1. **Что такое метка (label)?**

Метка в ассемблере - это имя, которое указывает на конкретное место в программе, на которое можно сослаться в коде.

Необходимы для повышения человекочитаемости кода. С помощью них удобно создавать подобие переменных в секции .data, а также обращаться к отдельным участкам программы, т.е. совершать переходы

Сами по себе метки не занимают место в памяти.

Должны иметь уникальные имена.

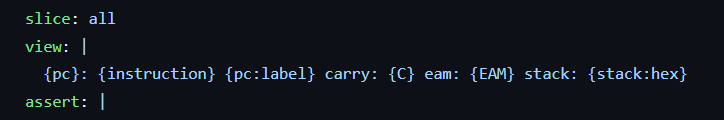
1. **Откуда рассчитываются значения меток (label)?**

Ассемблер делает два прохода по тексту программы. На первом проходе проверяется правильность синтаксиса и рассчитываются значения меток. Тому, как он это делает, и будет посвящена наша сегодняшняя история.

Ассемблер идет последовательно, начиная с начала сегмента .data (ячейка памяти с адресом 0) Он считает размер каждой директивы (.byte, .word) или инструкции в байтах, а когда встречает метку - присваивает метке значение, равное текущему счетчику байтов в таблице значений меток (пикча из первого вопроса) Размер каждой инструкции известен ассемблеру (зависит от архитектуры)

На втором проходе assembler чекает таблицу значений из первого прохода и подставляет в код вместо меток реальные адреса.

1. **Как вывести в отчёте состояние регистра Acc в hex формате?**



1. **Назначение alignment (неиспользуемый буфер в секции данных)?**

Alignment (выравнивание) в секции данных используется для размещения данных на адресах, кратных определённому значению (например, 4 байта), чтобы обеспечить эффективный доступ процессора к памяти

ИЛИ

Посмотрим на dump до alignment. Видим, что инструкция load\_ind содержит в себе ячейку 128, куда не надо полезла, согласны? Сообщим ей ласково с помощью alignment, чтобы она подвинулась. Инструкции могут начинаться с 136 ячейки включительно. 136-128 = 8, делим на 4, получаем 2 байта. Таким образом, если мы сдвинем эту инструкцию на 2 байта, то она будет начинаться с 136. Просто вставляем эти 2 байта в alignment

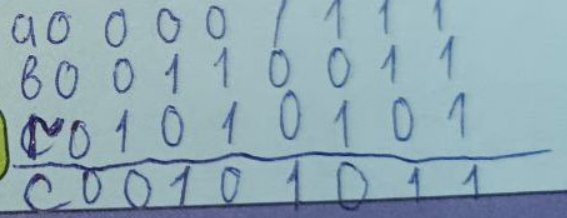
1. **acc32 Зачем определены дублирующиеся инструкции load\_addr и load?**

load занимает меньше места, но использует относительную адресацию.

Instruction size: 1 byte for opcode, 4 bytes for absolute operand, 2 bytes for relative operand. Control flow, Load/Store Immediate/Indirect/Addr use absolute address, other -- relative.

1. **acc32 Переполнение и перенос. Overflow и Carry флаги, иные способы обработки.**

Вопрос об иных способах обработки расплывчатый, но предположим. 1 способ. Если у вас был дискродед на первом курсе, то шарите, как это можно сделать с помощью анализа старшего бита каждого из операндов и старшего бита результата. Вот, к примеру, я расписывал все возможные комбы для вычисления, будет ли carry или нет, где a - старший бит операнда a, b - b, r - r, логично. c - это carry, 1 - установлен, 0 - не установлен. Если будете вникать в эти комбы и думать, почему a = 0, b = 0, a r = 1, к примеру, то помните про возможный перенос 1 при сложении младших разрядов. Для флага переполнения свои комбы. Итак, в программе это можно реализовать, получится программный контроль флагов, а не аппаратный.



Есть еще подход, который не поддерживается в нашей архитектуре, но имеет место быть теоретически. Расширение операндов и результата до большего размера, к примеру, 64 бит, выполнение вычислений и далее проверка, что он не выходит за минимальную и максимальную границы 32 битного числа.

Еще как-то мутил другую темку, кароч если нам нужно сделать сложение и проверить, будет ли переполнение, мы из максимально возможного 32-битного числа отнимаем второй операнд, и проверяем, что если первый операнд больше получившейся разности, то переполнению быть. Вот так вот, братцы.

1. **Что такое Memory Mapped IO?**

У нас есть различные устройства ввода-вывода. С ними нужно обмениваться инфой, то есть принимать для обработки данные и отправлять полученные результаты. Memory Mapped IO позволяет процессору работать с устройствами как с обычными адресами в памяти (команды чтения и записи). То есть он пишет в адрес или читает из него, а на самом деле происходит обращение к устройству. То есть мы мапим определенный адрес, к примеру 0x80, и говорим, что это input port.

Это необходимо для того, чтобы, во-первых, не создавать отдельное адресное пространство для портов ввода-вывода, а во-вторых, не заводить специальные инструкции для операция записи и чтения из этих портов.

1. **Что будет, если процессор попытается прочитать инструкцию из адреса ввода-вывода?**



В другой архитектуре просто может быть считана мусорная инструкция. Но

1. **Что означает инструкция ...? Объясните что происходит в строках ....**

* Branch if Less Than Zero

Syntax: **ble** <address>

Description: Jump to a specific address if the accumulator is less than zero.

Operation: if acc < 0 then pc <- <address>

* Branch if Equal to Zero

Syntax: **beqz** <address>

Description: Jump to a specific address if the accumulator is zero.

Operation: if acc == 0 then pc <- <address>

* Branch if Not Equal to Zero

Syntax: bnez <address>

Description: Jump to a specific address if the accumulator is not zero.

Operation: if acc != 0 then pc <- <address>

1. **Как сократить объём машинного кода? Какие ограничение/проблемы это создаст?**

На примере f32a

lit 1 +

Целых две команды, но мы ведь знаем, что это доступный в других языках программирования inc. Так вот, чтобы сократить объем машинного кода, мы берём и вводим новые инструкции. У этого подхода есть минус, связанный с увеличением сложности процессора или интерпретатора команд.

Второй подход можно уменьшить размер загружаемых immediate\_value или адресов, тогда размер инструкций сократится, но диапазон доступной памяти и данных ограничится.

1. **Особенности кодирования инструкций, плотность кода, проблемы доступа.**

Особенности кодирования инструкций: Есть архитектуры с фиксированной длиной инструкций, к примеру, risc-iv все по 4 байта, а есть архитектуры с переменной длиной, к примеру, в f32a есть по 5, 1, ил 4 байта. К недостаткам первой можно отнести низкую плотность кода, но при этом такие инструкции легко декодировать, ведь заранее известно, через сколько байт закончится инструкция. К недостаткам второй относится более сложное декодирование (необходим механизм, определяющий длину инструкции), однако это окупается высокой плотностью кода (простые операции занимают меньше места в памяти)

Плотность кода: Нужно выполнить набор команд. Высокая плотность кода будет означать, что программы будут использовать меньшее количество памяти для выполнения определенного функционала, чем программы с более низкой плотностью кода. Грубо говоря, плотность показывает, насколько эффективно используются n ячеек памяти для решения поставленной задачи.

Проблемы доступа: Гонки за владение ресурсами при многопоточной обработке, недоступность для ввода-вывода устройств, ограничения размера памяти, перегрузка шины данных из-за большого числа устройств и тд.

1. **Как вы можете сократить объём машинного кода? Продемонстрируйте.**

Спорный вопрос, по идее возникнуть не должен, так как я уверен, что делал все максимально эффективно, но всякое бывает. На практике, к примеру, повторяющиеся участки кода можно выделить в одну процедуру и вызывать её, это сократит объем кода. Просто, смотрите на куски кода и думаете, как я могу это сделать за меньшее число команд. Как вариант, можно добавить чутка неэффективности в программу для защиты от пу-пу-пууу во время ответа на этот вопрос на сдаче.

1. **f32a Почему в F32a удобно использовать процедуры?**

Процедуры - это исполняемые части кода, вызов процедуры делаем с помощью call, возврат из неё - с помощью return.

В f32a очень удобно их использовать благодаря стеку, так как не нужно продумывать передачу аргументов, они просто кладутся через push на стек, а процедура снимает их через pop(у).

Также существует независимость между процедурами, то есть одна никак не влияет на другую, ведь можно считать, что у каждой процедуры выделена своя часть стека, с которой она взаимодействует.

Возврат из процедуры тоже бархатно проходит, спасибо стеку возврата, который хранит адреса возвратов.

Также не нужно делать сохранение и восстановление регистров.

Рекурсивные вызовы офигенно работают в стековой архитектуре, опять же благодаря независимому пространству стека у каждой процедуры.

1. **f32a Extended arithmetic mode. Назначение и использование.**

EAM - extended arithmetic mode - он позволяет нам совершать операцию сложения add с добавлением к результату carry флага, если предыдущий результат сложения превысил размер 32 бита, то есть флаг carry был установлен. Чтобы понять, как это работает посмотрите программу и результаты её тестирования

1. **risc-iv Что означает конструкция 0(t1)? Зачем она нужна?**

Если кратко, то это по типу указателей на языке C, разыменование адреса. Если спросят про смещение, расскажите про итерации в массиве.

Если смещение 0, то загрузка происходит напрямую из адреса, записанного в регистр. Если немного знакомы с синтаксисом языка C, то 0(t0) похоже на разыменование адреса в C, то есть \*ptr, к примеру. Если мы работаем с массивами, то offset можно использовать для доступа к отдельным элементам, то есть 0(t0) это первый элемент, 4(t0) это второй и т.д (4, потому что размер инструкции 4 байта, помним, гордимся)

Это нужно, когда данные находятся не точно по адресу в регистре, а на некотором расстоянии от него, например, при работе с массивами, структурами, локальными переменными в стеке или аргументами, переданными через стек. Ненулевой офсет позволяет гибко адресовать память без необходимости модифицировать базовый адрес в регистре.

1. **risc-iv Почему для загрузки слова регистр нужно две инструкции?**

Это делается с помощью побитового умножения на маску. Необходимость их использование объясняется тем, что в стандартных инструкциях не предусмотрена возможность загружать 32 битное значение, ибо не хватит места для всяких опкодов, адресов rd и rs регистров и т.д.

**Ограничение размера непосредственного значения**:

* В RISC-V инструкции имеют фиксированную длину 32 бита, из которых часть бит выделяется на код операции, регистры и непосредственное значение.
* Непосредственные значения в инструкциях ограничены:
  + Для **I-type** инструкций (например, addi) непосредственное значение — **12 бит** (знаковое, от -2048 до 2047).
  + Для **U-type** инструкций (например, lui) непосредственное значение — **20 бит**, но оно загружается в старшие биты регистра.
* Чтобы загрузить полное **32-битное значение** (например, адрес 0x12345678), нужно комбинировать инструкции, так как ни одна инструкция не может вместить 32-битное значение целиком.

**Загрузка 32-битного значения за одну инструкцию потребовала бы:**

* Либо переменную длину инструкций (как в CISC, например, x86), что усложняет декодирование.
* Либо выделение всех 32 бит под непосредственное значение, что невозможно в 32-битной инструкции, так как нужно место для кода операции и регистра.

RISC-V выбирает простоту: фиксированная длина инструкций и разделение загрузки на два этапа (старшие и младшие биты).

1. **risc-iv Работа со стеком. Выделение памяти, передача данных в/из процедур.**

В архитектуре RISC-V работа со стеком — это ключевой механизм для управления памятью, передачи данных в процедуры и из них, а также поддержки рекурсии. Стек используется для хранения локальных переменных, сохранённых регистров, адресов возврата и аргументов функций. Поскольку ваш код включает MMIO (адреса 0x80, 0x84), рекурсивную функцию little\_to\_big и инициализацию стека через lui sp, 0x200, я объясню, как организовать выделение памяти и передачу данных в RISC-V, с примерами, основанными на вашем контексте.

**Основы работы со стеком в RISC-V**

1. **Стек**:
   * Стек — область памяти, организованная по принципу **LIFO** (Last In, First Out).
   * В RISC-V стек **растёт вниз** (от больших адресов к меньшим), как в вашем коде с lui sp, 0x200 (устанавливает sp в 0x200000).
   * Указатель стека (**sp**, регистр x2) указывает на **вершину стека** (наименьший адрес текущего фрейма).
   * Стек должен быть выровнен по **16 байт** (требование ABI).
2. **Фрейм стека**:
   * Фрейм стека — участок памяти, выделенный для одной функции.
   * Создаётся в **прологе** функции и уничтожается в **эпилоге**.
   * Содержит:
     + **Адрес возврата** (ra, регистр x1).
     + **Сохранённые регистры** (callee-saved: s0–s11, или caller-saved, если нужно).
     + **Локальные переменные**.
     + **Аргументы** для вызова других функций (если передаются через стек).
     + **Указатель фрейма** (fp, обычно s0/x8), если используется.
3. **ABI RISC-V**:
   * **Caller-saved регистры** (t0–t6, a0–a7, ra): Вызывающая функция сохраняет их, если значения нужны после вызова.
   * **Callee-saved регистры** (s0–s11, sp): Вызываемая функция сохраняет их, если использует.
   * Аргументы передаются через a0–a7; если их больше 8, используются стек.
   * Результат возвращается в a0 (и a1, если нужно два значения).