Архитектура вычислительных систем

Методические указания к самостоятельной работе студентов

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

31 августа 2023 г.

Оглавление

Введение

В методических указаниях представлен материал, предназначенный для самостоядельного изучения ряда вопросов по дисциплине «Архитектура вычислительных систем». Данный материал является вспомогательным для освоения тем, рассматриваемых на семинарских занятиях. Он также полезен при выполнении домашних заданий, выдаваемых для закрепления знаний и навыков полученных на семинарах, а также при выполнении индивидуальных заданий.

Цель и задачи

Целью дисциплины является рассмотрение особенностей архитектур вычислительных систем различного уровня организации и более получение более детального представления об архитектуре уровня системы команд и поддерживающей его микроархитектуре.

Для достижения поставленной цели предполагается рассмотреть следующие основные разделы:

- Общее понятие архитектуры ВС. Классификация архитектур ВС
- Особенности архитектуры ВС уровня системы команд.
- Отображение архитектуры ВС уровня системы команд в языках системного программирования и ассемблере.
- Уровень операционной системы и его использование в низкоуровневом программировании.
- Разнообразие архитектур уровня системы команд.
- Поддержка архитектуры уровня системы команд на уровне микроархитектур.
- Отображение параллелизма в архитектурах ВС.

Основные темы, затрагиваемые при изучении дисциплины

При всей многоуровневости и разнообразии архитектур ВС рамках изучаемой дисциплины основной упор предполагается сделать на уровень системы команд и

параллелизм. Это обуславливается тем, что в объеме одного семестра можно либо глубоко рассмотреть ограниченный набор архитектурных решений, либо поверхностно пробежаться по более широкому их числу, избегая конкретики. Я предпочитаю первый вариант. Исходя из этого самым верхним будет являться архитектурный уровень языков системного программирования, который рассматривается как основополагающая связь между более верхними уровнями прикладных языков и всем тем, что находится ниже.

В качестве языка программирования, отражающего системный уровень выбран язык C, который практически однозначно отображает в своих конструкциях низкоуровневые решения. Он позволяет рассмотреть использование библиотек, определяющих архитектуры уровня операционной системы (ОС), а также продемонстрировать непосредственную связь с языками ассемблера для различных архитектур уровня системы команд. Помимо этого использование данного языка позволяет применить его библиотеку функций при программировании на языке ассемблера, обеспечивая более высокоуровневый ввод-вывод данных, работу с файлами, а также ряд других манипуляций по сравнению с использованием аналогичных системных вызовов уровня операционной системы.

Рассмотрение уровня операционной системы позволяет рассмотреть реализацию параллелизма, что в настоящее время является неотъемлимой практикой в программировании. Помимо системных вызовов предполагается также рассмотреть более высокоуровневую поддержку параллельных вычислений для различных архитектур, которая опирается на системные вызовы ОС.

Практические занятия и задания

В рамках практических занятий основной упор делается на выполнение заданий, закрепляющих знания низкоуровневых архитектур, таких как уровень ОС, ассемблера, системы команд. Для написания программы при этом используется язык ассемблера. Помимо этого имеются задания связанные с изучением и практическим использованием многопоточных архитектур на уровне ОС и библиотеки Posix threads. Используемое при этом многопоточное программирования осуществляется на языке программирования C.

При выполнении заданий предполагается использование следующих инструментальных средств и систем программирования:

- вычислительную систему (ПК, ноутбук) с архитектурой x86-64 (AMD-64);
- операционную систему Linux;
- языки программирования C (gcc, clang), GNU ассемблер (as);
- библиотеки уровня ОС и языка C (stdio.h, stdlib.h, string.h и т.д.)

Выбор архитектуры машинного уровня x86-64 обуславливается ее массовым распространением, что позволяет не использовать различные эмуляторы и упрощает непосредственное взаимодействие с компьютером.

Свободно распространяемой ОС Linux вполне достаточно для решения заданий. Помимо возможной непосредственной установки ее можно легко запускать в различных эмулирующих средах. В частности, под ОС Windows можно использовать Windows Subsystem for Linux (WSL). На любой платформе можно также запускать Linux под виртуальной машиной, например, VirtualBox. Описания вариантов установки доступны в сети Интернет. Практически можно использовать любой дистрибутив. При этом достаточно консольной версии, так как в результате выполнения заданий должны создаваться только консольные приложения.

Компиляторы языков программирования C имеются практически в любом дистрибутиве Linux. Проще при этом ориентироваться на семейство Gnu Compiler Collection (GCC). Однако можно использовать и clang.

Среди всего разнообразия ассемблеров, которые используются в ОС Linux, предлагается ориентироваться на тот из них, который по сути является основным инструментом в данной операционной системе. Это GNU ассемблер (GAS). В этом случае программы, разрабатываемые непосредственно на ассембле, можно легко сопоставлять с программами, написанными на C и откомпилированными в ассемблер, что позволяет быстрее изучать и отыскивать необходимые машинные команды по аналогии. Также следует отметить, что язык C позволяет использовать ассемблерные вставки, написанные на GAS, что также облегчает изучение архитектур уровней ассемблера и системы команд. Помимо этого основной отладчик в Linux, Gnu Debugger (gdb), так же поддерживает мнемонику данного ассемблера. При этом выбор мнемоники (intel или AT&T(?)). Можно осуществлять по собственному усмотрению.

Состав библиотек определяется тем, что они поддерживаются практически всеми компиляторами C, обеспечивая также работу с языками ассемблера.

Предлагаемые задания достаточно простые и не требуют для их написания интегрированных средств разработки. Достаточно текстовых редакторов. В качестве дополнительных инструментов могут опционально пригодиться средства сборки проектов cmake и make. Для сохранения результатов работы и обеспечения их проверки необходимо пользоваться одной из систем контроля версий в сети Интернет (предлагается использовать github).

Краткое содержание курса

> ***Примечание***
> Пока оставил текст из предыдущего описания. Нужно продумать и разделить, что пойдет в ЛМС, а что на сайт. Предполагается, что на сайте будет выложен более подробный материал. Болшее число тем.

Основной упор я предполагаю сделать на традиционных архитектурах в разрезе их многоуровневости. То есть, пройтись по вертикали от логических схем и архитектуре на уровне этих схем, рассмотрев построение систем на кристалле. На верхнем уровне, скорее всего, будут специализированные архитектуры и архитектуры параллельных вычислительных систем. С каждым из уровней предполагается

увязать свой язык программирования, рассмотреть основной набор команд и обобщенную структуру виртуальной машины данного уровня.

Изложение скорее всего не будет упорядочено по уровням (сверху вниз или снизу вверх). Я планирую отталкиваться от известных архитектур универсальных языков высокого уровня, опускаясь в начале до командного уровня (до языков ассемблера). На уровне системы команд предполагаю разобрать различные варианты современных архитектур.

Исходя из этого планируется следующая последовательность подачи материала в лекционном курсе.

Архитектура процедурной императивной машины

В рамках данной темы планируется затронуть упрощенную организацию высокоуровневой императивной машины, использующей статическую типизацию данных. Начать стоит с «Фортран-машины», то есть, с нулевого метауровня, не содержащего абстракция более высоко уровня. Показать отличия систем типов на основе методов задания однозначности. Рассмотреть особенности структур различного вида: бестиповой, статически типизированной, динамически типизированной. Увязать их с различными видами архитектур. Рассмотреть соответствующие примеры на языках C++ (в стиле C), Python (или JS). пояснить, почему динамическая типизация чаще используется в интерпретируемых языках.

Может быть при анализе типов также стоит отметить, что между статической и функциональной однозначностью существуе однозначный переход.

После этого можно перейти к первому метауровню, на котором рассматриваются абстрактные типы данных. В этот же метауровень можно внести формальные параметры функций (процедур). Рассмотреть структурную организацию и обобщенную архитектурную поддержку. Показать, как данный метауровень реализуется в статически типизированных процедурныъ языках на агрегативных данных (структурах), пояснив его суть. Пояснить, почему он не используется особо в языках с динамической типизацией.

После этого можно перейти к использованию АТД для описания альтернативных типов данных. Пояснить, что в любой мало-мальски приличной программе всегда необходима проверка типов во время выполняения. Продемонстрировать на примере. Пояснить растипизацию (разыменование типов). Показать наличие растипизации и необходимость дополнительных механизмов проверки типов. Рассказать о бестиповом подходе с явным заданием признака. Также рассказать о более надежном решении, которое используетс в языке программирования Ада.

Помимо этого можно сопоставить императивный и функциональный подход. Показать, в чем заключается основная специфика функционального программирования и его отличия от процедурного. Разобрать архитектуры функциональных машин. Показать, что в них также может использоваться различная система типов.

В качестве задания и тем для обсуждения на практических занятиях планируется предложить разработку программы для процедурной машины. Может быть для разнообразия в качестве альтернативных решений стоит добавить использование

различных систем типов и уровней абстракции (м.б. и языков программирования). Хотя, скорее всего основная идея - это использование статической типизации и абстрактных типов данных при процедурном подходе.

Архитектура объектно-ориентированной машины

Архитектура уровня системы команд

Архитектура уровня микрокоманд

Архитектура логического уровня

Архитектуры параллельных вычислительных систем

Зачем читать эту книгу?

Учитывая, что существует множество отличных языков высокого уровня, которые позволяют вам писать программы, не заботясь о том, как машинные инструкции управляют оборудованием, вы можете задаться вопросом, зачем вам изучать материал этой книги. Все языки высокого уровня в конечном счете переводятся в машинные инструкции, управляющие оборудованием. ***Понимание того, что делает аппаратное обеспечение и как инструкции управляют им, поможет понять возможности и ограничения компьютера. Я считаю, что это понимание может сделать вас лучшим программистом, даже если вы работаете с языком высокого уровня.***

Если вас в первую очередь интересует аппаратное обеспечение, я думаю, важно понимать, как аппаратное обеспечение будет использоваться программой.

Вам может понравиться программирование на ассемблере, и вы захотите продолжить. Например, если ваши интересы приводят вас к системному программированию — написанию частей операционной системы, написанию компилятора или даже разработке другого языка более высокого уровня — эти усилия обычно требуют понимания на уровне языка ассемблера.

Много сложных возможностей также существует в программировании встроенных систем, систем, в которых компьютер выполняет специальную задачу. Примеры являются неотъемлемой частью нашей повседневной жизни: сотовые телефоны; бытовая техника; автомобили; системы отопления, вентиляции и кондиционирования воздуха (HVAC); медицинское оборудование; и так далее. Встроенные системы являются важным компонентом технологии Интернета вещей (IoT). Их программирование часто требует понимания того, как компьютер взаимодействует с различными аппаратными устройствами на уровне языка ассемблера.

1 Эмулятор RARS процессора RISC-V

RARS — это интегрированная среда разработки, отладки и выполнения программ процессора RISC-V в режиме симуляции. Она написана на языке программирования Java, что обеспечивает переносимость между компьютерами с различной архитектурой, использующих разнообразные операционные системы. Распространяется в виде исполняемого файла JAR. Среда включает:

- текстовый редактор, обеспечивающий написание программ;
- ассемблер, позволяющий получить исполняемый код путем сборки одного или нескольких модулей;
- эмулятор, осуществляющий моделирование процессора RISC-V и позволяющий выполнять программы, разработанные на языке ассемблера;
- отладчик, работающий как в пошаговом режиме, так и использующий точки останова;
- средства, обеспечивающие отображение информации о состоянии регистров моделируемого процессора и его оперативной памяти;
- модели внешних устройств, позволяющие имитировать взаимодействие с внешней средой в различных режимах.

Помимо этого компоненты, входящие в среду разработки можно запускать из командной строки.

> **Примечание**: Остальне будет дописываться по мере развития методы...

1.1 Использование RARS через интегрированную среду разработки (IDE)

IDE вызывается, когда RARS запускается без аргументов команды, например: `java -jar rars.jar`. Его также можно запустить из графического интерфейса, дважды щелкнув значок `rars.jar`, представляющий этот исполняемый файл JAR. IDE предоставляет интуитивно понятные базовые возможности редактирования, сборки и выполнения. Ниже представлены некоторые доступные функции:

- **Меню и панель инструментов.** Большинство элементов меню имеют одинаковые значки на панели инструментов. Если функция значка на панели инструментов не очевидна, просто наведите на него указатель мыши, и вскоре появится всплывающая подсказка. Почти все пункты меню также имеют сочетания клавиш. Любой пункт меню, не подходящий в данной ситуации, отключается.

- **Редактор.** RARS включает два встроенных текстовых редактора. Редактор по умолчанию имеет цветовую подсветку большинства языковых элементов с учетом синтаксиса и всплывающие инструкции. Помимо этого имеется ранее разработанный универсальный текстовый редактор без этих функций. Его можно выбрать в диалоговом окне «Настройки редактора». Он поддерживает один шрифт, который можно изменить в диалоговом окне настроек редактора. Нижняя граница каждого из этих редакторов включает строку курсора и положение столбца. Также отображаются номера строк. *Можно использовать внешний редактор. RARS предоставляет удобную настройку, которая будет автоматически собирать файл, как только он будет открыт. См. меню «Настройки».*

- **Области сообщений.** В нижней части экрана есть две области сообщений с вкладками. Вкладка «**Run I/O**» используется во время выполнения для отображения ввода–вывода консоли по мере выполнения программы. Также имеется возможность осуществлять консольный ввод во всплывающем диалоговом окне, а затем отображать его в области сообщений. Вкладка с названием «**Сообщения RARS**» используется для других сообщений, таких как ошибки сборки или выполнения, а также для информационных сообщений. Вы можете нажать на сообщения об ошибках сборки, чтобы выбрать соответствующую строку кода в редакторе.

- **Регистры.** Регистры отображаются постоянно, даже тогда, когда программа редактируется, а не запускается. При написании программы это служит полезным справочником по именам регистров и их обычному использованию (можно навести указатель мыши на имя регистра, чтобы увидеть всплывающие подсказки). Имеется три вкладки регистров:
  - файл регистров, содержащий целочисленные регистры от `x0` до `x31`, регистры `LO` и `HI`, а также счетчик программ (`pc`);
  - *выбранные регистры сопроцессора 0 (исключения и прерывания)*;
  - регистры с плавающей запятой сопроцессора 1.

- **Сборка.** Выберите «Собрать» в меню «Выполнить» или соответствующий значок на панели инструментов, чтобы собрать файл, который в данный момент находится на вкладке «Правка». Файлы в текущем каталоге. Если параметр «Собрать все» включен, ассемблер соберет текущий файл как «основную» программу, а также соберет все другие файлы сборки (`*.asm`; `*.s`) в том

же каталоге. А с опцией «Assemble Open», открытый в данный момент ассемблер также соберет открытые в данный момент файлы. Результаты связаны, и если все эти операции были успешными, программа может быть выполнена. На метки, объявленные глобальными с помощью директивы `.globl`, можно ссылаться в любом другом файле проекта. Существует также параметр, разрешающий автоматическую загрузку и сборку выбранного файла обработчика исключений.

- **Выполнение.** После успешной сборки программы инициализируются регистры и заполняются три окна на вкладке «Выполнение»: текстовый сегмент, сегмент данных и метки программы. Основные функции времени выполнения описаны ниже.

- **Окно меток.** Отображение окна меток (таблица символов) управляется через меню настроек. При отображении вы можете щелкнуть любую метку или связанный с ней адрес, чтобы отцентрировать и выделить содержимое этого адреса в окне «Текстовый сегмент» или в окне «Сегмент данных» в зависимости от ситуации.

Ассемблер и симулятор вызываются из IDE, когда вы выбираете операции Assemble, Go или Step в меню Run или соответствующие им значки на панели инструментов или сочетания клавиш. Сообщения RARS отображаются на вкладке Сообщения RARS области сообщений в нижней части экрана. Ввод и вывод консоли среды выполнения обрабатывается на вкладке Run I/O.

1.2 Интерактивные функции отладки

RARS предоставляет множество функций для интерактивной отладки на панели «Выполнение»:

- В пошаговом режиме выделяется следующая команда для выполнения, а содержимое памяти обновляется на каждом шаге.

- Для непрерывного выполнения выбирает опция «Go». Ее также можно использовать для продолжения выполения из состояния паузы (шаг, точка останова, пауза).

- Точки останова легко устанавливаются и сбрасываются с помощью флажков рядом с каждой инструкцией, отображаемой в окне «Текстовый сегмент». Можно временно приостановить точки останова, используя опцию «Toggle Breakpoints» в меню «Выполнить» или щелкнув заголовок столбца «Bkpt» в окне «Текстовый сегмент». Точки останова можно снова активировать повторным выбором.

- При работе в режиме «Go» вы можете выбрать скорость симуляции с помощью ползунка «Run Speed». Доступные скорости варьируются от 0,05 инструкций в секунду (20 секунд между шагами) до 30 инструкций в секунду, а выше этого предлагается «неограниченная» скорость. При использовании «неограниченной» скорости подсветка кода и обновление отображения памяти отключаются во время моделирования (но выполняется очень быстро!). При достижении точки останова происходит выделение и обновление. Скорость работы можно регулировать во время работы программы.

- При работе в режиме «Go» вы можете в любой момент приостановить или остановить симуляцию, используя функции «Pause» или «Stop». Первый приостановит выполнение и обновит отображение, как если бы вы выполняли пошаговое выполнение или достигли точки останова. Последний завершит выполнение и отобразит окончательные значения памяти и регистров. При работе на «неограниченной» скорости система может не реагировать немедленно, но ответит.

- У вас есть возможность интерактивно шагать «назад» через выполнение программы по одной инструкции за раз, чтобы «отменить» шаги выполнения. Он будет буферизовать до 2000 самых последних шагов выполнения (это ограничение хранится в файле свойств и может быть изменено). Он отменит изменения, внесенные в память, регистры или CSR (однако в настоящее время состояние прерывания не сохраняется), но не консольный или файловый ввод-вывод. Это должно быть отличным средством отладки. Он доступен в любое время приостановки выполнения и при завершении (даже если оно было прекращено из-за исключения).

- Когда выполнение программы приостановлено или остановлено, выберите «Сброс», чтобы сбросить все ячейки памяти и регистры до их исходных значений после сборки. Фактически «Reset» реализуется пересборкой программы.

- Адреса и значения памяти, а также значения регистров можно просматривать как в десятичном, так и в шестнадцатеричном формате. Все данные хранятся в порядке байтов с прямым порядком байтов (каждое слово состоит из байта 3, за которым следует байт 2, затем 1, затем 0). Обратите внимание, что каждое слово может содержать 4 символа строки, и эти 4 символа будут отображаться в порядке, обратном порядку строкового литерала.

- Содержимое сегмента данных отображается по 512 байт за раз (с прокруткой), начиная с базового адреса сегмента данных (0x10010000). Кнопки навигации предназначены для перехода к следующему разделу памяти, предыдущему или возврату к исходному (домашнему) диапазону. Поле со списком также предоставляется для просмотра содержимого памяти рядом с указателем стека (содержимое регистра sp), глобальным указателем (содержимое регистра gp), базовым адресом кучи (0x10040000), глобальными переменными .extern

(0x10000000) или отображением на память устройств ввода–выводв (MMIO, 0xFFFF0000). Содержимое необработанного текстового сегмента также может быть отображено.

- Содержимое любого слова памяти сегмента данных и почти любого регистра можно изменить, отредактировав отображаемую ячейку таблицы. Дважды щелкните ячейку, чтобы отредактировать ее, и нажмите клавишу Enter, когда закончите вводить новое значение. Если вы введете недопустимое 32-разрядное целое число, в ячейке появится слово INVALID, и содержимое памяти/регистра не изменится. Значения можно вводить как в десятичном, так и в шестнадцатеричном формате (начальный "0x"). Отрицательные шестнадцатеричные значения можно вводить либо в формате дополнения до двух, либо в формате со знаком. Обратите внимание, что три целочисленных регистра (ноль, программный счетчик, адрес возврата) не могут быть отредактированы.

- Если включен параметр «Самоизменяющийся код» (отключен по умолчанию, смотрите в меню «Настройки»), двоичный код текстового сегмента можно изменить, используя тот же метод, описанный выше. Его также можно изменить, дважды щелкнув ячейку в столбце «Код» дисплея «Текстовый сегмент».

- Содержимое ячеек, представляющих регистры с плавающей запятой, можно редактировать, как описано выше, и оно будет принимать допустимые шестнадцатеричные или десятичные значения с плавающей запятой. Поскольку каждый регистр двойной точности перекрывает два регистра одинарной точности, любые изменения в регистре двойной точности повлияют на одно или оба отображаемых содержимого соответствующих регистров одинарной точности. Изменения в регистре одинарной точности повлияют на отображение соответствующего ему регистра двойной точности. Значения, введенные в шестнадцатеричном формате, должны соответствовать формату IEEE-754. Значения, введенные в десятичном формате, вводятся с использованием десятичной точки и E-нотации (например, 12,5e3 — это 12,5 умножить на 10 в кубе).

- Содержимое ячейки можно редактировать во время выполнения программы, и после принятия оно будет применяться, начиная со следующей выполняемой инструкции.

- Нажатие на элемент окна «Ярлыки» приведет к тому, что местоположение, связанное с этим ярлыком, будет центрировано и выделено в окне «Текстовый сегмент» или «Сегмент данных» в зависимости от ситуации. Обратите внимание, что окно «Ярлыки» не отображается по умолчанию, но его можно открыть, выбрав его в меню «Настройки».

1.3 Дополнительные возможности: подключаемые инструменты

RARS может запускать стороннее программное обеспечение, которое взаимодействует с исполняемой программой и системными ресурсами. Требования к такой программе:

1. Он реализует интерфейс rars.tools.Tool.
2. Это часть пакета rars.tools.
3. Он аккуратно компилируется в файл «.class», хранящийся в каталоге rars/tools.

RARS обнаружит все подходящие инструменты при запуске и включит их в свое меню «Инструменты». Когда выбран пункт меню инструмента, его экземпляр будет создан с использованием его конструктора без аргументов, и будет вызван его метод action(). Если при запуске не найдено подходящих инструментов, меню «Инструменты» не появится.

Чтобы использовать такой инструмент, загрузите и соберите интересующую вас программу, затем выберите нужный инструмент в меню «Инструменты». Откроется окно инструмента, и в зависимости от того, как он написан, его либо нужно будет «подключить» к программе, нажав кнопку, либо он уже будет подключен. Запустите программу, как обычно, чтобы инициировать взаимодействие инструмента с исполняемой программой.

Абстрактный класс rars.tools.AbstractToolAndApplication включен в дистрибутив RARS, чтобы обеспечить существенную основу для реализации вашего собственного Инструмента. Подкласс, который расширяет его, реализуя как минимум два его абстрактных метода, может быть запущен не только из меню «Инструменты», но и как отдельное приложение, использующее ассемблер и симулятор RARS в фоновом режиме.

Несколько инструментов, разработанных на основе подкласса AbstractMarsToolAndApplication, включены в RARS: введение в инструменты, симулятор кэша данных, визуализатор ссылок на память и инструмент с плавающей запятой. Последний весьма полезен, даже если он не подключен к программе, потому что он отображает двоичные, шестнадцатеричные и десятичные представления для 32-битного значения с плавающей запятой; когда любой из них изменяется, два других также обновляются.

1.3.1 Добавление дополнительных системных вызовов

Системные вызовы (инструкция ecall) реализованы с использованием метода, аналогичного инструментальному. Это позволяет любому добавить новый системный вызов, определив новый класс, отвечающий следующим требованиям:

1. Он расширяет класс rars.riscv.AbstractSyscall.
2. Это часть пакета rars.riscv.syscalls.

3. Он аккуратно компилируется в файл ".class хранящийся в каталоге rars/riscv/syscalls.

4. Запись добавлена в Syscall.properties.

1.3.2 Расширение набора инструкций

Вы можете добавить настраиваемые псевдоинструкции в набор инструкций, отредактировав файл `PseudoOps.txt`. Форматы спецификаций инструкций объясняются в самом файле. Спецификация псевдоинструкции занимает одну строку. Он состоит из примера инструкции, созданной с использованием доступных символов спецификации инструкции, за которым следует список основных инструкций, разделенных точкой с запятой, до которых она будет расширена. Каждый из них представляет собой шаблон инструкции, построенный с использованием символов спецификации инструкции в сочетании со специальными символами спецификации шаблона. Последние допускают подстановку во время сборки программы операндов из пользовательской программы в расширенную псевдоинструкцию.

PseudoOps.txt считывается и обрабатывается при запуске, и если спецификация имеет неправильный формат, будут выдаваться сообщения об ошибках. Обратите внимание, что если вы хотите отредактировать его, вам сначала нужно извлечь его из файла JAR.

1.4 Использование RARS из командной строки

RARS можно запускать из интерпретатора командной строки для сборки и выполнения программы в пакетном режиме. Формат запуска:

```
java -jar rars.jar [options] program.asm [more files...] \
                   [ pa arg1 [more args...]]
```

Элементы в `[ ]` являются необязательными. Допустимые параметры (без учета регистра, разделенные пробелами):

Таблица 1.1 описывает используемые опции.

Если указано более одного имени файла, первое считается основным, если только в одном из файлов не определена метка глобального оператора main. Обработчик исключений не собирается автоматически. Добавьте его в список файлов. Используемые здесь параметры не влияют на значения меню настроек RARS и наоборот.

1.5 Ограничения ассемблера и эмулятора RARS

RARS разработан для эмуляции версии RV32IMFN. Ограничения RARS версии 1.0 включают:

- Сегменты памяти (текст, данные, стек) ограничены 4 МБ каждый, начиная с соответствующих базовых адресов.

- Конвейерного режима нет.
- Если вы откроете файл, который является ссылкой или ярлыком на другой файл, RARS не откроет целевой файл. Диалоговое окно открытия файла реализовано с использованием Java Swing JFileChooser, который не поддерживает ссылки.
- Очень немногие изменения конфигурации, кроме тех, что в меню настроек, сохраняются от одного сеанса к другому. Настройки редактора, включая настройки шрифта и отображение номеров строк, сохраняются.
- IDE будет работать только с ассемблером RARS. Его нельзя использовать ни с каким другим компилятором, ассемблером или симулятором. Ассемблер и симулятор RARS можно использовать либо через IDE, либо из командной строки.
- Поддержка операций с плавающей запятой не полностью совместима, поскольку Java не обеспечивает достаточно низкоуровневый доступ к операциям с плавающей запятой.
- Регистры управления и состояния не могут быть указаны в инструкциях по имени. Это полностью решаемая проблема.
- Поддержка прерываний имеет некоторые недостатки, необходимо проделать большую работу, чтобы приблизить ее к спецификации.
- Ошибка: Подсветка сообщений об ошибках не выбирает автоматически код первой ошибки сборки, если файл, содержащий ошибку, не открыт во время сборки (сборка-при-открытии, сборка-все).
- Ошибка: в редакторе произошла утечка памяти. Несколько разных людей независимо друг от друга сообщали об одном и том же: сильное замедление реакции редактора во время продолжительного интерактивного сеанса. Если выйти из RARS и перезапустить его, это поведение исчезает, и редактор мгновенно реагирует на действия.

Таблица 1.1 — Опции командной строки запуска RARS

Опция	Описание
a	Только собрать, не выполнять
ae<n>	Завершает RARS целочисленным кодом выхода <n>, если возникает ошибка сборки
ascii	Отображать содержимое памяти или регистров, интерпретируемое как коды ASCII
b	brief — не отображать адрес регистра/памяти вместе с содержимым
d	отображать операторы отладки RARS
dec	отображать содержимое памяти или регистра в десятичном формате
dump	<сегмент> <формат> <файл> — дамп памяти указанного сегмента памяти в указанном формате в указанный файл. Вариант может повторяться. Дамп происходит в конце симуляции, если не используется опция «a». Сегмент и формат чувствительны к регистру. Возможные значения: <сегмент> = .text, .data или диапазон, например 0x400000-0x10000000 <format> = SegmentWindow, HexText, AsciiText, HEX, Binary, BinaryText
g	включить режим графического интерфейса
h	показать эту справку. Использовать отдельно без имени файла.
hex	отображать содержимое памяти или регистра в шестнадцатеричном формате (по умолчанию)
ic	отображать количество основных инструкций, "исполненных"
mc <config>	установить конфигурацию памяти. Аргумент <config> равен чувствительны к регистру и возможные значения: Default для значения по умолчанию 32-битное адресное пространство, CompactDataAtZero для памяти 32 КБ с сегмент данных по адресу 0 или CompactTextAtZero для 32 КБ памяти с текстовым сегментом по адресу 0.
me	отображать сообщения RARS в стандартном формате err вместо стандартного вывода. Может отделять сообщения от вывода программы с помощью перенаправления
nc	не отображать уведомление об авторских правах (для более чистого перенаправленного/конвейерного вывода).
np	использование псевдоинструкций и форматов не разрешено
p	Режим проекта — собрать все файлы в том же каталоге, что и данный файл.
se<n>	завершить RARS целочисленным кодом выхода <n>, если возникает ошибка симуляции (запуска).
sm	начать выполнение с оператора с глобальной меткой main, если она определена
smc	самомодифицирующийся код — программа может записывать и переходить как к тексту, так и к сегменту данных.

Таблица 1.2 — Опции командной строки запуска RARS (продолжение)

Опция	**Описание**
`rv64`	включает 64-битную сборку и исполняемые файлы (не полностью совместим с rv32).
`<n>`	где <n> целочисленное максимальное количество шагов для моделирования. Если 0, отрицательное или не указано, максимума нет.
`x<reg>`	где <reg> номер или имя (например, 5, t3, f10) регистра, содержимое которого будет отображаться в конце выполнения. Вариант может повторяться.
`<reg_name>`	где `<reg_name>` - это имя (например, t3, f10) регистра, содержимое которого будет отображаться в конце выполнения. Вариант может повторяться.
`<m>-<n>`	диапазон адресов памяти от <m> до <n>, содержимое которого будет отображаться в конце выполнения. <m> и <n> могут быть шестнадцатеричными или десятичными, должны быть на границе слова, <m> <= <n>. Вариант может повторяться.
`pa`	Аргументы программы следуют в списке, разделенном пробелами. Эта опция должна быть помещена ПОСЛЕ ВСЕХ ИМЕН ФАЙЛОВ, потому что все, что следует за ней, интерпретируется как программный аргумент, который должен быть доступен программе во время выполнения.

2 Архитектура 32-х разрядного процессора RISC-V

2.1 Особенности семейства RISC-V

Принципы RISC:

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

2.2 Форматы команд

Процессор содержит:

- 32 регистра общего назначения, доступа к специализированным регистрам нет (в т. ч. нет регистра флагов, даже на аппаратном уровне!);
- 4 базовых типа команд (рисунок 2.1):
  - **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)
-
-
-
-
-

Обозначения на рисунке:

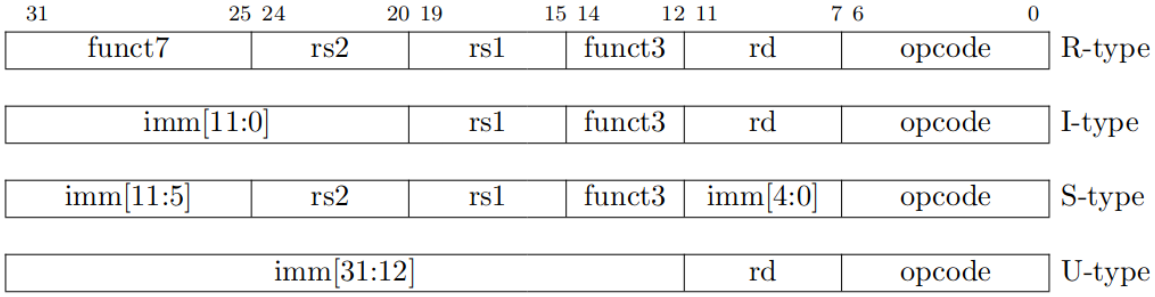

Рисунок 2.1 — Основные форматы команд 32-разрядного процессора RISC-V

- **opcode** — код операции (6 битов)
- **rs1** — № регистра-источника (5 битов)
- **rs2** — № регистра-опреанда (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.

Примечание: Ниже представлен альтернативных рисунок из Reference Card с большим числом форматов. На всякий случай. Просто ряд форматов имеют одинаковые поля, но различную семантику. Пока непонятно, что лучше и проще преподносить. Может сказать о второй форме как способ уточнения семантики?

Шесть базовых типов команд (рисунок 2.2):

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	opcode
I	imm[11:0]			rs1	funct3	rd	opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
B	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
J	imm[20|10:1|11|19:12]					rd	opcode

Рисунок 2.2 — Основные форматы команд 32-разрядного процессора RISC-V

2.3 Базовый набор команд процессора

Процессор содержит основной набор команд, каждая из который отображается в одну инструкцию на языке ассемблера. Вместе с тем ассемблер, для повышения удобства программирования, дополнительно поддерживает псевдокоманды, каждая из которых может кодировать до нескольких машинных команд или иметь специфические операнды, позволяющие сформировать мнемонику команды, удобную для восприятия человеком.

Арифметические команды представлены в таблице 2.1

Таблица 2.1 — Арифметические команды набора RV32I

Команда	**Формат**	**Описание**
`add rd,rs1,rs2`	R	Целочисленное сложение: `rd = rs1 + rs2`
`addi rd,rs1,int12`	I	Сложение непосредственно с числом: `rd = rs1 + int12`
`sub t1,t2,t3`	R	Subtraction: set t1 to (t2 minus t3)
`lui t1,100000`	U	Load upper immediate: set t1 to 20-bit followed by 12 0s
`auipc rd,100000`	U	Сложение старших 20 разрядов непосредственного числа с `pc`: `rd = pc + int[31:12]` (pc плюс 20-бит непосредственного операнда как старшие разряды 32-разрядного числа, расширенного нулями)

Логические команды представлены в таблице 2.2
Команды сдвига представлены в таблице 2.3
Команды сравнения представлены в таблице 2.4
Команды ветвления представлены в таблице 2.5
Команды перехода и связывания представлены в таблице 2.6
Команды синхронизации представлены в таблице 2.7
Команды взаимодействия с окружением представлены в таблице 2.9
Команды работы с регистром состояния представлены в таблице 2.9
Команды загрузки данных представлены в таблице 2.10

Таблица 2.2 — Логические команды набора RV32I

Команда	**Формат**	**Описание**
`add rd,rs1,rs2`	R	Целочисленное сложение: `rd = rs1 + rs2`
`addi rd,rs1,int12`	I	Сложение непосредственно с числом: `rd = rs1 + int12`
`sub t1,t2,t3`	R	Subtraction: set t1 to (t2 minus t3)
`lui t1,100000`	U	Load upper immediate: set t1 to 20-bit followed by 12 0s

Таблица 2.3 — Команды сдвига набора RV32I

Команда	**Формат**	**Описание**
`xor t1,t2,t3`	R	Bitwise XOR : Set t1 to bitwise XOR of t2 and t3
`xori t1,t2,-100`	I	Bitwise XOR immediate : Set t1 to bitwise XOR of t2 and sign-extended 12-bit immediate
`or t1,t2,t3`	R	Bitwise OR : Set t1 to bitwise OR of t2 and t3
`ori t1,t2,-100`	I	Bitwise OR immediate : Set t1 to bitwise OR of t2 and sign-extended 12-bit immediate
`and rd,rs1,rs2`	R	Поразрядное И: `rd = rs1 & rs2`
`andi rd,rs1,int12`	I	Поразрядное И непосредственно с числом: `rd = rs2 & int12` (с расширением знакового разряда у int12)

Команды выгрузки данных представлены в таблице 2.11

Команды умножения, деления, выделения остатка, расширяющие базовый набор, и используемые в эмулятора RARS, представлены в таблице 2.12

Команды для работы по прерываниям представлены в таблице 2.12

Введенные обозначения:

int12 — 12-разрядное целое со знаком

Таблица 2.4 — Команды сравнения набора RV32I

Команда	**Формат**	**Описание**
`slt t1,t2,t3`	R	Set less than : If t2 is less than t3, then set t1 to 1 else set t1 to 0
`slti t1,t2,-100`	I	Set less than immediate : If t2 is less than sign-extended 12-bit immediate, then set t1 to 1 else set t1 to 0
`sltu t1,t2,t3`	R	Set less than : If t2 is less than t3 using unsigned comparision, then set t1 to 1 else set t1 to 0
`sltiu t1,t2,-100`	I	Set less than immediate unsigned : If t2 is less than sign-extended 16-bit immediate using unsigned comparison, then set t1 to 1 else set t1 to 0

Таблица 2.5 — Команды ветвления набора RV32I

Команда	**Формат**	**Описание**
`beq t1,t2,label`	B	Переход, если равно: Переход к оператору по label, если `t1 == t2`.
`bge t1,t2,label`	B	Перейти, если больше или равно: перейти к оператору по label, если `t1 >= t2`
`bgeu t1,t2,label`	B	Переход, если больше или равно для беззнаковых чисел: переход к оператору по адресу метки, если `t1 >= t2` (с беззнаковой интерпретацией)
`blt t1,t2,label`	B	Переход, если меньше: Переход к оператору по label, если `t1 < t2`
`bltu t1,t2,label`	B	Переход, если меньше для беззнаковых чисел: переход к оператору по label, если `t1 < t2` (с беззнаковой интерпретацией)
`bne t1,t2,label`	B	Переход, если не равно: Переход к оператору по label, если `t1 != t2`.

Таблица 2.6 — Команды ветвления набора RV32I

Команда	**Формат**	**Описание**
`jal t1, target`	J	Jump and link : Set t1 to Program Counter (return address) then jump to statement at target address
`jalr t1, t2, -100`	I	Jump and link register: Set t1 to Program Counter (return address) then jump to statement at t2 + immediate

Таблица 2.7 — Команды синхронизации набора RV32I

Команда	**Формат**	**Описание**
`fence 1, 1`	I	Ensure that IO and memory accesses before the fence happen before the following IO and memory accesses as viewed by a different thread
`fence.i`	I	Ensure that stores to instruction memory are visible to instruction fetches

Таблица 2.8 — Команды взаимодействия с окружением набора RV32I

Команда	**Формат**	**Описание**
`ebreak`	I	Pause execution
`ecall`	I	Issue a system call: Execute the system call specified by value in a7

Таблица 2.9 — Команды работы с регистром состояния набора RV32I

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

Таблица 2.10 — Команды загрузки данных набора RV32I

Команда	**Формат**	**Описание**
`lb t1, -100(t2)`	I	Set t1 to sign-extended 8-bit value from effective memory byte address
`lbu t1, -100(t2)`	I	Set t1 to zero-extended 8-bit value from effective memory byte address
`lh t1, -100(t2)`	I	Set t1 to sign-extended 16-bit value from effective memory halfword address
`lhu t1, -100(t2)`	I	Set t1 to zero-extended 16-bit value from effective memory halfword address
`lw t1, -100(t2)`	I	Set t1 to contents of effective memory word address

Таблица 2.11 — Команды выгрузки данных набора RV32I

Команда	**Формат**	**Описание**
`sb t1, -100(t2)`	S	Store byte : Store the low-order 8 bits of t1 into the effective memory byte address
`sh t1, -100(t2)`	S	Store halfword : Store the low-order 16 bits of t1 into the effective memory halfword address
`sw t1, -100(t2)`	S	Store word : Store contents of t1 into effective memory word address

Таблица 2.12 — Команды умножения, деления, вычисления остатка набора RV32I

Команда	**Формат**	**Описание**
`mul t1,t2,t3`	R	Multiplication: set t1 to the lower 32 bits of t2*t3
`mulh t1,t2,t3`	R	Multiplication: set t1 to the upper 32 bits of t2*t3 using signed multiplication
`mulhsu t1,t2,t3`	R	Multiplication: set t1 to the upper 32 bits of t2*t3 where t2 is signed and t3 is unsigned
`mulhu t1,t2,t3`	R	Multiplication: set t1 to the upper 32 bits of t2*t3 using unsigned multiplication
`div t1,t2,t3`	R	Division: set t1 to the result of t2/t3
`divu t1,t2,t3`	R	Division: set t1 to the result of t2/t3 using unsigned division
`rem t1,t2,t3`	R	Remainder: set t1 to the remainder of t2/t3
`remu t1,t2,t3`	R	Remainder: set t1 to the remainder of t2/t3 using unsigned division

Таблица 2.13 — Команды для работы с прерываниями эмулятора RARS

Команда	**Формат**	**Описание**
`uret`	?	Return from handling an interrupt or exception (to uepc)
`wfi`	?	Wait for Interrupt

3 Сопроцессор с плавающей точкой

3.1 Система команд сопроцессора

> **Примечание**: Думаю, что имеет смысл, как и в случае базовых команд, разбить на несколько таблиц, сгруппировав по назначению.

Таблица 3.1 — Базовые команды процессора RISC-V (продолжение 1)

Опция	**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
`fclass.d t1, f1`	Classify a floating point number (64 bit)
`fclass.s t1, f1`	Classify a floating point number
`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
`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
`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
`fld f1, -100(t1)`	Load a double from memory
`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

Таблица 3.2 — Базовые команды процессора RISC-V (продолжение 2)

Опция	**Описание**
`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
`flw f1, -100(t1)`	Load a float from memory
`fmadd.d f1, f2, f3, f4, dynFused`	Multiply Add (64 bit): Assigns f2*f3+f4 to f1
`fmadd.s f1, f2, f3, f4, dynFused`	Multiply Add: 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
`fmsub.d f1, f2, f3, f4, dynFused`	Multiply Subatract: Assigns f2*f3-f4 to f1
`fmsub.s f1, f2, f3, f4, dynFused`	Multiply Subatract: Assigns f2*f3-f4 to f1
`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
`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
`fnmadd.d f1, f2, f3, f4, dynFused`	Negate Multiply Add (64 bit): Assigns -(f2*f3+f4) to f1
`fnmadd.s f1, f2, f3, f4, dynFused`	Negate Multiply Add: Assigns -(f2*f3+f4) to f1
`fnmsub.d f1, f2, f3, f4, dynFused`	Negated Multiply Subatract: Assigns -(f2*f3-f4) to f1
`fnmsub.s f1, f2, f3, f4, dynFused`	Negated Multiply Subatract: Assigns -(f2*f3-f4) to f1
`fsd f1, -100(t1)`	Store a double to memory
`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

Таблица 3.3 — Базовые команды процессора RISC-V (продолжение 3)

Опция	**Описание**
`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 (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
`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
`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
`fsw f1, -100(t1)`	Store a float to memory

Заключение

Продолжение следует...

Литература

[1] Таненбаум Э. Архитектура компьютера. 6-е изд. — СПб.: Изд. Питер, 2017. — 816 с.

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

[3] 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.

[4] Легалов А.И. О разработке программного обеспечения. Модельный взгляд. — `http://softcraft.ru/notes/devproc/`

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

[6] 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.

[7] Брайант Рэндал Э., О'Халларон Дэвид Р. Компьютерные системы: архитектура и программирование. 3-е изд. / пер. с англ. А. Н. Киселева. — М.: ДМК Пресс, 2022. — 994 с.

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

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

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

[11] Ричард Столмен, Роланд Пеш, Стан Шебс и др. Отладка с помощью GDB. — 2000. `https://www.opennet.ru/docs/RUS/gdb/`

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

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

[14] Кудрявцева И.А., Швецкий М.В. Программирование: теория типов: учебное пособие для вузов. — М.: Издательство Юрайт, 2022. — 652 с.

[15] Пирс Бенджамин Типы в языках программирования — Лямбда пресс, Добросвет, 2012.

[16] Майерс Г. Дж. Архитектура современных ЭВМ. В 2-х книгах. — М.: Мир, 1985. — 366+311 с.

[17] Органик Э. Организация системы Интел 432. — М.: Мир, 1987.

[18] IAPX 432. — Википедия `https://ru.wikipedia.org/wiki/IAPX_432`

[19] Пройдаков Эдуард. Микропроцессор 8080. `https://www.computer-museum.ru/technlgy/i8080.htm`