Ядро́ (англ. kernel) — центральная часть операционной системы (ОС), обеспечивающая приложениям

координированный доступ к ресурсам компьютера, таким как процессорное время, память,

внешнее аппаратное обеспечение, внешнее устройство ввода и вывода информации.

Также обычно ядро предоставляет сервисы файловой системы и сетевых протоколов.

Ядро - привилегированный код (имеет полный контроль над железом)

Ядро - реагирующая система, не работает постоянно, а тлько в ответ на

внешние события. Например,

1) приложение пытается вызвать сервис Операционной Системы,

оно делает !системный вызов и управление передается ядру ос

2) !Прерывание - практически тот же системный вызов, но со стороны устройства

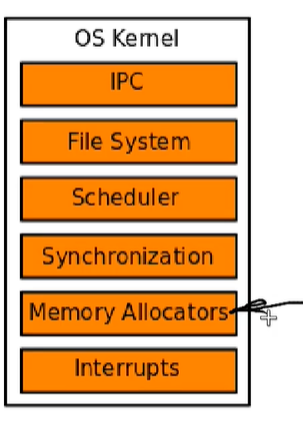
------

Какие сервисы ядро может предоставлять приложениям:

1) Доступ к устройствам

2) Файловая система

3)Средства взаимодействия между пользовательскими приложениями, средства их запуска, заверш.

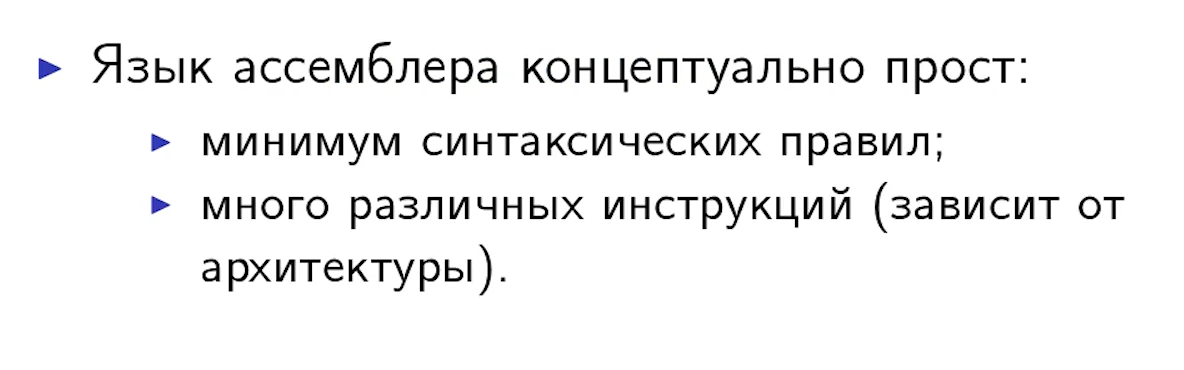


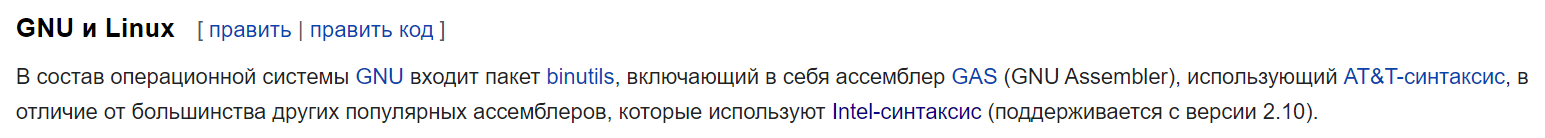
InterProcessCommunication – обмен данными между потоками одного или разных процессов

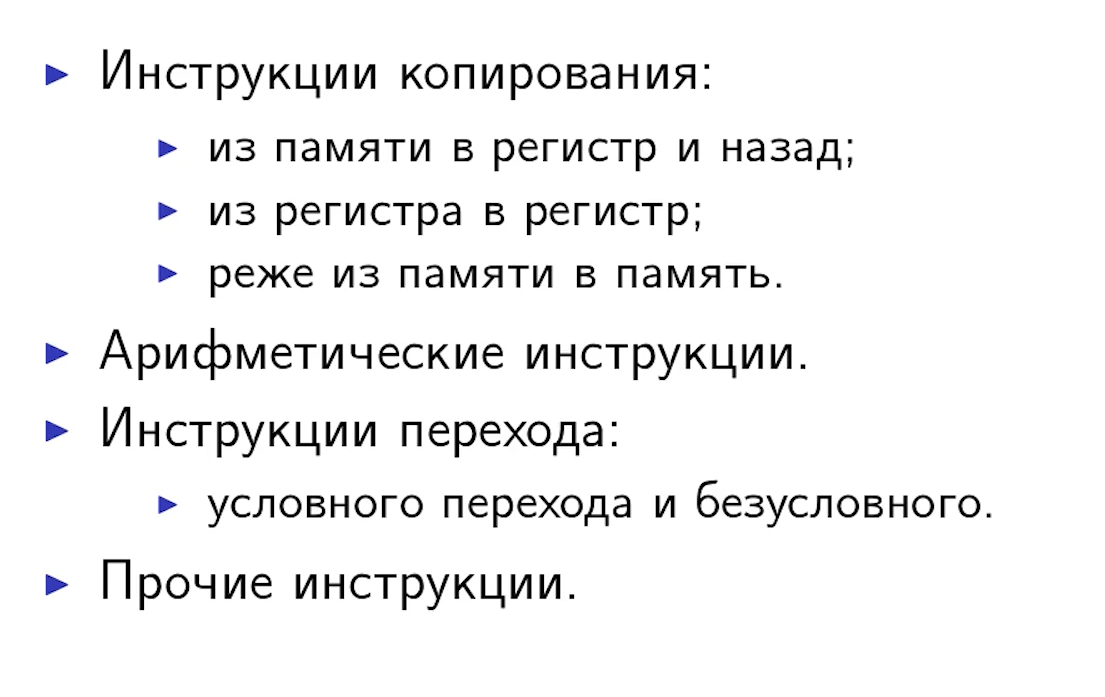
Планировщик – когда и какой поток должен исполняться на процессоре

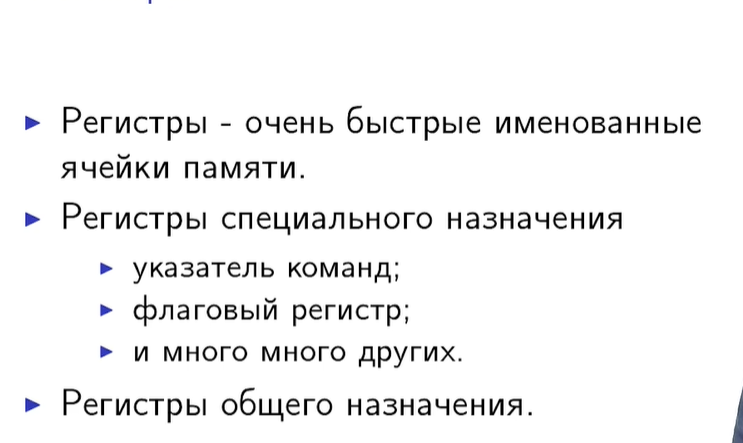
Динамическая аллокация памяти

Interrupts – прерывания





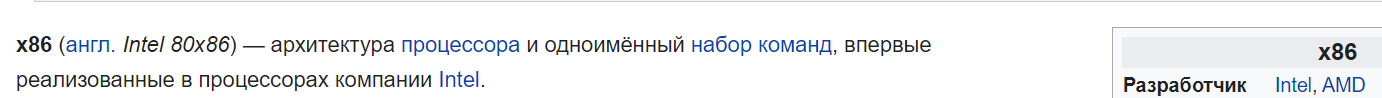


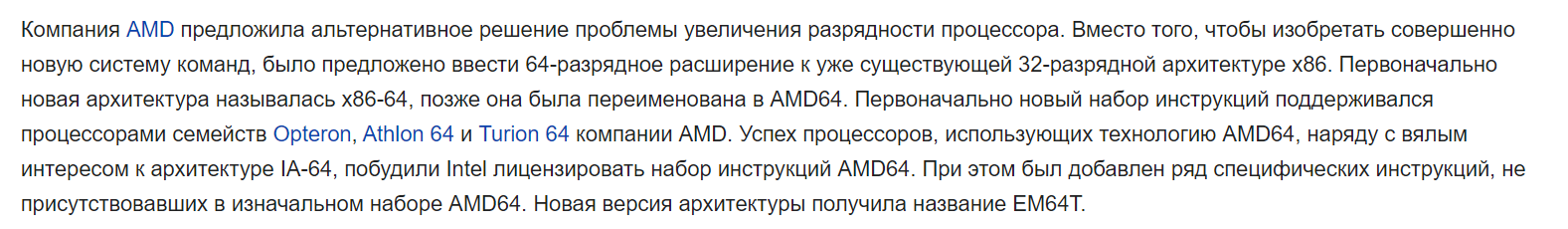


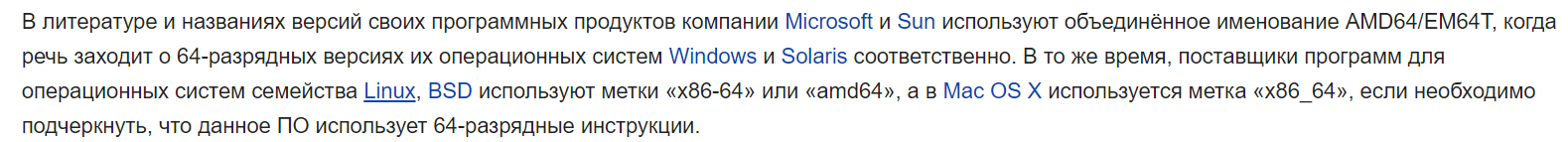
Например, регистр указатель команд – содержит следующую или посл. исполненую процессором инструкцию.

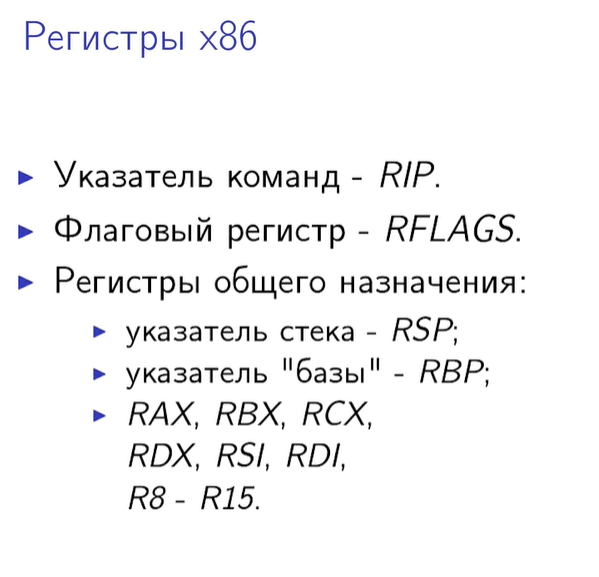
Флаговый регистр – хранит состояние процессора

Регистры общего назначения – например хотим посчитать суму 2-х чисел, одно загружаем из памяти в 1 регистр, второе в 2-й регистр, суму в 3-й. Условно говоря они взаимозаменяемые.



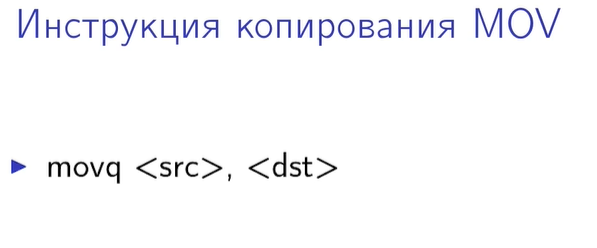






Instruction Pointer

Инструкции



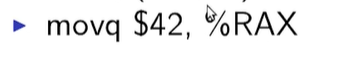
Именно копирование, не просто перенос!

Инструкция с суфиксом q, - означает что копируем 8-мибайтные (64-битные значения)

Перед именем регистра всегда ставим %



Скобки () – операция разоменования. Тоесть берем значение в RAX (64-хбитное число), интерпретируем его как адрес, обращаемся по этому адресу в память и достаем оттуда 8-мибайтное значение. И записываем это в %RAX



Записываем конкретное значение в RAX

Если не ставить $, то 42 будет интерепретирован как адрес, и инструкция обратится по адресу 42, прочитает оттуда 8-мибайтное значение и запишет его в RAX.

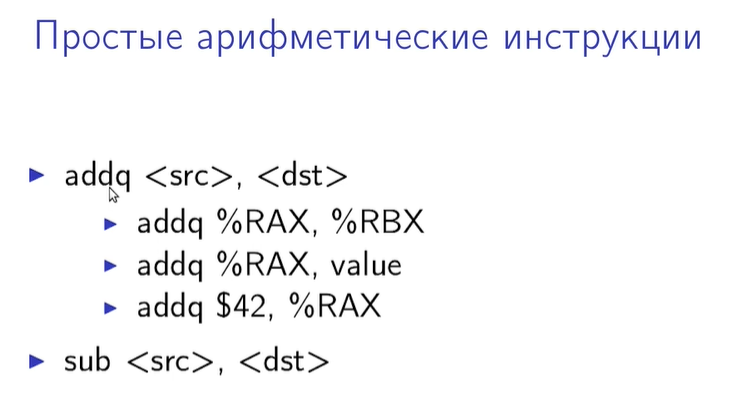


Когда есть имя value, которое соответствует какому-то адресу и мы перед именем ставим $, то значит мы используем сам адрес (значение адреса), и мы не пытаемся его разименовать.



Тут мы пытаемся разименовать, тоесть обратиться по адресу, достать значение и его скопировать в другое место

Простые арифметические инструкции



Складывает значения, на которые указывают параметры src и dst

Результат сохраняется во второй аргумент.



По факту в %RBX записывается %RAX + %RBX

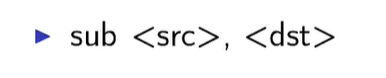


Берет значение в регистре %RAX и добавляет к значению, которое хранится по адресу, на который ссылается value.

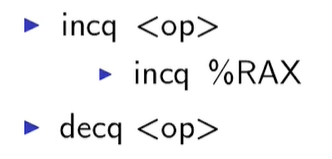


Просто добавить 42 в RAX

Вычитание (если указать суффикс, он будет говорить о размере операндов)

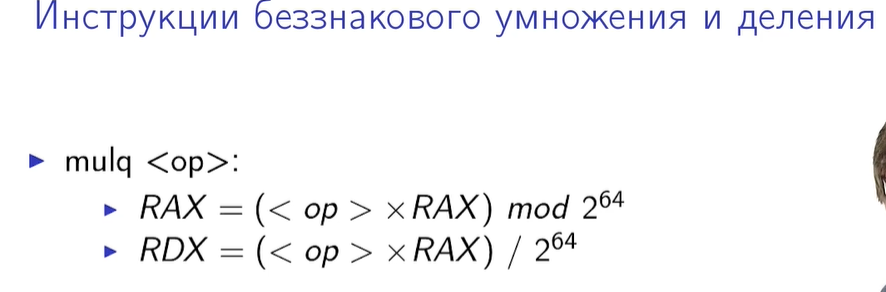


Вычитает 1-й аргумент из второго



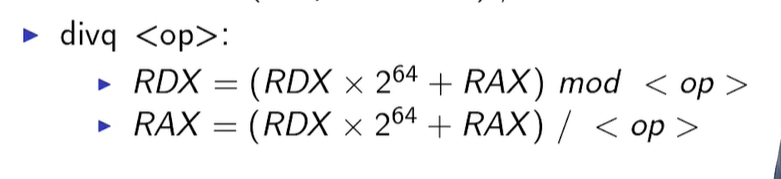
Инкремент и декремент (op – операнд)

Более сложные арифметические инструкции (умножение и деление)



Принимает 1 операнд, всегда умножает на %RAX, результат хранится в 128битном виде, где младшие биты хранятся в %RAX, старшие – в %RDX

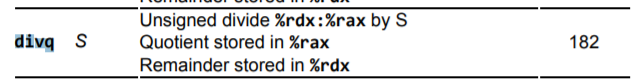
деление (divide)



в качестве 2-го аргумента тоже %RAX. (?)

Инструкция вычисляет сразу и частное и остаток. Остаток попадает в %RDX,

частное в %RAX

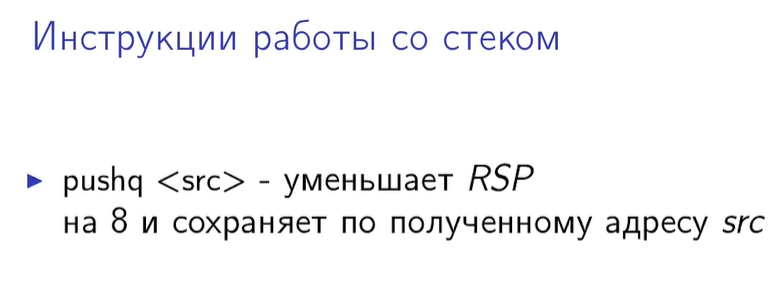


Стек – область памяти, на которую указывает регистр RSP

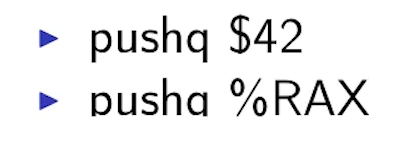


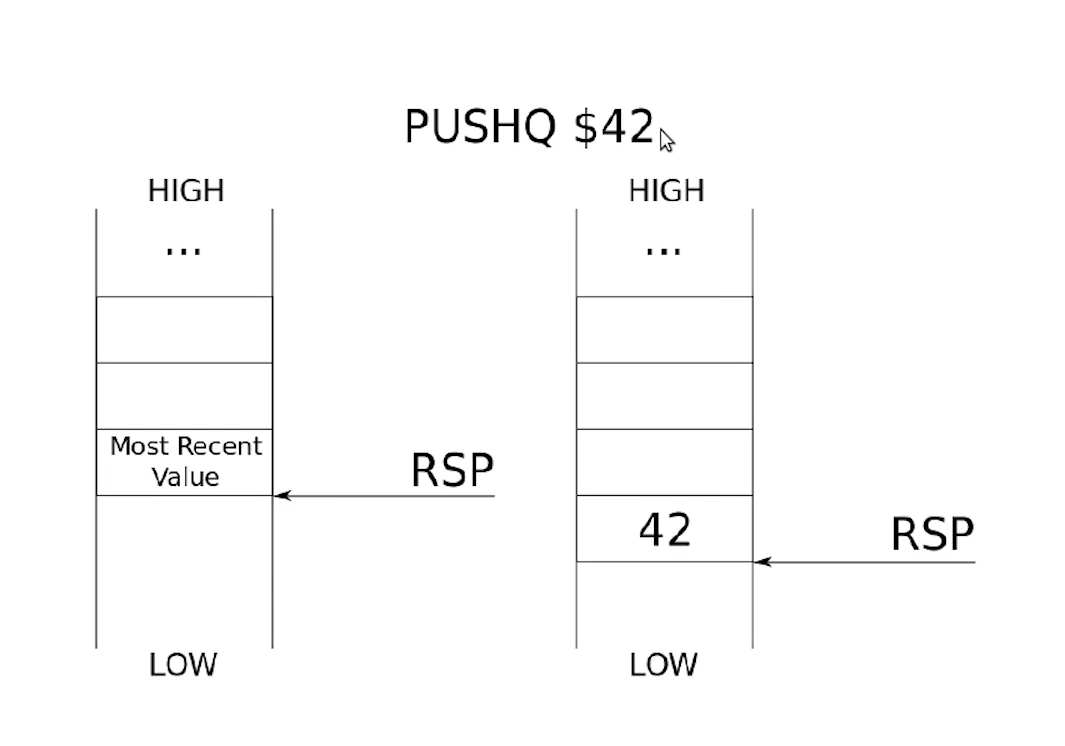
Стэк растет вниз, тоесть более новые значения на стеку хранятся по меньшим адресам, более старые – по большим адресам.

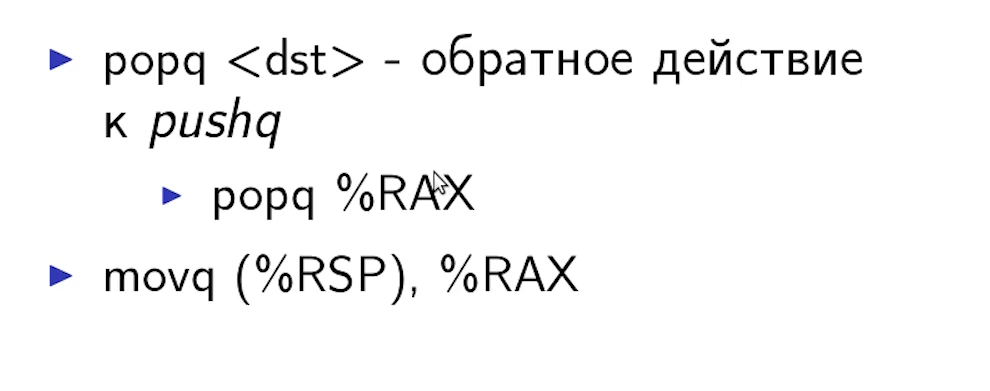
Положить значение на стек –



на 8 потому что суффикс q.

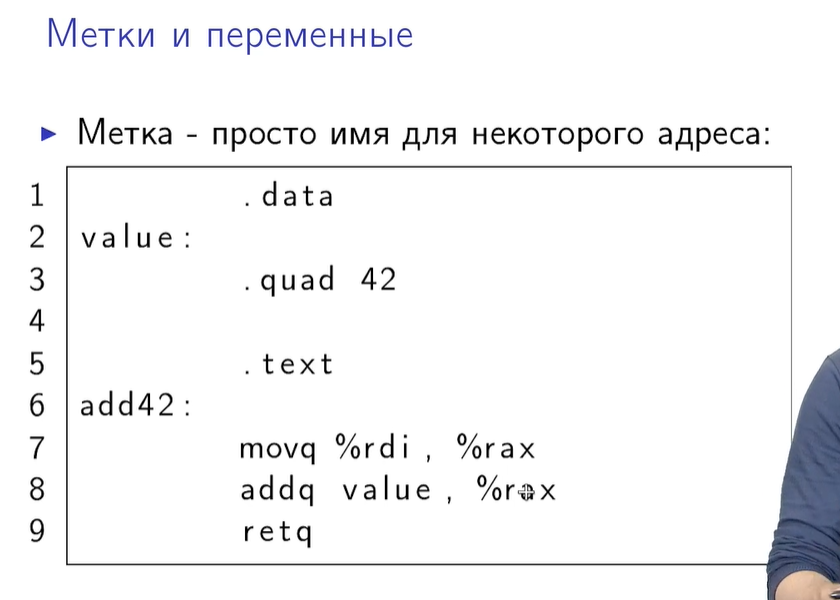




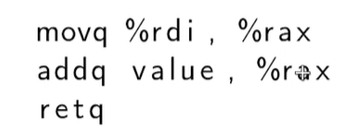


popq – если хотим извлечь, movq – если хотим переместить

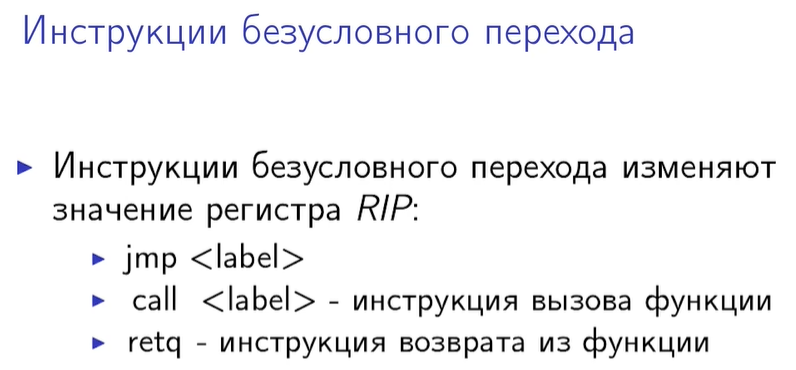
movq (%RSP), %RAX – в регистре RSP содержится адрес, и мы обращаемся по этому адресу, читаем 8-мибайтное значение (q) и сохраняем в %RAX.



здесь создаем метку value, которая ссылается на место в памяти, где хранится 8-мибайтное значение (quad) 42

Метку add42, которая указывает на начало кода 

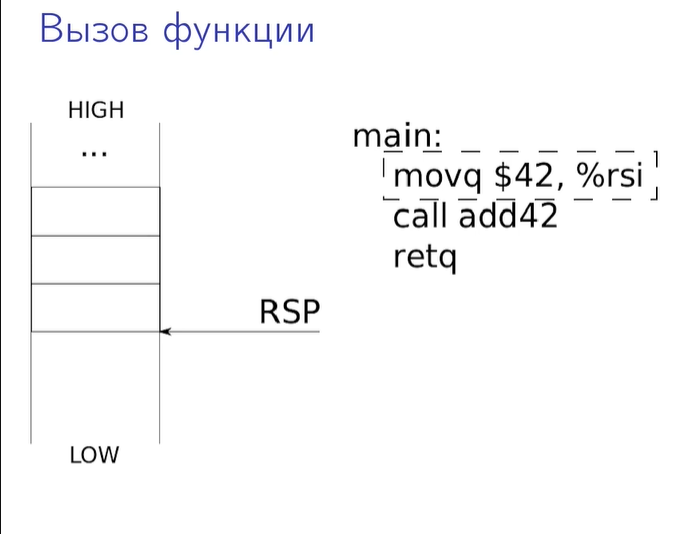
Процессор исполняет инструкции по порядку, если хотим передать управление не следующей инструкции, а какой-то другой, нам нужно использовать инструкцию перехода.



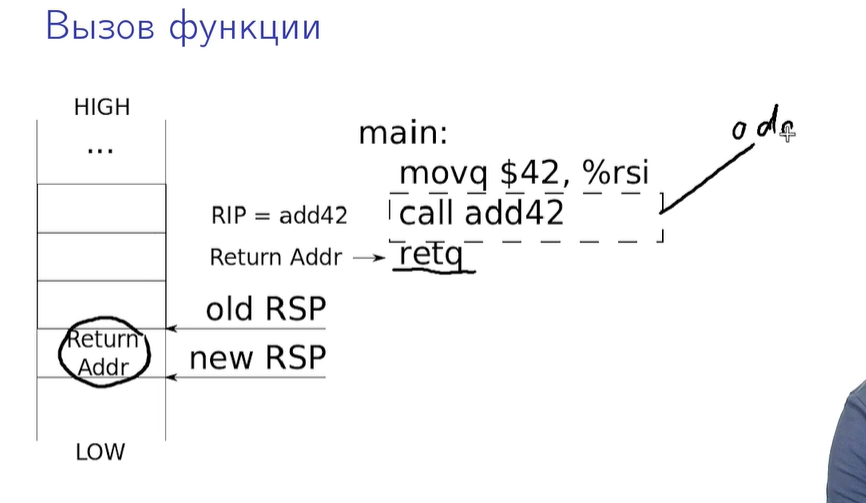
jmp <label> - принимает метку (ссылку на адрес по которому нужно передать управление)

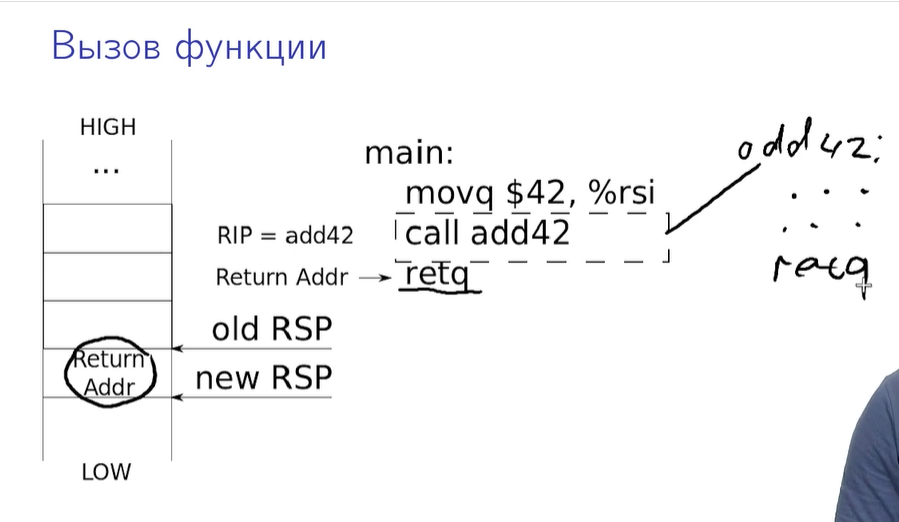
call <label> – более сложная, также передает управление, и еще делает дополнительные манипуляции

Функции

Функция должна возвращать управление вызывающему коду. Поэтому в функцие сохраняем данные откуда она была вызвана. 

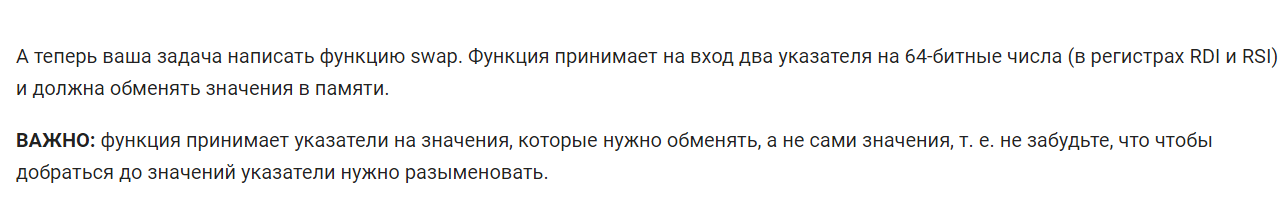
ф-я main вызывает ф-ю add42, процессор сначала запоминает адреса возврата (адреса следующей инструкции за call add42, тоесть retq). адреса возврата сохраняется на стек. Управление передается ф-и add42.

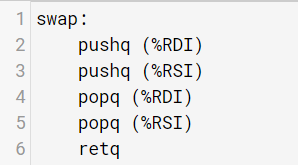




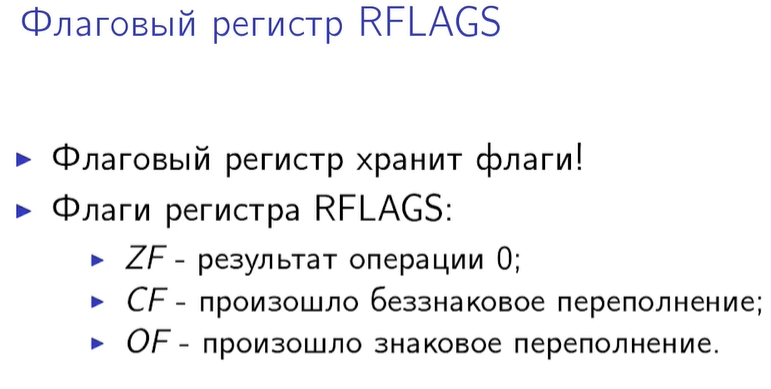
в какой-то момент ф-я add42 завершится и выполнит свою инструкции retq, которая заглянет на стек, найдет там последний адрес, примет его за адрес возврата, удалит его со стека и передаст управление этой инструкции.

!Если внутри ф-и add42 что-то сохраняли на стек, перед тем как вызывать инструкцию retq нужно все удалить с этого стека





Флаговый регистр RFLAGS



ZeroFlag

CarryFlag

OverflowFlag и другие

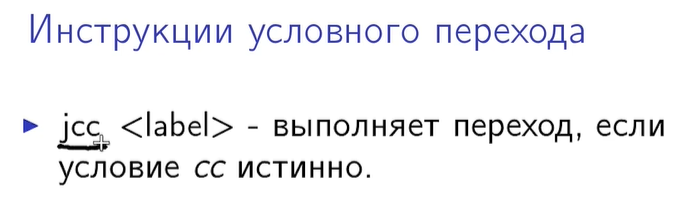
В зависимости от результата вычисления какой-то операции, будут выставлены флаги в регистре RFLAGS.

Например, сложили 2 числа и результат сложения = 0, будет установлен ZeroFlag,

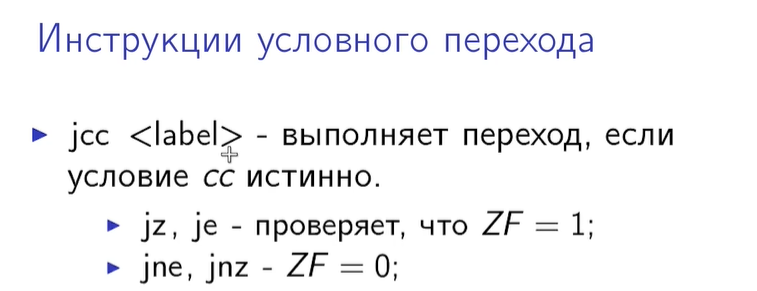
Если сложили и резалт не помещается в 64 бита, то старшие биты будут отброшены и резалт в CF (бит переноса)

Инструкции условного перехода

Используются для организации ветвлений (типа if), или циклов вроде while, которые проверяют какое-то условие и в зависимости от этого выполняют определенный код



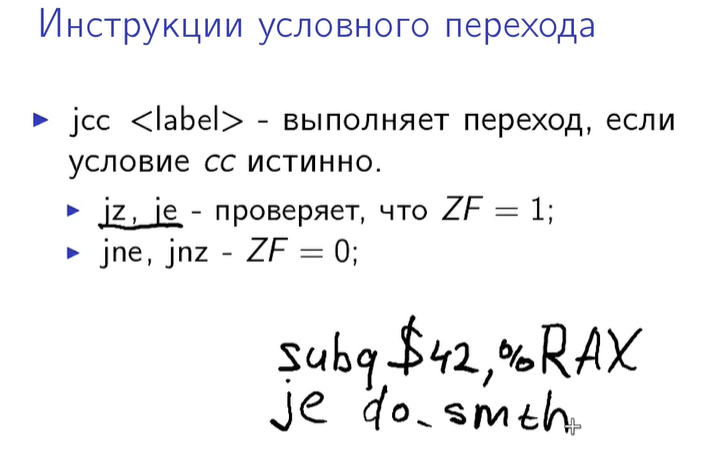
, где cc – условие, которое должна проверить инструкция, и если условие истинно, то управление передается по этой метке

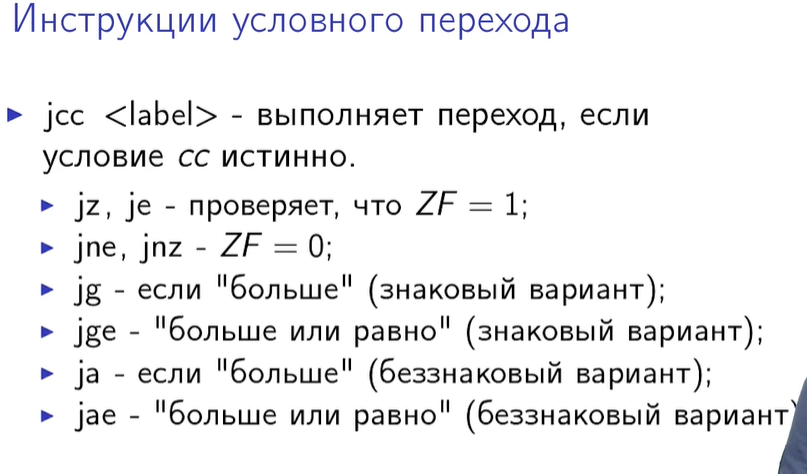


JumpIfZero, JumpIfEqual,

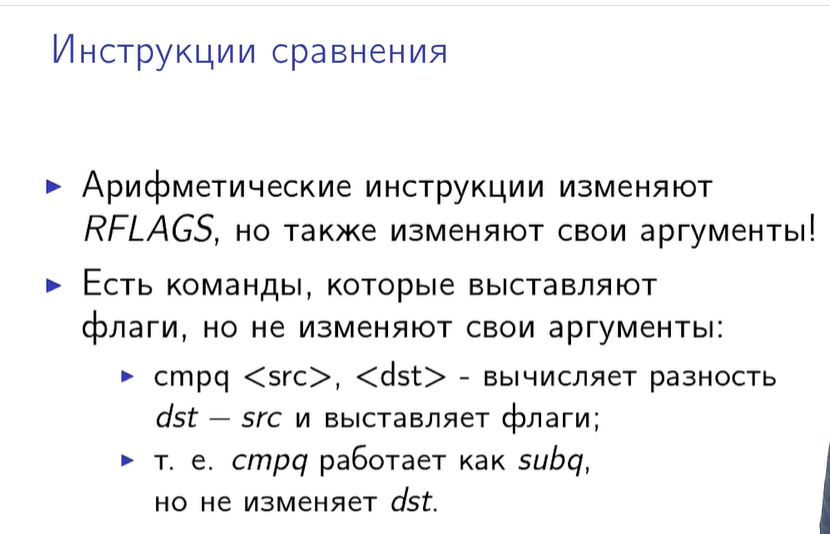
Например проверим содержится ли в %RAX число 42 и если да то do\_smth

!Но здесь также изменится значение в RAX, что побочное действие





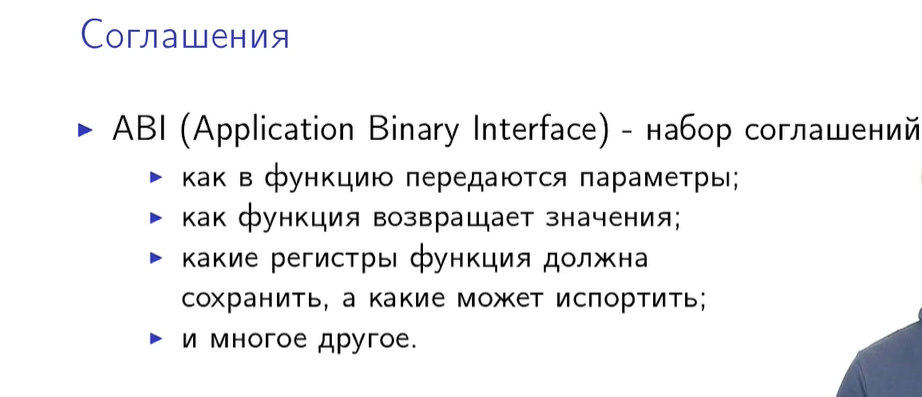
Чтобы просто выставить флаги, не изменяя значений в регистрах (просто проверка какого-то регистра на число и т.п.)

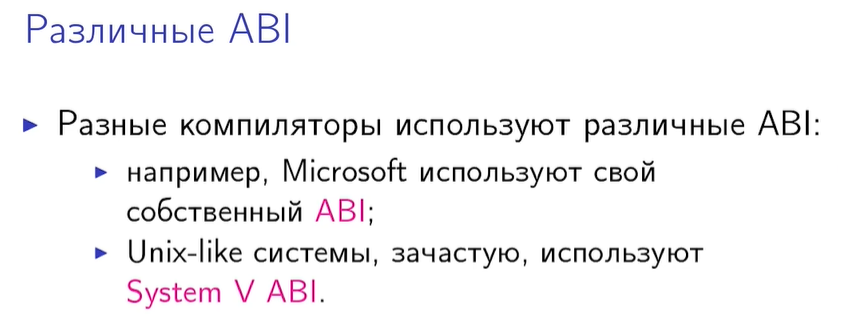


Пример: ищем большее число из %RDI , %RSI и записываем его в %RSA (после movq %rsi, %rax не надо ret, потому что и так перейдем дальше в ret)

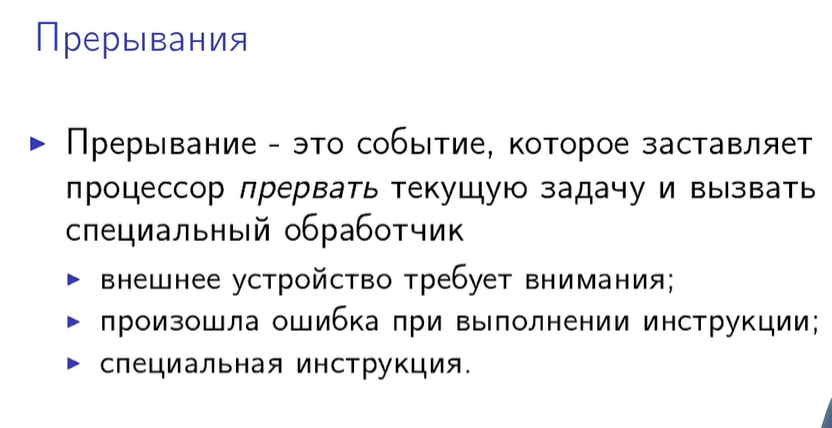


ja – jump if above





Прерывания

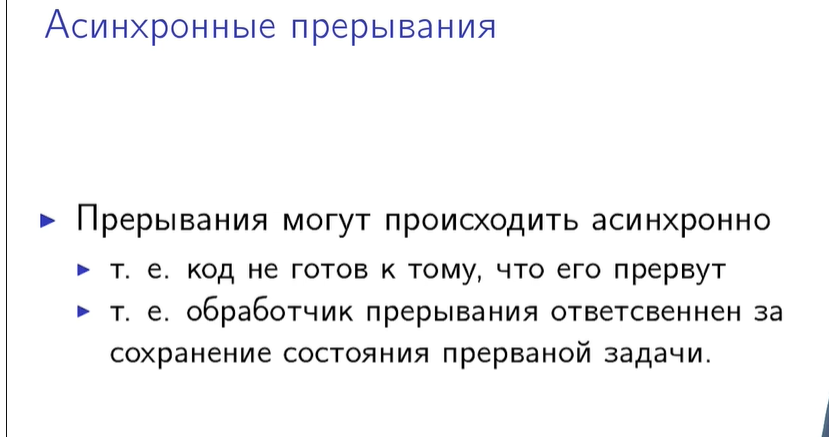


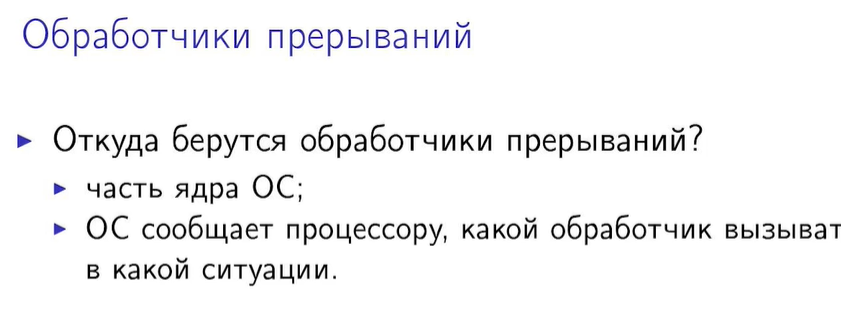
Например сетевая карта получила данные, сообщила процессору, обработчик прерывания обращается к ней и забирает данные, сохраняет их куда-то и возвращает управление прерванной задаче.

2-й пример ошибка деления на 0

3-й специально вызываем

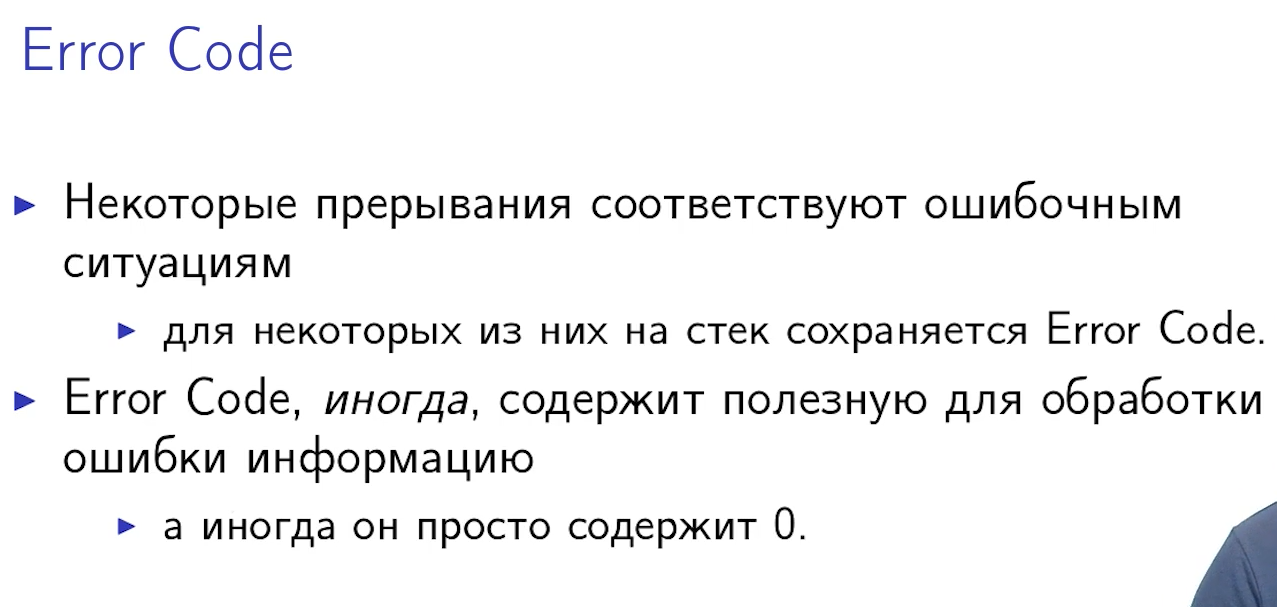
Вообще прерырвания могут происходить асинхронно к коду который они прерывают, это опасно





2 варианта обработчика прерываний

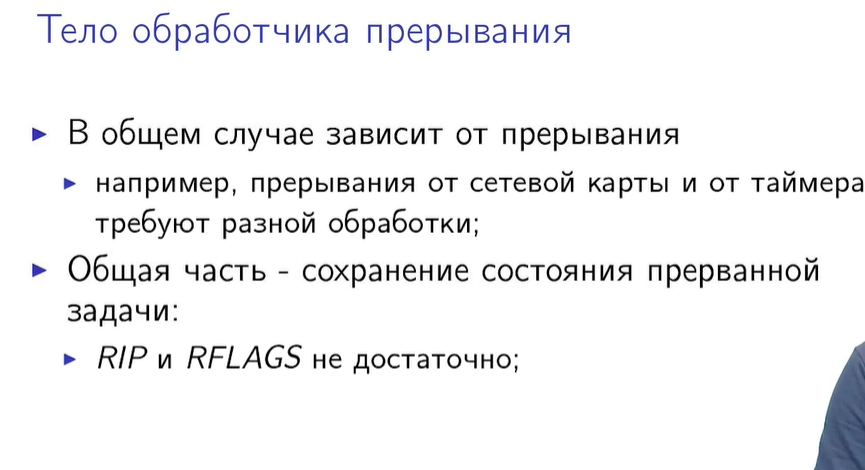


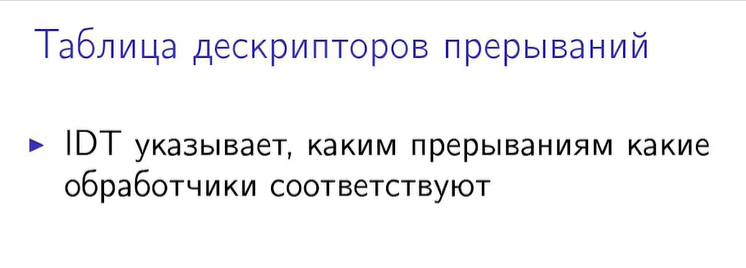




Перед вызовом iretq нужно удалить errir code если он соохранялся

Также если обработчик что-то прервал, а это что-то вычисляло например что-то, то он должен сохранить это (сохранитьь все регистры общего назначения)





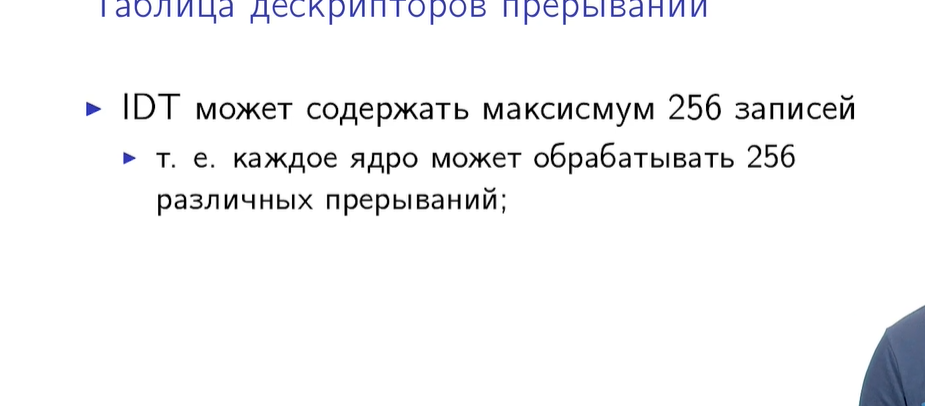
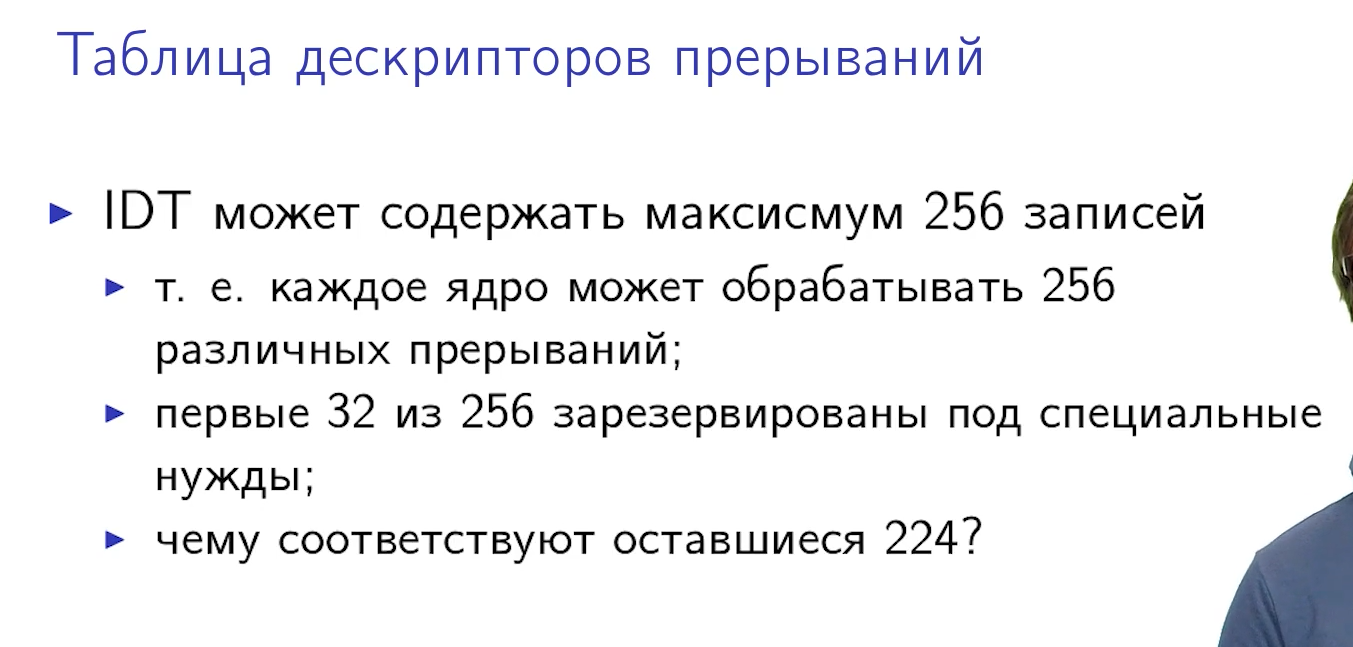
idt – interrupt descriptor table

Сообщает процессору какой обработчик когда вызывать

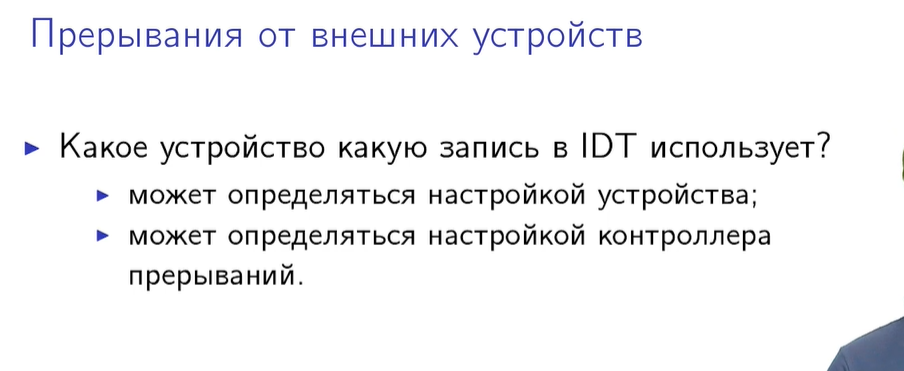
Каждая запись в дескрипторе состоит из 16-ти байт



Поля Offset – отвечают какой обработчик когда вызвать

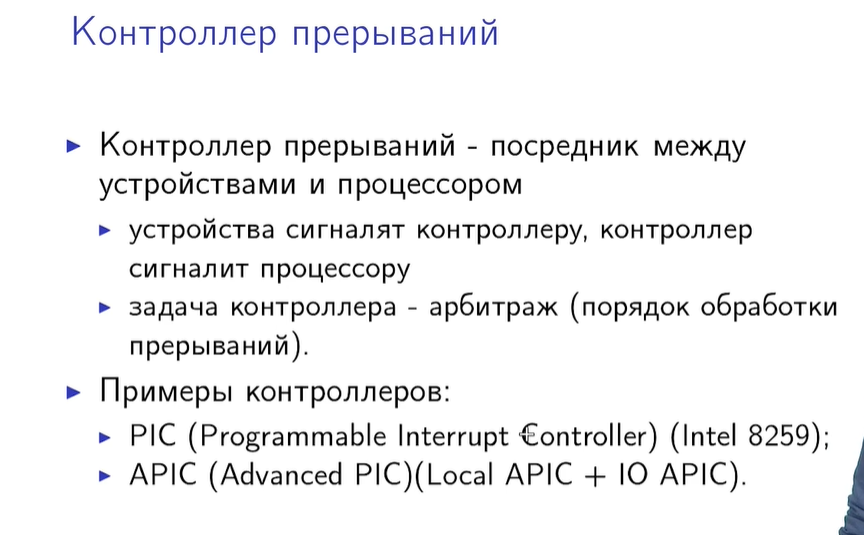
Каждое ядро содержит свою табл

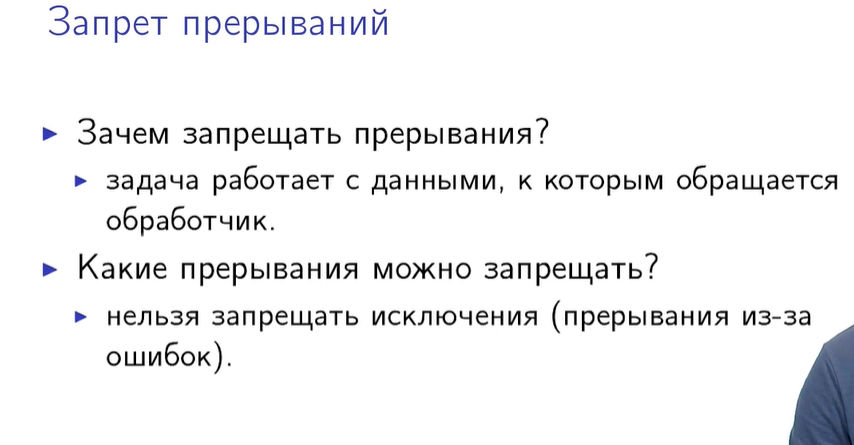
Остальные 224 обычно для обработки прерываний от внешних устройств

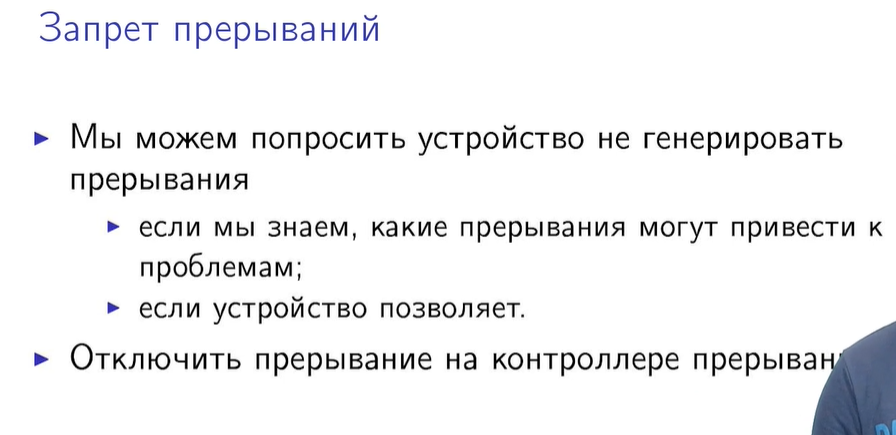


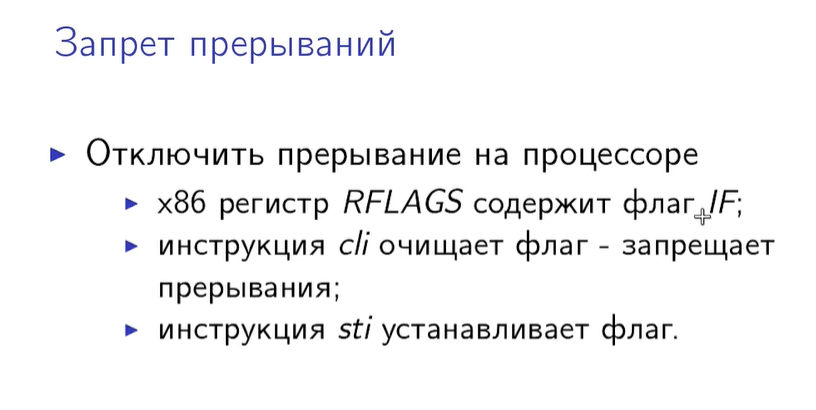
например драйвер при инициализации устройства запрашивает свободный idt номер у процессора и потом когда устройство требует внимания, оно вызывает его

может устройство быть привязано к контроллеру прерываний





3 варианта запрещения прерываний



(clear / set)