. Можно поиграться с этим. Также можно посмотреть потоки с использованием системного монитора. Также там описано использование редукции. Можно поиграться с различными прагмами распараллеливания.

7. Пример 11 (05. Вычисление интеграла с использованием редукции). Разные варианты и число потоков для просмотра. Тоже можно поиграться с режимами.

Во всех альтернативных программах можно использовать код как на Си, так и на плюсах. Они работают одинаково.

Пусть студенты сами сделают прогоны и сформируют результаты.

18.1 Домашнее задание

Установить на домашней системе openMP и прислать отчет (в виде сканов), демонстрирующих выполнение программы вычисления интеграла с использованием openMP. Использовать свои данные, изменив в программе функцию, используемую для интегрирования и интервалы интегрирования. Три варианта запуска с разными интервалами.

19 Семинар 19. Микроархитектура. Предсказание переходов. Кеширование

Целью семинара является продолжение изучения особенностей организации и использования интерфейса передачи сообщений на основе библиотеки Message Passing Inteface (MPI).

Изучение примеров с MPI, прилагающихся к лекциям. Предыдущие и эти примеры достаточно короткие. Поэтому можно просто пробежаться. Вполне возможно, что MPI придется установить. Пока на Убунте у меня не удалось настроить OpenMPI, хотя сам пакет установился проблемы оказались с настройкой путей к заголовочным файлам. Пока же не проходит компиляция. Не находит этих путей. Конфигурировать пробовал, но пока результа не достигнут. Буду пробовать еще. Не получится — попробую установить и настроить MPICH. Это еще одна версия MPI, доступная в Убунте. Когда-то она была основной в Linux. Результаты сообщу. Текст подправлю. Также хочу попробовать под Virtual Box Simply Linux.

Есть еще один дежурный вариант, который может устроить всех: установка MPI на рабочей системе. Он есть как под Виндой, так и под Яблоком. Это в принципе можно учесть при раздаче домашнего задания. Если у тебя получится под Линуксом — попробуй Винду. Можно потом прописать в семинарах. Поищи ссылки в сети. Можешь их вставить в свою презентацию и выложить в семинар. Пусть делают ДЗ на своих ОС. Это непринципиально.

Кстати, отсутствующие а ЛМС семинары я постараюсь обозначить по темам и можно туда включить твои презентации. Лекционные примеры, наверное, переносить не буду, но в последующем что-то можно поискать и добавить. В частности, домашние задания.

Если не получится ничего установить в классе, то можно, как я планирую, посмотреть функционирование кода на проекторе со своего компьютера.

Примерный перечень вопросов (часть перенесена из предыдущего семинара, так как не рассматривалась на лекции):

1. Пример 01 (01. Простое распределенное приложение: "Привет от MPI"). На нем име можно поиграться как с параметрами файла хостов, так и числом запускаемых процессов. Показать, что при ограниченном числе слотов в кон-

фигурационном файле запуск большего числа процессов не проходит. То же можно просмотреть и без указания слотов в команде, когда раределение процессов по узлам идет по кругу, но до предела, определяемого умолчанием. Расширить этот предел до «неограниченного» числа можно использованием опции `--oversubscribe`. Здесь удобно тем, что при превышении числа слотов, указанного в конфигурационном файле, выдается сообщение, подсказывающее о вариантах использования. Еще можно попытаться задать огромное число процессов. Например, 1000. Система должна ругнуться на ограниченность процессов и пайпов, которые она может запустить. Особенно на пайпы, которые, видимо используются внутри MPI.

2. Пример 02 (02. MPI. Пример передачи и приема сообщений). Пример ориентирован на взаимодействие только двух процессов по простейшему блокирующему передаче и приему. В данном случае интересно обсудить, каким образом осуществляется ограничение процессов. Можно также попробовать предварительно до приема прочитать статус. Посмотрет (count) переданное число байт. Но в целом это базовая схема для различных других вариантов использования. Как варианты, можно предложить сделать модификации, в которых вместо строки осуществляется передача целых, действительных чисел.

3. Пример 03 (03. MPI. Использование барьеров). Барьерная задача очень проста. Но в принципе на ней можно увидеть, что будет, если барьер убрать. Тогда последнее сообщение, стоящее после барьера, будет выведено не в конце.

4. Пример 04 (04. MPI. Широковещательная рассылка). Детально разобрать пример, программы, осуществляющей широковещательную рассылку. Можно раскомментировать выводы, демонстрирующие количество процессов коммуникаторах. Позапускать программу с разным числом процессов (начиная с 1). Четным и нечетным. Оценить поведение.

5. Пример 05 (05. MPI. Вычисление суммы квадратов). Как вариант из предыдущих семинаров: можно увеличить нагрузк на цикл, добавив, например, вычисление квадратного корня или чего-то еще. Также интересно посмотреть вычисление на разном числе узлов с разным и сопоставить время вычислений при одинаковом числе элементов в массиве.

6. Пример 06 (06. MPI. Вычисление числа Пи). Пример завершает работу при вводе нуля. На нем можно посмотреть широковещательную рассылку, использование суммирования и свертки с применением MPI. Также интересно несколько раз задать разную точность вычисления путем задания количества интервалов. Сопоставить время получения результата.

19.1 Домашнее задание

Установить на домашней системе MPI и прислать отчет (в виде сканов), демонстрирующих выполнение программы осуществляющей пересылки текстовых сообщений между тремя потоками в соответствии со схемой, представленной на слайде 19 лекции по MPI. За основу можно взять пример 02.

20 Семинар 20. Микроархитектура. Конвейеризация

Целью семинара является сопоставление методов псевдопараллельного и параллельного програмирования на примере событийного подхода, используемого в различных системах (C#, Qt и др.) и многопоточного программировния на основе `pthread`. То есть, для упрощения вычислений и привязки к заданию свести тему к многопочности, а не параллелизму в общем виде.

Честно говоря, у меня начинается потеря связи между лекционным материалом и его отображением на практике, что связано с невозможностью воспроизвести на текущем оборудовании ряда задач, которые еще могу рассмотреть на лекции. Поэтому я решил попробовать посвятить семинар материалу, который можно использовать при выполнении четвертого задания, с одной стороны. С другой стороны этот материал можно также использовать для формирования представления о методах программирования, используемых в современных последовательных системах для организации псевдопараллельных вычислений.

Для этого я предлагаю рассмотреть библиотеку Qt и тот подход, который там используется для взаимодействия отдельных функций в асинхронном режиме. На примере программы имитации роста растений и поедающих эту растительность животных рассмотреть поведение системы и попытаться отобразить его (пока концептуально) в поведение многопоточной системы, аналогичной тем программам, которые реализуются в четвертом задании.

Оставшееся время (или, если что-то не пойдет с этим вариантом) можно посвятить разбору третьего задания. Я думаю, что сильно в это погружаться не стоит, так как ключевым является переход к концептуальному обсуждению схемы задачи, описывающей параллелизм и методов ее реализации в реальной многопоточной среде. То есть, рассмотреть на уровне общей структуры, не забираясь в само кодирование. Чтобы при выполнении своих заданий были некоторые стереотипы, куда идти.

Примерный перечень вопросов:

1. Краткий рассказ об особенностях событийного программирования на примере практически любых систем. По имеющейся информации нынешние второкурсники изучали основы обработки событий в C#. Проблемы могут возникнуть с теми, кто пришел с потока ИСП РАН. Можно кратко охарактеризовать на пальцах. Ты ведь знаешь C# и другие. Можно сориентироваться и на библио-

теки Qt. Есть у меня соответствующая лекция для первокурсников. Она будет в 13-м семинаре, как и соответствующие примеры. Обсудить, что программа по сути формируется как набор псевдопараллельных компонент, связанных между собой «проводами», по которым передаются сообщения, являющиеся по сути сигналами. Реализация этого механизма осуществляется за счет формирования очередей сообщений, из которой некоторым диспетчером осуществляется последовательная выборка с запуском очередной функции или метода. При этом ОС система дополнительно управляет переключением потоков, отвечающих за различные другие функции, что в данном случае является несущественным для рассматриваемой программы, но поддерживает общий параллелизм. Псевдопараллелизм поддерживается тем, что сами выполняемые функции не являются нагруженными задачей. **Но сильно погружаться (заморачиваться) не нужно. На уровне интуиции.** *Можно во время демонстрации примеров.*

2. Далее (или сразу же) можно привести примеры игры elife, имитирующей поедание растущей травы. Есть (выложена) чужая реализация на JS и моя на Qt. Положил проекты на qmake и cmake. Исходя из этого можно обсудить внутреннюю структуру, например, обработки на JS. Запустить по ходу можно несколько раз. Можно одновременно оба на одном экране.

3. После этого можно приступить в вариантам возможной реализации с применением многопоточного программирования. Фиксируя, например, на доске, предлагаемые варианты. Пусть студенты изначально сами предложат версии того, как можно написать параллельную (многопоточную) программу. На первом этапе можно не вдаваться в детали, а породить именно варианты возможного воплощения сценария в потоках. То есть, то что им нужно сделать (сформировать хотя бы один сценарий) по заданию 4. Возможные варианты:
   - Один поток — поведение всей еды, другой — поведение поедателей. Синхронизация при доступе к поляне.
   - Каждый поток определяет один элемент еды или поедателя. Поляна — набор данных вокруг которого идет синхронизация. Здесь речь пойдет о создании потоками новых потоков и завершении тех потоков при соответствующих условиях.
   - Каждая клетка поляны - поток, который содержит в виде данных одно из состояний (пусто, еда, поедатель). Здесь возникает синхронизации с соседними клетками.
   - Возможны и другие варианты. Пусть в начале предлагают свои. После этого, если останутся, соответствующие из выше предложенных.

4. После этого можно обсудить специфику каждого из сценариев. То есть, как и какие порождать потоки. Как их синхронизировать. Где проще организовать. Какой вариант обладает бОльшим параллелизмом. Какие варианты синхронизации возникают. В принципе здесь могут появиться и подварианты. Можно

попробовать устроить конкурс: кому какой вариант больше нравится. И разбить анализ по подгруппам. Что у какой получится...

5. Доводить до реализации не стоит. Но если будет время, то детали каких-то вариантов можно проработать. То есть, именно порождение концептуальных решений.

Остаток занятия или перерыв можно оставить под обсуждение результатов выполнения задания 3

20.1 Домашнее задание

Скорее всего выдавать не стоит, так как дедлайн по заданию 4. Пусть используют идеи в своем варианте.

21 Семинар 21. Поддержка многозадачности. Виртуализация. Многоядерность

Целью семинара является изучение основных принципов подхода, направленного на создание архитектурно-независимых параллельных программ. Основная идея связана с тем, чтобы при разработке программ сделать основной акцент на функциональные и алгоритмические зависимости, максимально абстрагировавшись от использования вычислительных ресурсов.

Для изучения подхода предлагается использовать функционально-потоковую парадигму параллельного программирования. Ее описание, а также язык и IDE представлены на сайте по ссылке: `http://softcraft.ru/parallel/fpp/`. В рамках семинара предлагается изучить основные идеи подхода и разобрать особенности построения функционально потоковых параллельных (ФПП) программ. Среда функционирует под Виндой. Программы и сама среда выложены в ЛМС.

Примерный порядок проведения семинара:

1. Первоначально предлагается установить и запустить среду. Посмотреть на ее основные элементы управления. Можно практически сразу загрузить файл с множеством функций и на его примере рассмотреть особенности конструкции языка и их использование для описания параллельных вычислений. При этом имеет смысл пользоваться тем описанием, которое лежит на сайте. В целом там вполне понятное описание. Если что по нему непонятно, то могу пояснить.

2. Далее можно начать понемногу рассматривать основные фунции и оснобенности работы с ними по порядку. На примере функции

```
ParASMD << funcdef Param
{
 Param:[+,-,*,/,%]:(.) >>return
};
```

можно показать, как задается одновременность в обработке одного потока данных. Аргументами функции, задаваемым в табе аргументов может слу-

жить любой двухэлементный список. Можно, используя примеры ниже, запускать функцию с любыми тестовыми данными. В приведенных тестовых примерах функции как бы запускаются внутри параллельного списка. То есть, «одновременно». Одновременность была бы при наличии соответствующего параллельного интерпретатора. Можно также пробежаться с пошаговым отладчиком по фунции, чтобы посмотреть как он работает.

3. На функции вычисления абсолютной величины можно рассмотреть каким образом задаются условия и организуются ветвления.

```
Abs << funcdef Param
{
 ({Param:-},{Param}):[(Param,0):(<,=>):?]:.>>return
};
```

Особенность в том, что нет ветвления потоков, а имеется их выборка из альтернативных вариантов. То есть, поток всегда есть и сводися в общий путь. При этом для того, чтобы ненужные вычисления не выполнялись, используется задержанный список (`{...}`.

4. При вычислении квадратного корня используется обычный последовательный алгоритм. В принципе здесь можно рассмотреть, как он отображается на информационный граф, когда отстутствует явное управление вычислениями, а используется управленипе по готовности данных. Можно в принципе особо и не рассматривать.

5. Далее можно остановиться на организации параллельной рекурсии.

```
VSum << funcdef Param
{
 Len<<Param:|;
 return<< .^[((Len,2):[<,=,>]):?]^
 ({Param:[]},
 {Param:+},
 {block{
        OddVec<< Param:[(1,Len,2):..];
        EvenVec<< Param:[(2,Len,2):..];
        ([OddVec,EvenVec]:VSum):+ >>break}
 })
};
```

Пояснить специфику разделения списка данных (вектора) на два посписка, на каждым из которых снова может выполняться дихотомия. В данном случае идет разделение по четным и нечетным элементам. До списка с одним

или двумя элементами. Над последним выполняется суммирование, результат поднимается вверх и суммируется с результатом из другого подсписка.

6. Аналогичный прием можно обобщить на функции высшего порядка.

```
VHigh << funcdef Param
{
 Len<<Param:1:|;
 Func<<Param:2;
 return<< .^[((Len,2):[<,=,>]):?]^
 ({Param:1:[]},
 {Param:1:Func},
 {block{
        OddVec<< Param:1:[(1,Len,2):..];
        EvenVec<< Param:1:[(2,Len,2):..];
        ([(OddVec, Func),(EvenVec,Func)]:VHigh):Func >>break}
 })
};
```

   Основная идея здесь заключается в передаче функции в качестве аргумента. Это используется в тестовых примерах.

7. Далее приведены различные функции работы с векторами. Их можно рассмотреть более бегло, останавливаясь только на конструкциях вызывающих вопросы. Ответы на большую часть вопросов можно получить по ссылке, указанной выше.

8. И на закуску, если останется время, можно рассмотреть параллельную интерпретацию задачи о Ханойской башне и сортировки, представленные в виде отдельных файлов.

21.1 Домашнее задание

Скорее всего выдавать не стоит, так как дедлайн остается занятие для разбора полетов и то не у всех.

22 Семинар 22.

23 Семинар 22. Мультипроцессоры. Специализированные процессоры

Целью семинара является изучение основных принципов подхода, направленного на создание архитектурно-независимых параллельных программ. Основная идея связана с тем, чтобы при разработке программ сделать основной акцент на функциональные и алгоритмические зависимости, максимально абстрагировавшись от использования вычислительных ресурсов.

Для изучения подхода предлагается использовать функционально-потоковую парадигму параллельного программирования. Ее описание, а также язык и IDE представлены на сайте по ссылке: `http://softcraft.ru/parallel/fpp/`. В рамках семинара предлагается изучить основные идеи подхода и разобрать особенности построения функционально потоковых параллельных (ФПП) программ. Среда функционирует под Виндой. Программы и сама среда выложены в ЛМС.

Примерный порядок проведения семинара:

1. Первоначально предлагается установить и запустить среду. Посмотреть на ее основные элементы управления. Можно практически сразу загрузить файл с множеством функций и на его примере рассмотреть особенности конструкции языка и их использование для описания параллельных вычислений. При этом имеет смысл пользоваться тем описанием, которое лежит на сайте. В целом там вполне понятное описание. Если что по нему непонятно, то могу пояснить.

2. Далее можно начать понемногу рассматривать основные фунции и оснобенности работы с ними по порядку. На примере функции

```
ParASMD << funcdef Param
{
 Param:[+,-,*,/,%]:(.) >>return
};
```

можно показать, как задается одновременность в обработке одного потока данных. Аргументами функции, задаваемым в табе аргументов может слу-

жить любой двухэлементный список. Можно, используя примеры ниже, запускать функцию с любыми тестовыми данными. В приведенных тестовых примерах функции как бы запускаются внутри параллельного списка. То есть, «одновременно». Одновременность была бы при наличии соответствующего параллельного интерпретатора. Можно также пробежаться с пошаговым отладчиком по фунции, чтобы посмотреть как он работает.

3. На функции вычисления абсолютной величины можно рассмотреть каким образом задаются условия и организуются ветвления.

```
Abs << funcdef Param
{
 ({Param:-},{Param}):[(Param,0):(<,=>):?]:.>>return
};
```

Особенность в том, что нет ветвления потоков, а имеется их выборка из альтернативных вариантов. То есть, поток всегда есть и сводися в общий путь. При этом для того, чтобы ненужные вычисления не выполнялись, используется задержанный список (`{...}`.

4. При вычислении квадратного корня используется обычный последовательный алгоритм. В принципе здесь можно рассмотреть, как он отображается на информационный граф, когда отстутствует явное управление вычислениями, а используется управленипе по готовности данных. Можно в принципе особо и не рассматривать.

5. Далее можно остановиться на организации параллельной рекурсии.

```
VSum << funcdef Param
{
 Len<<Param:|;
 return<< .^[((Len,2):[<,=,>]):?]^
 ({Param:[]},
 {Param:+},
 {block{
        OddVec<< Param:[(1,Len,2):..];
        EvenVec<< Param:[(2,Len,2):..];
        ([OddVec,EvenVec]:VSum):+ >>break}
 })
};
```

Пояснить специфику разделения списка данных (вектора) на два посписка, на каждым из которых снова может выполняться дихотомия. В данном случае идет разделение по четным и нечетным элементам. До списка с одним

или двумя элементами. Над последним выполняется суммирование, результат поднимается вверх и суммируется с результатом из другого подсписка.

6. Аналогичный прием можно обобщить на функции высшего порядка.

```
VHigh << funcdef Param
{
 Len<<Param:1:|;
 Func<<Param:2;
 return<< .^[((Len,2):[<,=,>]):?]^
 ({Param:1:[]},
 {Param:1:Func},
 {block{
        OddVec<< Param:1:[(1,Len,2):..];
        EvenVec<< Param:1:[(2,Len,2):..];
        ([(OddVec, Func),(EvenVec,Func)]:VHigh):Func >>break}
 })
};
```

   Основная идея здесь заключается в передаче функции в качестве аргумента. Это используется в тестовых примерах.

7. Далее приведены различные функции работы с векторами. Их можно рассмотреть более бегло, останавливаясь только на конструкциях вызывающих вопросы. Ответы на большую часть вопросов можно получить по ссылке, указанной выше.

8. И на закуску, если останется время, можно рассмотреть параллельную интерпретацию задачи о Ханойской башне и сортировки, представленные в виде отдельных файлов.

23.1 Домашнее задание

Скорее всего выдавать не стоит, так как дедлайн остается занятие для разбора полетов и то не у всех.

Заключение

Продолжение следует...

Приложение А Задание 1. Целочисленная арифметика и массивы

Разработать программу, которая получает одномерный массив $\mathbf{A}_N$, после чего формирует из элементов массива **A** новый массив **B** по правилам, указанным в варианте, и выводит его. Память под массивы может выделяться статически, на стеке, в области кучи по выбору разработчика. При решении задачи необходимо использовать подпрограммы для реализации ввода, вывода и формирования нового массива. Ввод-вывод элементов реализовать в окно выполнения эмулятора с использованием его системных вызовов.

1. Сформировать массив **B** из положительных элементов массива **A**.
2. Сформировать массив **B** только из тех элементов массива **A**, которые не совпадают с его первым и последним элементами.
3. Сформировать массив **B** из сумм соседних элементов **A** по следующим правилам: $\mathbf{B}_0 = \mathbf{A}_0 + \mathbf{A}_1$, $\mathbf{B}_1 = \mathbf{A}_1 + \mathbf{A}_2$, **...**
4. Массив **B** формируется по следующим правилам:
   - $\mathbf{B}_i$ **= 1, если** $\mathbf{A}_i > \mathbf{0}$,
   - $\mathbf{B}_i$ **= -1, если** $\mathbf{A}_i < \mathbf{0}$,
   - $\mathbf{B}_i$ **= 0, если** $\mathbf{A}_i = \mathbf{0}$.
5. Сформировать массив **B**, состоящий из элементов массива **A**, значение которых не совпадает с дополнительно введённым числом **X**.
6. Сформировать массив **B**, состоящи из элементов массива **A**, значения которых кратны введенному числу **X**.
7. Сформировать массив **B** из индексов положительных элементов массива **A**.
8. Сформировать массив **B** по следующим правилам:
   - **если** $\mathbf{A}_i > \mathbf{5}$**, то увеличить элемент** $\mathbf{B}_i$ **на 5,**,
   - **если** $\mathbf{A}_i <$ **-5, то уменьшить** $\mathbf{B}_i$ **на 5,**,
   - **остальные** $\mathbf{B}_i$ **обнулить.**.

9. Сформировать массив **B** из нечётных элементов массива **A**.

10. Сформировать массив **B** из отрицательных элементов массива **A**, расположенных обратном порядке.

11. Сформировать массив **B** из элементов **A**, расположенных в обратном порядке, исключая первый положительный элемент.

12. Сформировать массив **B** из элементов массива **A**, исключив первый положительный и последний отрицательный элементы.

13. Сформировать массив **B** из элементов массива **A**, за исключением элементов, значения которых совпадают с минимальным элементом массива **A**.

14. Сформировать массив **B** из элементов массива **A** заменой всех отрицательных значений на максимум из массива **A**.

15. Сформировать массив **B** из элементов массива **A** заменой всех нулевых элементов значением минимального элемента.

16. Сформировать массив **B** из элементов массива **A**, заменой на среднее арифметическое тех значений, которые больше среднего арифметического.

17. Сформировать массив **B** из элементов массива **A**, расположенных после последнего положительного элемента.

18. Сформировать массив **B** из элементов массива **A** уменьшением всех элементов, расположенных до первого положительного, на **5**.

19. Сформировать массив **B** из элементов массива **A** заменой нулевых элементов, предшествующих первому отрицательному, единицей.

20. Сформировать массив **B** из элементов массива **A** перестановкой местами минимального и первого элементов.

21. Сформировать отсортированный по возрастанию массив **B** из элементов массива **A**.

22. Сформировать отсортированный по убыванию массив **B** из элементов массива **A**.

23. Сформировать массив **B**, элементы которого являются расстояниями пройденными телом при свободном падении на землю за время в секундах, указанное в массиве **A**. Решение получить в целых числах, приняв ускорение свободного падения за **10**.

24. Сформировать массив **B** из элементов массива **A** поменяв местами элементы, стоящие на чётных и нечётных местах: $\mathbf{A}_0 \leftrightarrow \mathbf{A}_1$**;** $\mathbf{A}_2 \leftrightarrow \mathbf{A}_3$ **...**

25. Сформировать массив **B** из элементов массива **A** заменив все положительные числа значением **2**, а отрицательные — увеличить на **5**.

26. Сформировать массив **B** из сумм трех соседних элементов массива **A**, сумма значений которых максимальна. Если элементов в массиве **A** менее трёх, то заполнить массив **B** нулями.

27. Сформировать массив **B** из элементов массива **A**. Элементы массива **A**, оканчивающиеся цифрой **4**, уменьшить вдвое.

28. Сформировать массив **B** из элементов массива **A**, которые образуют неубывающую последовательность. Неубывающей последовательностью считать элементы идущие подряд, которые равны между собой или каждый последующий больше предыдущего.

29. Сформировать массив **B** из произведения соседних элементов **A** по следующему правилу: $\mathbf{B}_0 = \mathbf{A}_0$ * $\mathbf{A}_m$, $\mathbf{B}_1 = \mathbf{A}_1$ * $\mathbf{A}_m$, ..., где **m** – либо номер первого четного отрицательного элемента массива A, либо номер последнего элемента, если в массиве A нет отрицательных элементов.

30. Сформировать массив **B** из тех элементов массива **A**, которые больше, чем элементы, стоящие перед ними.

31. Сформировать массив **B** из элементов массива **A** в следующем порядке: сначала заполняем массив **B** числами, стоящими на нечетных местах, а затем - стоящие на четных местах в массиве **A**.

32. Сформировать массив **B** из элементов массива **A**, которые меньше суммы элементов, расположенных на четных местах.

33. Сформировать массив **B** на основе элементов массива **A**, полученных как разность соседних элементов.

34. Сформировать массив **B** из элементов массива **A** заменив элементы на четным местах суммой всех положительных элементов, а элементы на нечетных местах суммой отрицательных элементов.

35. Сформировать массив **B** из элементов массива **A** сгруппировав положительные элементы массива **A** в начале массива **B**, а отрицательные — в конце.

36. Сформировать массив **B** из элементов массива **A** сгруппировав элементы с четными индексами в начале массива **B** , а элементы с нечетными индексами сгруппировать в конце массива **B**.

37. Сформировать массив **B** из элементов массива **A** в следующем порядке: элементы с индексами $\mathbf{i} \leq (\mathbf{N}+\mathbf{1})/\mathbf{2}$ переместить на позиции с четными индексами массива **B** с сохранением их исходного порядка относительно друг друга, а оставшиеся элементы ($\mathbf{i} > (\mathbf{N}+\mathbf{1})/\mathbf{2}$) разместить на позициях с нечетными индексами массива **B** также с сохранением их исходного порядка.

38. Сформировать массив **B** из элементов массива **A**, которые одновременно имеют четные и отрицательные значения.

39. Сформировать массив **B**, элементы которого являются площадью прямоугольников со сторонами указанные в массиве **A**: $\mathbf{B}_i = \mathbf{A}_i$ ***** $\mathbf{A}_{i+1}$.

40. Сформировать массив **B** из суммы соседних элементов **A** по следующему правилу: $\mathbf{B}_0 = \mathbf{A}_0$**,** $\mathbf{B}_1 = \mathbf{A}_0 + \mathbf{A}_1$**, ...,** $\mathbf{B}_m = \mathbf{A}_0 +$ **...** $+ \mathbf{A}_m$, где **m** – номер первого элемента массива **A** большего, чем среднее арифметического этого массива.

Приложение Б Задание 2. Вычисления с плавающей точкой

Разработать программы на языках программирования C и Ассемблер, выполняющие вычисления над числами с плавающей точкой. Разработанные программы должны принимать числа в допустимом диапазоне. Например, нужно учитывать области определения и допустимых значений, если это связано с условием задачи.

1. Разработать программу, вычисляющую с помощью степенного ряда с точностью не хуже 0,05% значение функции $\sqrt{1+x}$ для заданного параметра x.

2. Разработать программу, вычисляющую с помощью степенного ряда с точностью не хуже 0,1% значение функции гиперболического синуса $\mathrm{sh}\,(x) = (e^x - e^{-x})/2$ для заданного параметра x.

3. Разработать программу, вычисляющую с помощью степенного ряда с точностью не хуже 0,1% значение функции $\cos(x)$ для заданного параметра x.

4. Разработать программу, вычисляющую с помощью степенного ряда с точностью не хуже 0,1% значение биномиальной функции $(1+x)^m$ для конкретных параметров m и x.

5. Разработать программу, вычисляющую с помощью степенного ряда с точностью не хуже 0,05% значение функции $\arcsin(x)$ для заданного параметра x.

6. Разработать программу, вычисляющую с помощью степенного ряда с точностью не хуже 0,05% значение функции $1/e^x$ для заданного параметра x.

7. Разработать программу, вычисляющую с помощью степенного ряда с точностью не хуже 0,05% значение функции $\sin(x)$ для заданного параметра x.

8. Разработать программу, вычисляющую с помощью степенного ряда с точностью не хуже 0,1% значение функции $\arccos(x)$ для заданного параметра x.

9. Разработать программу, вычисляющую с помощью степенного ряда с точностью не хуже 0,05% значение функции $\arctan(x)$ для заданного параметра x.

10. Разработать программу, вычисляющую с помощью степенного ряда с точностью не хуже 0,05% значение функции гиперболического тангенса $\tanh(x) = (e^x - e^{-x})/(e^x + e^{-x})$ для заданного параметра x.

11. Разработать программу, вычисляющую с помощью степенного ряда с точностью не хуже 0,05% значение функции *1/(1-x)* для заданного параметра x.

12. Разработать программу, вычисляющую с помощью степенного ряда с точностью не хуже 0,05% значение функции $\tan(x)$ для заданного параметра x.

13. Разработать программу, вычисляющую с помощью степенного ряда с точностью не хуже 0,1% значение функции e^x для заданного параметра x.

14. Разработать программу, вычисляющую с помощью степенного ряда с точностью не хуже 0,1% значение функции гиперболического котангенса $\operatorname{cth}(x) = (e^x + e^{-x})/(e^x - e^{-x})$ для заданного параметра x.

15. Разработать программу, вычисляющую с помощью степенного ряда с точностью не хуже 0,05% значение функции гиперболического косинуса $\operatorname{ch}(x) = (e^x + e^{-x})/2$ для заданного параметра x.

16. Разработать программу, вычисляющую с помощью степенного ряда с точностью не хуже 0,05% значение функции e^{-x} для заданного параметра x.

17. Разработать программу, вычисляющую с помощью степенного ряда с точностью не хуже 0,1% значение функции $\ln(1 - x)$ для входного параметра x.

18. Разработать программу вычисления корня квадратного по итерационной формуле Герона Александрийского с точностью не хуже 0,05%.

19. Разработать программу вычисления корня кубического из заданного числа согласно быстро сходящемуся итерационному алгоритму определения корня n-ной степени с точностью не хуже 0,05%.

20. Разработать программу вычисления числа π с точностью не хуже 0,05% посредством произведения элементов ряда Виета.

21. Разработать программу вычисления числа π с точностью не хуже 0,05% посредством ряда Нилаканта.

22. Разработать программу вычисления числа π с точностью не хуже 0,1% посредством дзета-функции Римана.

23. Разработать программу вычисления числа π с точностью не хуже 0,05% посредством произведения элементов ряда Валлиса.

24. Разработать программу, вычисляющую с помощью ряда Эйлера с точностью не хуже 0,1% значение числа e.

25. Разработать программу, решающую вопрос о принадлежности заданных 4-х точек одной окружности.

26. Разработать программу вычисления корня пятой степени согласно быстро сходящемуся итерационному алгоритму определения корня n-той степени с точностью не хуже 0,1%.

27. Разработать программу интегрирования функции $y = a+b*x^{-2}$ (задаётся двумя числами a,b) в заданном диапазоне (задаётся так же) методом Симпсона (точность вычислений = 0.0001).

28. Разработать программу численного интегрирования функции $y = a + b * x^4$ (задаётся действительными числами a,b) в определённом диапазоне целых (задаётся так же) методом прямоугольников с избытком (точность вычислений = 0.0001).

29. Разработать программу численного интегрирования функции $y = a + b * x^{-4}$ (задаётся действительными числами a,b) в определённом диапазоне целых (задаётся так же) методом средних (точность вычислений = 0.0001).

30. Разработать программу численного интегрирования функции $y = a + b * x^3$ (задаётся действительными числами a,b) в определённом диапазоне целых (задаётся так же) методом трапеций (точность вычислений = 0.0001).

31. Разработать программу численного интегрирования функции $y = a+b*x^3$ (задаётся действительными числами a,b) в определённом диапазоне целых (задаётся так же) методом прямоугольников с недостатком (точность вычислений = 0.0001).

32. Разработать программу, определяющую корень уравнения $2^{x^2+1} + x^2 - 4 = 0$ методом половинного деления с точностью от 0,001 до 0,00000001 в диапазоне [0;1]. Если диапазон некорректен, то подобрать корректный диапазон.

33. Разработать программу, определяющую корень уравнения $x^3 - 0.5x^2 + 0.2x - 4 = 0$ методом половинного деления с точностью от 0,001 до 0,00000001 в диапазоне [1;3]. Если диапазон некорректен, то подобрать корректный диапазон.

34. Разработать программу, определяющую корень уравнения $x^4 + 2x^3 - x - 1 = 0$ методом половинного деления с точностью = от 0,001 до 0,00000001 в диапазоне [0;1]. Если диапазон некорректен, то подобрать корректный диапазон.

35. Разработать программу, определяющую корень уравнения $x^4 - x^3 - 2.5 = 0$ методом хорд с точностью от 0,001 до 0,00000001 в диапазоне [1;2]. Если диапазон некорректен, то подобрать корректный диапазон.

36. Разработать программу, определяющую корень уравнения $2^{x^2+1} + x - 3 = 0$ методом хорд с точностью от 0,001 до 0,00000001 в диапазоне [2;3]. Если диапазон некорректен, то подобрать корректный диапазон.

37. Разработать программу, определяющую корень уравнения $x^5 - x - 0.2 = 0$ методом хорд с точностью от 0,001 до 0,00000001 в диапазоне [1;1.1]. Если диапазон некорректен, то подобрать корректный диапазон.

Приложение В Задание 3. Обработка строк символов

ASCII-строка — строка, содержащая символы таблицы кодировки ASCII. (`https://ru.wikipedia.org/wiki/ASCII`). Размер строки может быть достаточно большим, чтобы вмещать многостраничные тексты, например, главы из книг, если задача связана с использованием файлов или строк, порождаемых генератором случайных чисел. Тексты при этом могут не нести смыслового содержания. Для обработки в программе предлагается использовать данные, содержащие символы только из первой половины таблицы (коды в диапазоне $0–127_{10}$), что связано с использованием кодировки UTF-8 в ОС Linux в качестве основной. Символы, содержащие коды выше 127_{10}, должны отсутствовать во входных данных кроме оговоренных специально случаев.

1. Разработать программу, которая «переворачивает» заданную позициями $\mathbf{N}_1$–$\mathbf{N}_2$ часть ASCII–строки символов ($\mathbf{N}_1$, $\mathbf{N}_2$ вводятся как параметры).

2. Разработать программу, находящую в заданной ASCII–строке первую при обходе **от конца к началу** последовательность **N** символов, каждый элемент которой определяется по условию «больше предшествующего» (**N** вводится в качестве параметра).

3. Разработать программу, находящую в заданной ASCII-строке первую слева направо последовательность **N** символов, каждый элемент которой определяется по условию «меньше предшествующего» (***N*** *вводится как отдельный параметр*).

4. Разработать программу, находящую в заданной ASCII-строке последнюю при перемещении слева направо последовательность **N** символов, каждый элемент которой определяется по условию «больше предшествующего» (***N*** *вводится как отдельный параметр*).

5. Разработать программу, заменяющую все строчные гласные буквы в заданной ASCII-строке заглавными.

6. Разработать программу, заменяющую все согласные буквы в заданной ASCII-строке их **ASCII кодами в десятичной системе счисления**.

7. Разработать программу, заменяющую все гласные буквы в заданной ASCII-строке их **ASCII кодами в шестнадцатиричной системе счисления**. Код

каждого символа задавать в формате «**0xDD**», где **D** — шестнадцатиричная цифра от 0 до F.

8. Разработать программу, заменяющую все цифры (за исключением нуля) в заданной ASCII-строке **римскими цифрами**. То есть, соответствующим комбинациями букв для цифр от 1 до 9.

9. Разработать программу, которая «**переворачивает на месте**» заданную ASCII-строку символов (не копируя строку в другой буфер).

10. Разработать программу, которая меняет на обратный порядок следования символов **каждого слова** в ASCII-строке символов. Порядок слов остается неизменным. Слова состоят только из букв. Разделителями слов являются все прочие символы.

11. Разработать программу вычисления **отдельно количества гласных и согласных букв** в ASCII-строке.

12. Разработать программу, определяющую **минимальный и максимальный (по числовому значению) символ в заданной** ASCII-строке.

13. Разработать программу, заменяющую все строчные буквы в заданной ASCII-строке прописными, а прописные буквы — строчными.

14. Разработать программу, вычисляющую отдельно **число прописных и строчных букв** в заданной ASCII-строке.

15. Разработать программу, которая на основе заданной ASCII-строки символов, решает вопрос, является ли данная строка **палиндромом**.

16. Разработать программу, которая вычисляет **количество цифр и букв** в заданной ASCII-строке.

17. Разработать программу, которая **меняет на обратный порядок следование слов** в ASCII-строке символов.

18. Разработать программу, **заменяющую все согласные буквы в заданной ASCII-строке на заглавные**.

19. Разработать программу, вычисляющую число вхождений различных **отображаемых символов** в заданной ASCII-строке.

20. Разработать программу, вычисляющую **число вхождений различных цифр** в заданной ASCII-строке.

21. Разработать программу, осуществляющую **нахождение суммы всех цифр** в заданной ASCII-строке.

22. Разработать программу, вычисляющую **число вхождений различных знаков препинания** в заданной ASCII-строке.

23. Разработать программу, которая ищет в ASCII-строке заданную подстроку и возвращает **индекс первого символа первого вхождения подстроки в строке**. Подстрока вводится как параметр.

24. Разработать программу, которая ищет в ASCII-строке заданную подстроку и возвращает **список индексов первого символа для всех вхождений** подстроки в строке. Подстрока вводится как параметр.

25. Разработать программу, которая определяет в ASCII-строке **частоту встречаемости различных идентификаторов**, являющихся словами, состоящими из букв и цифр, начинающихся с буквы. Разделителями являются все другие символы. Для тестирования можно использовать программы, написанные на различных языках программирования.

26. Разработать программу, которая определяет **количество целых чисел** в ASCII-строке. числа состоят из цифр от 0 до 9. Разделителями являются все другие символы.

27. Разработать программу, которая определяет **частоту встречаемости (сколько раз встретилось в тексте) пяти ключевых слов языка программирования C**, в произвольной ASCII-строке. Ключевые слова не должны являться частью идентификаторов. Пять искомых ключевых слов выбрать **по своему усмотрению**. Тестировать можно на файлах программ.

28. Разработать программу, которая в заданной ASCII-строке определяет **корректность вложенности круглых скобок** «`(`» и «`)`». Необходимо учесть, что вложенные скобки могут образовывать в тексте различные группы. Например: `...(...)...(...)...`.

29. Разработать программу, которая ищет в ASCII-строке **слова — палиндромы и формирует из них новую строку**, в которой слова разделяются пробелами. Слова состоят из букв. Все остальные символы являются разделителями слов.

30. Разработать программу, которая ищет в ASCII-строке слова, **начинающиеся с заглавной буквы и формирует из них новую строку**, в которой слова разделяются пробелами. Слова состоят из букв. Все остальные символы являются разделителями слов.

31. Разработать программу, которая ищет в ASCII-строке **целые числа и формирует из них новую строку**, в которой числа разделяются знаком «`+`». Слова состоят из букв. Все остальные символы являются разделителями слов.

32. Разработать программу, которая на основе анализа двух входных ASCII-строк формирует на выходе две другие строки. В первой из выводимых строк содержатся символы, которых нет во второй исходной строке. Во второй выводимой строке содержатся символы, отсутствующие в первой входной строке (**разности символов**). Каждый символ в соответствующей выходной строке должен встречаться только один раз.

33. Разработать программу, которая на основе анализа двух ASCII-строк формирует на выходе строку, содержащую символы, присутствующие в обеих строках (**пересечение символов**). Каждый символ в соответствующей выходной строке должен встречаться только один раз.

34. Разработать программу, которая на основе анализа двух ASCII-строк формирует на выходе строку, содержащую символы, присутствующие в одной или другой (**объединение символов**). Каждый символ в соответствующей выходной строке должен встречаться только один раз

Приложение Г Задание 4. Обработка строк символов

ASCII-строка — строка, содержащая символы таблицы кодировки ASCII. (`https://ru.wikipedia.org/wiki/ASCII`). Размер строки может быть достаточно большим, чтобы вмещать многостраничные тексты, например, главы из книг, если задача связана с использованием файлов или строк, порождаемых генератором случайных чисел. Тексты при этом могут не нести смыслового содержания. Для обработки в программе предлагается использовать данные, содержащие символы только из первой половины таблицы (коды в диапазоне 0–127_{10}), что связано с использованием кодировки UTF-8 в ОС Linux в качестве основной. Символы, содержащие коды выше 127_{10}, должны отсутствовать во входных данных кроме оговоренных специально случаев.

1. Разработать программу, которая «переворачивает» заданную позициями $\mathbf{N}_1$–$\mathbf{N}_2$ часть ASCII–строки символов ($\mathbf{N}_1$, $\mathbf{N}_2$ вводятся как параметры).

2. Разработать программу, находящую в заданной ASCII–строке первую при обходе **от конца к началу** последовательность **N** символов, каждый элемент которой определяется по условию «больше предшествующего» (**N** вводится в качестве параметра).

3. Разработать программу, находящую в заданной ASCII-строке первую слева направо последовательность **N** символов, каждый элемент которой определяется по условию «меньше предшествующего» (***N*** *вводится как отдельный параметр*).

4. Разработать программу, находящую в заданной ASCII-строке последнюю при перемещении слева направо последовательность **N** символов, каждый элемент которой определяется по условию «больше предшествующего» (***N*** *вводится как отдельный параметр*).

5. Разработать программу, заменяющую все строчные гласные буквы в заданной ASCII-строке заглавными.

6. Разработать программу, заменяющую все согласные буквы в заданной ASCII-строке их **ASCII кодами в десятичной системе счисления**.

7. Разработать программу, заменяющую все гласные буквы в заданной ASCII-строке их **ASCII кодами в шестнадцатиричной системе счисления**. Код

каждого символа задавать в формате «**0xDD**», где **D** — шестнадцатиричная цифра от 0 до F.

8. Разработать программу, заменяющую все цифры (за исключением нуля) в заданной ASCII-строке **римскими цифрами**. То есть, соответствующим комбинациями букв для цифр от 1 до 9.

9. Разработать программу, которая «**переворачивает на месте**» заданную ASCII-строку символов (не копируя строку в другой буфер).

10. Разработать программу, которая меняет на обратный порядок следования символов **каждого слова** в ASCII-строке символов. Порядок слов остается неизменным. Слова состоят только из букв. Разделителями слов являются все прочие символы.

11. Разработать программу вычисления **отдельно количества гласных и согласных букв** в ASCII-строке.

12. Разработать программу, определяющую **минимальный и максимальный (по числовому значению) символ в заданной** ASCII-строке.

13. Разработать программу, заменяющую все строчные буквы в заданной ASCII-строке прописными, а прописные буквы — строчными.

14. Разработать программу, вычисляющую отдельно **число прописных и строчных букв** в заданной ASCII-строке.

15. Разработать программу, которая на основе заданной ASCII-строки символов, решает вопрос, является ли данная строка **палиндромом**.

16. Разработать программу, которая вычисляет **количество цифр и букв** в заданной ASCII-строке.

17. Разработать программу, которая **меняет на обратный порядок следование слов** в ASCII-строке символов.

18. Разработать программу, **заменяющую все согласные буквы в заданной ASCII-строке на заглавные**.

19. Разработать программу, вычисляющую число вхождений различных **отображаемых символов** в заданной ASCII-строке.

20. Разработать программу, вычисляющую **число вхождений различных цифр** в заданной ASCII-строке.

21. Разработать программу, осуществляющую **нахождение суммы всех цифр** в заданной ASCII-строке.

22. Разработать программу, вычисляющую **число вхождений различных знаков препинания** в заданной ASCII-строке.

23. Разработать программу, которая ищет в ASCII-строке заданную подстроку и возвращает **индекс первого символа первого вхождения подстроки в строке**. Подстрока вводится как параметр.

24. Разработать программу, которая ищет в ASCII-строке заданную подстроку и возвращает **список индексов первого символа для всех вхождений** подстроки в строке. Подстрока вводится как параметр.

25. Разработать программу, которая определяет в ASCII-строке **частоту встречаемости различных идентификаторов**, являющихся словами, состоящими из букв и цифр, начинающихся с буквы. Разделителями являются все другие символы. Для тестирования можно использовать программы, написанные на различных языках программирования.

26. Разработать программу, которая определяет **количество целых чисел** в ASCII-строке. числа состоят из цифр от 0 до 9. Разделителями являются все другие символы.

27. Разработать программу, которая определяет **частоту встречаемости (сколько раз встретилось в тексте) пяти ключевых слов языка программирования C**, в произвольной ASCII-строке. Ключевые слова не должны являться частью идентификаторов. Пять искомых ключевых слов выбрать **по своему усмотрению**. Тестировать можно на файлах программ.

28. Разработать программу, которая в заданной ASCII-строке определяет **корректность вложенности круглых скобок** «`(`» и «`)`». Необходимо учесть, что вложенные скобки могут образовывать в тексте различные группы. Например: `...(...)...(...)....`

29. Разработать программу, которая ищет в ASCII-строке **слова — палиндромы и формирует из них новую строку**, в которой слова разделяются пробелами. Слова состоят из букв. Все остальные символы являются разделителями слов.

30. Разработать программу, которая ищет в ASCII-строке слова, **начинающиеся с заглавной буквы и формирует из них новую строку**, в которой слова разделяются пробелами. Слова состоят из букв. Все остальные символы являются разделителями слов.

31. Разработать программу, которая ищет в ASCII-строке **целые числа и формирует из них новую строку**, в которой числа разделяются знаком «`+`». Слова состоят из букв. Все остальные символы являются разделителями слов.

32. Разработать программу, которая на основе анализа двух входных ASCII-строк формирует на выходе две другие строки. В первой из выводимых строк содержатся символы, которых нет во второй исходной строке. Во второй выводимой строке содержатся символы, отсутствующие в первой входной строке (**разности символов**). Каждый символ в соответствующей выходной строке должен встречаться только один раз.

33. Разработать программу, которая на основе анализа двух ASCII-строк формирует на выходе строку, содержащую символы, присутствующие в обеих строках (**пересечение символов**). Каждый символ в соответствующей выходной строке должен встречаться только один раз.

34. Разработать программу, которая на основе анализа двух ASCII-строк формирует на выходе строку, содержащую символы, присутствующие в одной или другой (**объединение символов**). Каждый символ в соответствующей выходной строке должен встречаться только один раз

Литература

[1] RISC-V International. Описание архитектуры и ее обоснование. — https://riscv.org/

[2] Сара Л. Харрис, Дэвид Харрис. Цифровая схемотехника и архитектура компьютера: RISC-V / пер. с англ. В. С. Яценкова, А. Ю. Романова; под ред. А. Ю. Романова. — М.: ДМК Пресс, 2021. — 810 с.

[3] Edson Borin An Introduction to Assembly Programming with RISC-V / Document version: May 9, 2022 — https://riscv-programming.org/

[4] Курячий Георгий. Архитектура и язык ассемблера RISC-V. Весна 2022. — http://uneex.ru/LecturesCMC/ArchitectureAssembler2022

[5] Семестровый забег "Архитектур процессорных систем— https://github.com/MPSU/APS

[6] [UNИХ] Архитектура и язык ассемблера RISC-V. Видео лекции. Весна 2022. — https://www.youtube.com/playlist?list=PL6kSdcHYB3x6cjkby4H1RuRMzfbEGSNBi

[7] Архитектуры процессорных систем — https://www.youtube.com/c/%D0%90%D0%9F%D0%A1%D0%9F%D0%BE%D0%BF%D0%BE%D0%B2

[8] RARS – RISC-V Assembler and Runtime Simulator — https://github.com/TheThirdOne/rars

[9] Ripes. A visual computer architecture simulator and assembly code editor. — https://github.com/mortbopet/Ripes

[10] QtRvSim–RISC-V CPU simulator for education — https://github.com/cvut/qtrvsim

[11] Goossens Bernard. Guide to Computer Processor Architecture. A RISC-V Approach, with High-Level Synthesis. — Springer Nature. Switzerland AG — 2023.

[12] Буч Г. Объектно-ориентированное проектирование с примерами применения. /Пер. с англ. — М.: Конкорд, 1992. — 519 с.

[13] Gay Warren. RISC-V Assembly Language Programming. Using ESP32-C3 and QEMU. — Elektor International Media B.V. — 2022.

[14] Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е изд./Пер. с англ. — М.: «Издательства Бином», СПб: «Невский диалект», 1998 г. — 560 с., ил.

[15] Англо-русско-немецко-французский толковый словарь по вычислительной технике и обработке данных, 4132 термина. Под. ред. А.А. Дородницына. М.: 1978. — 416 с.

[16] Гагарина Л.Г., Кононова А.И. Архитектура вычислительных систем и Ассемблер с приложением методических указаний к лабораторным работам. Учебное пособие. — М.: СОЛОН-Пресс, 2019. — 368 с.

[17] Формат файла ELF64. — `https://uclibc.org/docs/elf-64-gen.pdf`

[18] Plantz Robert G. Introduction to Computer Organization. — 2022

[19] Ричард Столмен, Роланд Пеш, Стан Шебс и др. Отладка с помощью GDB. — 2000

[20] Андреас Целлер Почему не работают программы. — М.: Эксмо, 2011. — 560 с.

[21] Suzanne J. Matthews, Tia Newhall, Kevin C. Webb. Dive into Systems. — 2022

[22] Округление. Статья в Википедии. — `https://ru.wikipedia.org/wiki/%D0%9E%D0%BA%D1%80%D1%83%D0%B3%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5`

[23] Прохоренок Н.А. Язык C. Самое необходимое. — СПб.: БХВ-Петербург, 2020. — 480 с.

[24] Йо Ван Гуй. Программирование на ассемблере x64: от начального уровня до профессионального использования AVX. — М.: ДМК Пресс, 2021. — 332 с.

[25] Установка Linux на Windows с помощью WSL — `https://docs.microsoft.com/ru-ru/windows/wsl/install`

[26] Установка Linux на Virtualbox. — `https://losst.ru/ustanovka-linux-na-virtualbox`

[27] System V Application Binary Interface AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models). Version 1.0. — 2018. — `https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-1.0.pdf`