### Возможные причины переупорядочивания

1. **Оптимизация производительности**:
   * Компилятор может переупорядочить команды, чтобы улучшить использование конвейера процессора и уменьшить задержки.
2. **Минимизация зависимостей**:
   * Переупорядочивание команд может помочь минимизировать зависимости между инструкциями, что может улучшить параллелизм на уровне команд.
3. **Упрощение логики**:
   * В некоторых случаях переупорядочивание команд может упростить логику кода, что может улучшить читаемость и поддерживаемость.

**Раскрутка циклов** — это метод оптимизации, при котором несколько итераций цикла выполняются в одной итерации. Это может улучшить производительность за счет уменьшения количества переходов и увеличения параллелизма на уровне команд.

**1. КРАТКОЕ ОПИСАНИЕ АРХИТЕКТУРЫ х86 и x86-64**

Архитектура x86 в настоящий момент является самой распространенной программной архитектурой настольных и серверных вычислительных систем. Она используется с 1978 года, когда был выпущен 16-разрядный процессор i8086. Впоследствии архитектура была естественным образом расширена до 32-битной (IA-32), а затем и до 64- битной (x64, AMD64, x86-64, IA-32e, EM64T, Intel 64). При каждом очередном расширении сохранялась программная совместимость с предыдущим поколением архитектуры. При этом набор команд расширялся, а новые регистры большего размера включали в качестве своих частей регистры предыдущего поколения архитектуры.

**x86** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Intel 80x86*) — [архитектура](https://ru.wikipedia.org/w/index.php?title=%D0%90%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80%D0%B0&action=edit&redlink=1) [процессора](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80) и одноимённый [набор команд](https://ru.wikipedia.org/wiki/%D0%90%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%BD%D0%B0%D0%B1%D0%BE%D1%80%D0%B0_%D0%BA%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4), впервые реализованные в процессорах компании [Intel](https://ru.wikipedia.org/wiki/Intel" \o "Intel).

Название образовано от двух цифр, которыми заканчивались названия процессоров Intel ранних моделей — [8086](https://ru.wikipedia.org/wiki/8086), [80186](https://ru.wikipedia.org/wiki/80186), [80286](https://ru.wikipedia.org/wiki/80286) (i286), [80386](https://ru.wikipedia.org/wiki/80386) (i386), [80486](https://ru.wikipedia.org/wiki/80486) (i486).

x86 — это [CISC](https://ru.wikipedia.org/wiki/CISC)-архитектура. **CISC** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *complex instruction set computing* или *complex instruction set computer*) — тип [процессорной архитектуры](https://ru.wikipedia.org/w/index.php?title=%D0%90%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80%D0%B0&action=edit&redlink=1), которая характеризуется следующим набором свойств:

* длина команды произвольна (в отличие от RISC архитектуры, в которой длина команды зафиксирована, например, 32 бита);
* арифметические действия кодируются в одной команде;
* небольшое число [регистров](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B3%D0%B8%D1%81%D1%82%D1%80_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80%D0%B0), каждый из которых выполняет строго определённую функцию.

Методика построения системы команд CISC противостоит методике, применяемой в другом распространённом типе процессорных архитектур — [RISC](https://ru.wikipedia.org/wiki/RISC), где используется набор упрощённых инструкций.

Типичными представителями CISC-архитектуры являются процессоры на основе команд [x86](https://ru.wikipedia.org/wiki/X86), процессоры [Motorola](https://ru.wikipedia.org/wiki/Motorola" \o "Motorola) [MC680x0](https://ru.wikipedia.org/wiki/M68k), Zilog Z80, процессоры [мейнфреймов](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D0%B9%D0%BD%D1%84%D1%80%D0%B5%D0%B9%D0%BC" \o "Мейнфрейм) [zSeries](https://ru.wikipedia.org/wiki/ZSeries" \o "ZSeries).

Архитектура x86 является наиболее распространенной в настольных и мобильных компьютерах. Обычно процессоры соответствующей категории производятся организациями AMD и Intel. Они обладают поддержкой широкого набора инструкций и предусматривают высокий уровень совместимости с программным обеспечением.

Архитектура x86 является наиболее распространенной в настольных и мобильных компьютерах. Обычно процессоры соответствующей категории производятся организациями AMD и Intel. Они обладают поддержкой широкого набора инструкций и предусматривают высокий уровень совместимости с программным обеспечением.

X86 – это ISA для компьютерных процессоров, разработанных в 1978 году. Базируется на основании процессора Intel 8086. Именно отсюда произошло название архитектуры. Изначально x86 была представлена 16-битной системой команд для 16-битных процессоров. Позже она стала поддерживать 32-битные командные системы.

Одним из ключевых ограничений x86 является возможность обработки максимум 4096 Мб памяти. Сегодня x86 обозначает любой 32-разрядный процессор, способный на выполнение системы команд x86.

К преимуществам таких процессоров относят следующие моменты:

1. Широкая поддержка. X86 – это один из наиболее популярных и распространенных типов процессоров. Это значит, что основная масса существующих операционных систем, программ и платформ разработаны с учетом совместимости с этой «моделью». Это делает x86 удобным и доступным широкому кругу пользователей вариантом.
2. Обширная экосистема. У x86 поддерживается обширная экосистема, включая процессоры различных производителей, различные конфигурации и модели, множество производителей материнских плат, оперативной памяти и других составляющих. Все это обеспечивает гибкость в процессе создания систем.
3. Высокий уровень производительности. X86 позволяет добиться эффективной обработки инструкций, а также реализации достаточно сложных вычислительных операций. Технологические улучшения и увеличение количества ядер привело к тому, что x86 могут обрабатывать многопоточные задачи, а также работать с требовательными приложениями и играми.

Архитектура х86 также имеет ряд недостатков:

1. Ограничения совместимости. Некоторые старые приложения и операционные системы могут не поддерживаться на новых системах. При использовании x86 иногда также возникают ограничения в области энергоэффективности и масштабируемости.
2. Сложная структура. Это связано с тем, что x86 обладает сложной иерархией команд и спецификаций. Для программирования этот вариант тоже достаточно трудный. Из-за подобных особенностей x86 может потребовать определенных усилий и времени для досконального изучения. То же самое касается оптимизации программного обеспечения.
3. Ограничения энергоэффективности. X86 обладает большей потребляемой мощностью и может генерировать больше тепла при повышенной нагрузке. Данную особенность предстоит учитывать при выборе системы охлаждения – она должна быть более мощной.
4. Ограничения масштабируемости. X86 предусматривают достаточно высокий уровень производительности, но в плане масштабируемости здесь предусматриваются ограничения. В крупных вычислительных системах, где требуются такие операции, как синхронизация и распараллеливание большого количества процессоров, лучше устанавливать другие процессоры. Пример – RISC.

Процессоры x64 (сокращение от x86-64) – это архитектура системы команд, поддерживающая 64-битный код. Именно этот вариант послужил основой разработки x86. Впервые x64 была запущена в 2000 году. Тогда процессоры поддерживали два режима функционирования:

* 64-битный режим;
* режим совместимости, позволяющий запускать 16-битные и 32-битные программы.

X64 поддерживает все команды, что и x86. Это приводит к тому, что старые исполняемые файлы (приложения) функционируют на устройствах практически без снижения реального уровня производительности.

X64 может использовать 16 миллиардов гигабайт памяти. За счет этой особенности можно задействовать соответствующие процессоры в суперкомпьютерах и машинах, требующих доступ к большому объему ресурсов.

## Ключевая разница между архитектурами x86 и x64 заключается в количестве данных, которые могут быть ими обработаны за каждый тактовый цикл. Сюда также можно отнести ширину процессорного регистра. Функции и ограничения x86.

У x86 предусматриваются следующие функции и особенности:

1. Использование сложной архитектуры со сложным набором команд. А именно – CISC-архитектуры.
2. X86 поддерживает больше доступных регистров, но меньше памяти.
3. Процессоры соответствующего типа разработаны с меньшим количеством конвейеров обработки запросов. Эта особенность перекрывается возможностью обработки сложных адресов.
4. Оптимизация производительности системы достигается за счет аппаратного подхода. X86 задействует физические компоненты памяти для компенсации ее нехватки.
5. Использование программной технологии DEP (Data Execution Prevention).

Ограничения здесь следующие:

1. X86 architecture обладает ограниченным пулом адресуемой памяти.
2. Скорость обработки здесь ниже, чем у x64.
3. Приложения и игры для 32-битных операционных систем больше не разрабатываются. Это приводит к проблемам совместимости.
4. Все устройства в системе совместно пользуются доступной оперативной памятью, оставляя все меньше «пространства» для операционной системы и программного обеспечения.

**Соглашения о вызовах**:

* 1. В архитектуре x86-64 соглашения о вызовах (например, System V AMD64 ABI) определяют, что вещественные числа (типа double) передаются через регистры xmm0-xmm7.
  2. В версии -O3 компилятор может предполагать, что второй аргумент функции уже находится в регистре xmm0, так как это соответствует соглашениям о вызовах.
  3. В версии -O1 компилятор может быть более консервативным и сохранять второй аргумент в стеке, чтобы избежать потенциальных проблем с сохранением состояния регистров.

**Про стек**

В верхней части user mode space расположен стековый сегмент. Большинство языков программирования используют его для хранения локальных переменных и аргументов, переданных в функцию. Вызов функции или метода приводит к помещению в стек т.н. стекового фрейма. Когда функция возвращает управление, стековый фрейм уничтожается. Стек устроен достаточно просто — данные обрабатываются в соответствии с принципом [«последним пришёл — первым обслужен» (LIFO)](http://en.wikipedia.org/wiki/Lifo). По этой причине, для отслеживания содержания стека не нужно сложных управляющих структур – достаточно всего лишь указателя на верхушку стека. Добавление данных в стек и их удаление – быстрая и четко определенная операция. Более того, многократное использование одних и тех же областей стекового сегмента приводит к тому, что они, как правило, находятся в [кеше процессора](http://duartes.org/gustavo/blog/post/intel-cpu-caches" \t "_blank), что еще более ускоряет доступ. Каждый тред в рамках процесса работает с собственным стеком.

Сте́ковый кадр (от [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *stack frame*) — механизм передачи аргументов и выделения временной памяти (в процедурах языков программирования высокого уровня) с использованием [системного стека](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B5%D0%BA_%D0%B2%D1%8B%D0%B7%D0%BE%D0%B2%D0%BE%D0%B2).

Обычно системный стек используется для сохранения адресов возврата при вызове подпрограмм, а также сохранения/восстановления значений регистров процессора.

### Передача аргументов

[[править](https://ru.wikipedia.org/w/index.php?title=%D0%A1%D1%82%D0%B5%D0%BA%D0%BE%D0%B2%D1%8B%D0%B9_%D0%BA%D0%B0%D0%B4%D1%80&veaction=edit&section=2) | [править код](https://ru.wikipedia.org/w/index.php?title=%D0%A1%D1%82%D0%B5%D0%BA%D0%BE%D0%B2%D1%8B%D0%B9_%D0%BA%D0%B0%D0%B4%D1%80&action=edit&section=2)]

При вызове процедуры аргументы отправляются в стек, и только потом производится вызов подпрограммы. Таким образом, процедура получает стек, на вершине которого лежит адрес возврата, а под ним — аргументы, с которыми она была вызвана.

При возвращении из процедуры (или после него, см. ниже) аргументы должны быть сняты со стека.

### Выделение временной памяти

[[править](https://ru.wikipedia.org/w/index.php?title=%D0%A1%D1%82%D0%B5%D0%BA%D0%BE%D0%B2%D1%8B%D0%B9_%D0%BA%D0%B0%D0%B4%D1%80&veaction=edit&section=3) | [править код](https://ru.wikipedia.org/w/index.php?title=%D0%A1%D1%82%D0%B5%D0%BA%D0%BE%D0%B2%D1%8B%D0%B9_%D0%BA%D0%B0%D0%B4%D1%80&action=edit&section=3)]

Если указатель стека сместить «выше» (в сторону увеличения стека), то часть памяти в стеке окажется незадействованной (в том числе и при вызове третьей процедуры) и может использоваться процедурой по своему усмотрению, вплоть до момента возврата в вызвавшую её процедуру. Таким образом, языки высокого уровня организуют переменные, существующие только внутри процедуры (язык Си называет их «автоматическими»).

Перед возвратом процедура должна вернуть указатель стека в оригинальное положение (то есть на адрес возврата).

**Регистры**

**Регистры архитектуры x86**

Регистры - специальные ячейки памяти, расположенные физически внутри процессора, доступ к которым осуществляется не по адресам, как к основной памяти, а по именам.

Младшие части регистров имеют в названии букву L (от слова Low), а старшие H (от слова High). Некоторые команды неявно используют определённый регистр, например, CX может выполнять роль счетчика цикла. Индексные регистры предназначены для хранения индексов при работе с массивами.

Архитектуры x86 и x86-64 (также известная как AMD64 или x64) являются расширениями друг друга и имеют много общего, но также есть существенные отличия. Давайте сравним эти архитектуры и выделим характерные отличия в листингах ассемблера.

### Общие черты

1. **Совместимость**:
   * x86-64 сохраняет обратную совместимость с x86, что означает, что программы, написанные для x86, могут работать на процессорах x86-64 без изменений.
2. **Инструкции**:
   * Многие инструкции в x86-64 аналогичны инструкциям в x86, но есть и новые инструкции, специфичные для x86-64.

### Отличия

1. **Регистры**:
   * **x86**: Имеет 8 регистров общего назначения: eax, ebx, ecx, edx, esi, edi, ebp, esp.
   * **x86-64**: Имеет 16 регистров общего назначения: rax, rbx, rcx, rdx, rsi, rdi, rbp, rsp, r8-r15. Регистры eax, ebx, ecx, edx, esi, edi, ebp, esp являются младшими частями соответствующих 64-битных регистров.
2. **Размер регистров**:
   * **x86**: 32-битные регистры (eax, ebx, ecx, edx, и т.д.).
   * **x86-64**: 64-битные регистры (rax, rbx, rcx, rdx, и т.д.).
3. **Соглашения о вызовах**:
   * **x86**: Аргументы функций передаются через стек.
   * **x86-64**: Аргументы функций передаются через регистры (rdi, rsi, rdx, rcx, r8, r9 для первых 6 аргументов, остальные через стек).
4. **Адресация памяти**:
   * **x86**: 32-битная адресация памяти (до 4 ГБ).
   * **x86-64**: 64-битная адресация памяти (теоретически до 16 ЭБ).

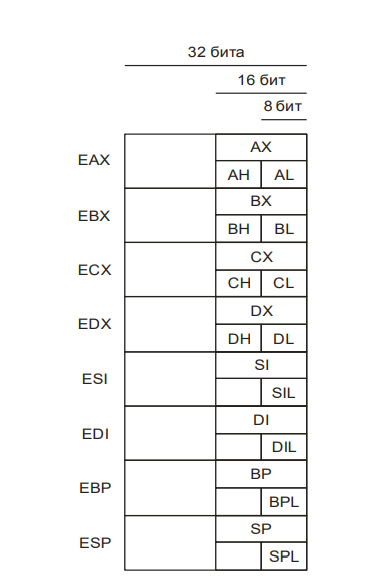
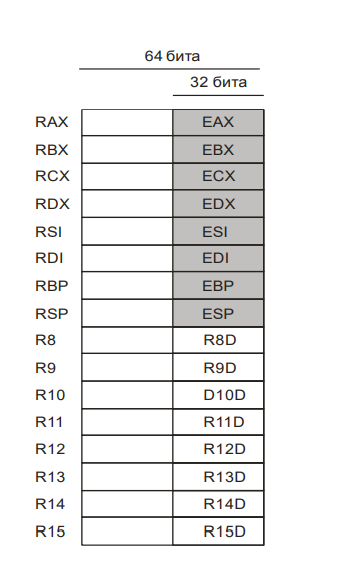
Архитектура x86-64 является расширением архитектуры x86 и добавляет новые регистры, увеличивает размер регистров и адресации памяти, а также изменяет соглашения о вызовах. В листингах ассемблера эти отличия проявляются в использовании 64-битных регистров (rax, rbx, rcx, rdx, и т.д.) и изменении способа передачи аргументов функций.

Локальные переменные лежат по адресам (%ebp - x),

                                    // а параметры функции - по адресам (%ebp + x)  т.к. они уже были в стеке

// до вызова функции.

**1.Регистры общего назначения**

 ****

Используются для хранения произвольных целочисленных данных. При использовании в большинстве команд в качестве операндов эти регистры полностью взаимозаменяемы, но некоторые команды используют только конкретные регистры. Все регистры этой группы позволяют обращаться к своим младшим частям. Для самостоятельной адресации можно использовать только младшие 16 и 8-битные части этих регистров. Старшие 16 бит этих регистров как самостоятельные объекты недоступны. Это сделано, для совместимости с младшими 16-разрядными моделями микропроцессоров фирмы Intel.

Могут использоваться программистами для хранения данных и адресов

RAX(eax, ax, ah, al)- аккумулятор, (для хранения промежуточных данных) , RBX(ebx, bx, bh, bl)-базовый регистер, (часто применяется для хранения базового адреса некоторого объекта в памяти), RCX(ecx, cx, ch, cl)-регистер счетчик, (применяется в командах, производящие некоторые повторяющие действия), RDX(edx, dx, dh, dl) -регистр данных ( хранит промежуточные данные) .

 Перечислим регистры, относящиеся к группе регистров общего назначения.

* **eax/ax/ah/al** (Accumulator register) - *аккумулятор*.

Применяется для хранения промежуточных данных. В некоторых командах использование этого регистра обязательно;

* **ebx/bx/bh/bl** (Base register) - *базовый* регистр.

Часто применяется для хранения базового адреса некоторого объекта в памяти;

* **ecx/cx/ch/cl** (Count register) - *регистр-счетчик*.

Применяется в командах, производящих некоторые повторяющиеся действия. Его использование зачастую неявно и скрыто в алгоритме работы соответствующей команды. К примеру, команда организации цикла loop кроме передачи управления команде, находящейся по некоторому адресу, анализирует и уменьшает на единицу значение регистра *ecx/cx*;

1. **Временное хранение данных**:
   * Регистр ecx может использоваться для временного хранения данных, которые не помещаются в другие регистры.
2. **Аргументы функций**:
   * В некоторых соглашениях о вызовах, таких как cdecl, регистр ecx может использоваться для передачи аргументов функции.

* **edx/dx/dh/dl** (Data register) - регистр *данных*.

Так же, как и регистр *eax/ax/ah/al*, он хранит промежуточные данные. В некоторых командах его использование обязательно; для некоторых команд это происходит неявно.

Следующие два регистра используются для поддержки так называемых строковых операций, то есть операций, производящих последовательную обработку строк элементов, каждый из которых может иметь длину 32, 16 или 8 бит:

* **esi/si** (Source Index register) - *индекс источника*.

Этот регистр в цепочечных операциях содержит текущий адрес элемента в строке-источнике;

* **edi/di** (Destination Index register) - *индекс приемника* (получателя).

Этот регистр в цепочечных операциях содержит текущий адрес в строке-приемнике.

**Регистр EDI** является одним из регистров общего назначения (GPR) в архитектуре x86. Он используется в основном для работы с данными в памяти, особенно в операциях копирования и сравнения.

### Основные характеристики:

* **32-битный регистр:** EDI (Extended Destination Index)
* **64-битный регистр:** RDI (в архитектуре x86-64)
* **16-битный регистр:** DI (младшие 16 бит EDI)

### Основные функции:

1. **Индекс назначения (Destination Index):**
   * Используется в командах копирования и сравнения, таких как MOVS, STOS, CMPS.
   * Указывает на адрес, куда будут записываться данные (в случае MOVS, STOS) или с которым будут сравниваться данные (в случае CMPS).
2. **Передача параметров:**
   * В архитектуре x86-64 RDI используется для передачи первого параметра в подпрограммы.

### Примеры использования:

**1. Копирование данных:**

assembly

Copy code

; Копирование 10 байт из [ESI] в [EDI]

mov ecx, 10

rep movsb

* ESI (Source Index) указывает на источник данных.
* EDI (Destination Index) указывает на место назначения.
* ECX (Count) определяет количество байт для копирования.
* rep movsb выполняет копирование, пока ECX не станет равен нулю.

**2. Заполнение памяти:**

assembly

Copy code

; Заполнение 10 байт по адресу [EDI] значением AL

mov ecx, 10

rep stosb

* AL (младший байт EAX) содержит значение, которым будет заполнена память.
* EDI указывает на начало области памяти.
* rep stosb выполняет заполнение, пока ECX не станет равен нулю.

**3. Передача параметра в подпрограмму:**

assembly

Copy code

; Вызов функции с параметром 0x12345678

mov edi, 0x12345678

call some\_function

* EDI содержит значение параметра.
* call some\_function вызывает подпрограмму, передавая параметр через EDI.

В архитектуре микропроцессора на программно-аппаратном уровне поддерживается такая структура данных, как ***стек***. Стек располагается в оперативной памяти и обычно используется для сохранения адреса возврата из подпрограммы, для передачи параметров в подпрограммы и размещения локальных переменных. Для работы со стеком в системе команд микропроцессора есть специальные команды, а в программной модели микропроцессора для этого существуют специальные регистры:

Для работы со стеком есть специальные регистры RAP(esp, sp)- регистр указателя стека, содержит указатель на вершину стека в текущем сегменте стека. RBP (ebp, bp) -регистр указателя базы кадра стека(для организации произвольного доступа к данным внутри стека, часто этот регистр хранит адрес начала локальных переменных текущей подпрограммы.

**esp/sp** (Stack Pointer register) - регистр *указателя стека*.

Содержит указатель вершины стека в текущем сегменте стека.

* **ebp/bp** (Base Pointer register) - регистр *указателя базы кадра стека*.

Предназначен для организации произвольного доступа к данным внутри стека. Часто регистр bp/ebp хранит адрес начала локальных переменных текущей подпрограммы.

Все регистры общего назначения (кроме esp) могут использоваться при программировании для хранения операндов практически в любых сочетаниях. Но некоторые команды используют фиксированные регистры для выполнения своих действий. Использование жесткого закрепления регистров для некоторых команд позволяет более компактно кодировать их машинное представление. (например MUL использует RAX RDX).)

Эти регистры можно использовать «по частям». Например, к младшим 16 битам регистра %eax можно обратиться как %ax. А %ax, в свою очередь, содержит две однобайтовых половинки, которые могут использоваться как самостоятельные регистры: старший %ah и младший %al. Аналогично можно обращаться к %ebx/%bx/%bh/%bl, %ecx/%cx/%ch/%cl, %edx/%dx/%dh/%dl, %esi/%si, %edi/%di.

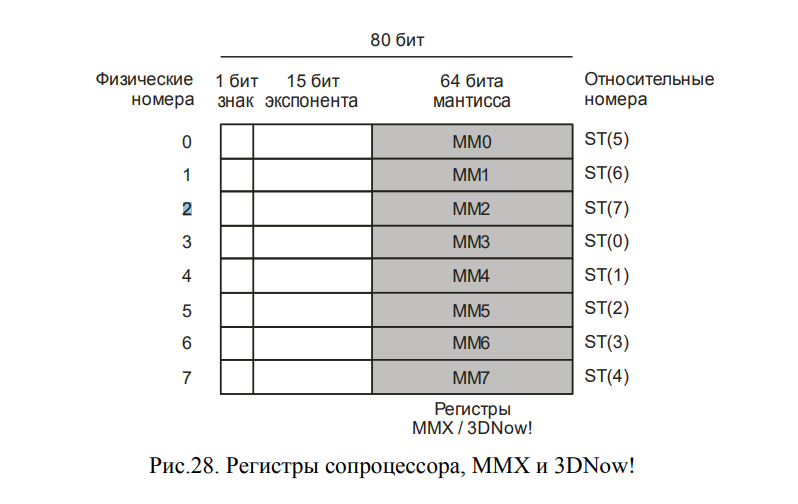
**2. Сопроцессор**

Сопроцессор (FPU – Floating Point Unit) **(специализированный процессор, расширяющий возможности центрального, может быть отдельной микросхемой или встроен в цп)** предназначен для выполнения операций над вещественными числами. С программной точки зрения сопроцессор содержит несколько управляющих регистров, а также блок из 8 регистров данных разрядностью 80 бит, организованных в стек (рис. 2.8). Номер регистра, являющегося текущей вершиной стека, хранится в специальном поле регистра состояния (указателе вершины стека). Операция push уменьшает значение указателя на 1 и помещает данные в регистр, являющийся новой вершиной стека. Операция pop записывает данные с вершины стека в память или регистр и увеличивает указатель на 1. Инструкции сопроцессора адресуют регистры либо явно, либо неявно. Неявная адресация (без указания конкретного регистра) подразумевает использование регистров, находящихся на вершине стека. Явная адресация подразумевает указание смещения регистра относительно вершины стека. [Rbp+смещение]Например: st(0) регистр на вершине стека, st(1) – следующий за ним и т.д.

Выталкивание из стека сопроцессора (FPU stack) — это операция, которая удаляет верхний элемент из стека FPU. В архитектуре x86, FPU (Floating Point Unit) использует стек для хранения и обработки чисел с плавающей точкой. Стек FPU имеет глубину 8 элементов, и каждый элемент может хранить 80-битное число с плавающей точкой.

### Операции со стеком FPU

1. **Помещение в стек (Push)**:
   * Операция, которая помещает значение на вершину стека FPU.
   * Пример: fld (Load Floating Point) — загружает значение в стек FPU.
2. **Выталкивание из стека (Pop)**:
   * Операция, которая удаляет верхний элемент из стека FPU.
   * Пример: fstp (Store Floating Point and Pop) — сохраняет значение из вершины стека FPU в память и удаляет его из стека.

****

╥         регистры расширения XMM

**Сегментные регистры**

В программной модели микропроцессора имеется шесть сегментных регистров: *cs, ss, ds, es, gs, fs*. Их существование обусловлено спецификой организации и использования оперативной памяти микропроцессорами Intel. Она заключается в том, что микропроцессор аппаратно поддерживает структурную организацию программы в виде трех частей, называемых *сегментами*. Соответственно, такая организация памяти называется ***сегментной***.

Для того чтобы указать на сегменты, к которым программа имеет доступ в конкретный момент времени, и предназначены *сегментные регистры*. Фактически, с небольшой поправкой, в этих регистрах содержатся адреса памяти, с которых начинаются соответствующие сегменты. Логика обработки машинной команды построена так, что при выборке команды, доступе к данным программы или к стеку неявно используются адреса во вполне определенных сегментных регистрах. Микропроцессор поддерживает следующие типы сегментов:

1. ***Сегмент кода***. Содержит команды программы.

Для доступа к этому сегменту служит регистр **cs** (code segment register) - *сегментный регистр кода*. Он содержит адрес сегмента с машинными командами, к которому имеет доступ процессор (то есть эти команды загружаются в конвейер микропроцессора).

1. ***Сегмент данных***. Содержит обрабатываемые программой данные.

Для доступа к этому сегменту служит регистр **ds** (data segment register) - *сегментный регистр данных*, который хранит адрес сегмента данных текущей программы.

1. ***Сегмент стека***. Этот сегмент представляет собой область памяти, называемую *стеком*.

Работу со стеком микропроцессор организует по следующему принципу: *последний записанный в эту область элемент выбирается первым*. Для доступа к этому сегменту служит регистр **ss** (stack segment register) - *сегментный регистр стека*, содержащий адрес сегмента стека.

1. ***Дополнительный сегмент данных***.

Большинство машинных команд предполагают, что обрабатываемые ими данные расположены в сегменте данных, адрес которого находится в регистре *ds*.

Если программе недостаточно одного сегмента данных, то она имеет возможность использовать еще три дополнительных сегмента данных. Но в отличие от основного сегмента данных, адрес которого содержится в сегментном регистре *ds*, при использовании дополнительных сегментов данных их адреса требуется указывать явно с помощью специальных *префиксов переопределения сегментов* в команде. Адреса дополнительных сегментов данных должны содержаться в регистрах **es, gs, fs** (extension data segment registers).

**Регистры состояния и управления**

В микропроцессор включены несколько регистров, которые постоянно содержат информацию о состоянии как самого микропроцессора, так и программы, команды которой в данный момент загружены на конвейер. К этим регистрам относятся:

* регистр флагов **eflags/flags**;
* регистр указателя команды **[eip/ip](file:///F:\\new1Bookwww.kolasc.net.rucdoprogrammesassembler%22%20l%20)**.

Используя эти регистры, можно получать информацию о результатах выполнения команд и влиять на состояние самого микропроцессора. Рассмотрим подробнее назначение и содержимое этих регистров:

***eflags/flags*** (flag register) - регистр *флагов*. Разрядность *eflags/flags* - 32/16 бит. Отдельные биты данного регистра имеют определенное функциональное назначение и называются флагами. Младшая часть этого регистра полностью аналогична регистру *flags* для микропроцессора i8086. На рисунке показано содержимое регистра *eflags*.

Исходя из особенностей использования, флаги регистра *eflags/flags* можно разделить на три группы:

* *8 флагов состояния*. Эти флаги могут изменяться после выполнения машинных команд.

*Флаги состояния*регистра *eflags* отражают особенности результата исполнения арифметических или логических операций. Это дает возможность анализировать состояние вычислительного процесса и реагировать на него с помощью условных команд, например, команд условных переходов и вызовов подпрограмм.

* *1 флаг направления*. Обозначается **df** (Directory Flag).

Он находится в 10-м бите регистра *eflags* и используется цепочечными командами. Значение флага *df* определяет направление поэлементной обработки в этих операциях: от начала строки к концу (*df* = 0) либо наоборот, от конца строки к ее началу (*df* = 1);

Для работы с флагом *df* существуют специальные команды: cld (снять флаг *df*) и std (установить флаг *df*).

Применение этих команд позволяет привести флаг *df* в соответствие с алгоритмом и обеспечить автоматическое увеличение или уменьшение счетчиков при выполнении операций со строками;

* *5 системных флагов*, управляющих вводом/выводом, маскируемыми прерываниями, отладкой, переключением между задачами и виртуальным режимом 8086.

Прикладным программам не рекомендуется модифицировать без необходимости эти флаги, так как в большинстве случаев это приведет к прерыванию работы программы.

**Векторные расширения.**

Векторные расширения реализуют технологию SIMD-вычислений.. На рисунке (рис.28) приведены регистры сопроцессора, MMX и 3DNow! Характеристики основных векторных расширений архитектуры x86 / x86-64 представлены в таблице 1.

Расширение mmx:

Было первым расширением, реализущим технологию Simd(принцип компьютерных вычислений, позволяющий обеспечить параллелизм(несколько вычислений одновременно) на уровне данных.) Основная идея simd заключается в одновременной обработке нескольких элементов данных одной операций.

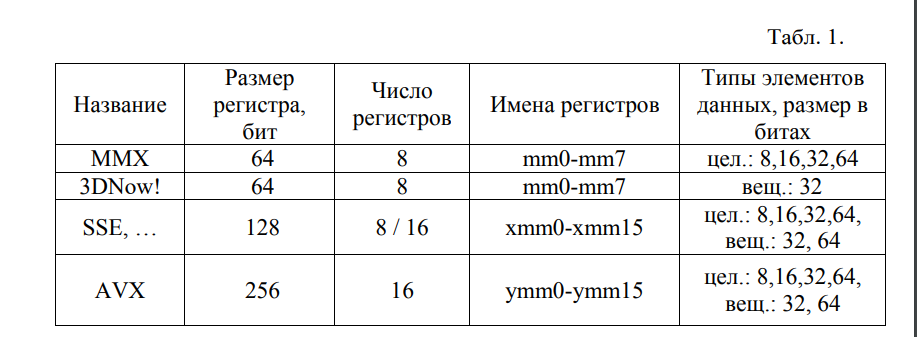
Расширение MMX использует новые типы упакованных 64 битные целочисленных данных:

8 упакованных байт. 4 упакованных слова.2 упакованных двойных слов. 1 учетверенное слово.

Эти типы обрабатываются в 64 битных регистрах MM0-MM7, представляющие собой младшие биты стека 80 битных регистров FPU. Каждая инструкция MMX выполняет действие сразу над всем комплектом операндов(числа), размещенных в адресуемых регистрах. Эти регистры не могут использоваться для адресной памяти.

Расширение 3DNОW! - технология AMD, расширяет возможности MMX. Позволяет оперировать с новыми типами данных - парой упакованных вещественных чисел одинарной точности. Эти числа занимают по двойному слову (32 бита) в 64-битных регистрах MMX .

Это расширение работает с упакованными данными в FP-формате( формат представления числа с плавающей точкой ) с одинарной точностью, а так же упакованными 8 байт, 4 слова, 2 двойных слова и 64 битными целыми числами, размещая их в младших 64 битах регистрах FPU/MMX.

****

Векторные вычисления — такие компьютерные вычисления, когда при выполнении одной инструкции процессора производится не одна операция, а одновременно несколько однотипных операций над несколькими порциями данных. Иначе этот принцип называется SIMD — от английского Single Instruction, Multiple Data. Название возникло из очевидной аналогии с векторной алгеброй: операции между векторами обозначаются одним символом, но подразумевают, что одновременно выполняются несколько арифметических действий над компонентами вектора.

Изначально векторные вычисления выполнялись специализированными сопроцессорами, которые являлись основной частью суперкомпьютеров. В 1990-е годы в некоторых процессорах архитектуры x86 и нескольких других появились векторные расширения: специальные регистры с повышенной разрядностью и специальные векторные инструкции, оперирующие этими регистрами.

Векторные инструкции применяются там, где требуется выполнение множества однотипных операций и высокая производительность вычислений. Это всевозможные задачи вычислительной математики и математического моделирования, компьютерная графика и компьютерные игры. Без векторных вычислений сегодня невозможно достичь такой производительности вычислительной системы, которая требуется для обработки видеосигналов и, в особенности, кодирования и декодирования видео. Необходимо отметить, что для некоторых задач и алгоритмов векторные инструкции не увеличивают производительность.

### ****Инструкции и регистры****

Векторные вычисления — такие вычисления, когда при выполнении одной инструкции процессора производится одновременно несколько однотипных операций. Этот принцип в настоящее время реализован не только в специализированных процессорах, но и в процессорах архитектуры x86 и ARM в виде векторных расширений. Эти расширения представляют собой специальные векторные регистры с повышенной относительно регистров общего назначения разрядностью. Для работы с этими регистрами имеются специальные векторные инструкции, которые дополняют систему инструкций процессора.

Как правило, векторные инструкции реализуют те же операции, что и скалярные (обычные) инструкции, но благодаря большому объёму обрабатываемых данных производительность этих инструкций выше. Если для регистра общего назначения при выполнении некоторой инструкции предполагается, что в нём находится только одна порция данных определённого типа (целое число определённой разрядности, число с плавающей запятой), то в векторном регистре одновременно находится столько независимых порций данных определённого типа, сколько позволяет разместить ёмкость регистра. И такое же количество одновременных независимых операций может быть произведено над этими данными при выполнении векторной инструкции — и во столько же раз повышается производительность вычислений. Повысить производительность процессора, выполняя несколько одинаковых операций одновременно, — основная задача векторных расширений.

В процессорах архитектуры x86 первым векторным расширением был набор инструкций MMX, оперирующих восемью 64-битными регистрами MM0-MM7. MMX сменили более производительные 128-битные инструкции SSE (инструкции для работ числами с плавающей запятой) и SSE2 (целочисленные инструкции и инструкции для работ с числами с плавающей запятой двойной точности), оперирующие регистрами xmm0-xmm7. Позже появились наборы 128-битных инструкций SSE3, SSSE3, SSE4.1 и SSE4.2, которые дополнили SSE и SSE2 несколькими полезными инструкциями. Большинство инструкций из перечисленных наборов используют два регистра-операнда, результат записывается в один из этих регистров, а его первоначальное содержимое теряется.

Следующий шаг в развитии векторных расширений — ещё более производительные 256-битные инструкции AVX и AVX2, которые оперируют 256-битными регистрами ymm0-ymm15. Кроме того, эти инструкции используют три регистра-операнда: исходные данные содержатся в двух регистрах, результат операции записывается в третий регистр, а содержимое двух других регистров остаётся неизменным. Новейший на сегодняшний день набор векторных инструкций — AVX-512, который оперирует 32 512-битными регистрами zmm0-zmm31. AVX-512 используется в некоторых серверных процессорах для высокопроизводительных вычислений.

С массовым появлением 64-битных процессоров инструкции MMX признаны устаревшими. Инструкции SSE и SSE2 с появлением AVX и AVX2 не вышли из употребления и продолжают активно использоваться. В процессорах x86 сохраняется обратная совместимость: если процессором поддерживается AVX2, то им поддерживаются и SSE/SSE2, а также SSE3, SSSE3, SSE4.1 и SSE4.2. Аналогично, процессор с поддержкой, например, SSSE3, поддерживает и все более ранние наборы инструкций.

**Регистры архитектуры x86-64 (AMD64)**

Выпущенные недавно фирмой AMD новые процессоры Athlon64 и Opteron имеют архитектуру x86-64, которая в отличие от архитектуры x86 является полностью 64-битной. Она естественным образом расширяет регистры общего назначения x86 до 64 битов и увеличивает их количество. Также удваивается число регистров XMM. Регистры FPU/MMX остаются без изменений.

В отличие от процессоров x86, у которых все вещественные вычисления производились в сопроцессоре, а блоку XMM отводились только векторные операции, процессоры x86-64 практически все вещественные вычисления выполняют в блоке XMM.

Отличие 86x и 64 у 64 регистр вмещает 64 бита и регистров больше( например xmm стало в 2 раза больше), длина командного слова больше. В отличие от 86x, у которых все вещественные вычисления производились в сопроцессоре, а блоку xmm отводились только встроенные векторные операции, процессоры x86-64 практически все вещественные вычисления выполняются в "блоке xmm" ( новое потоковое расширение SSE , реализующиеся независим блоком, имеющим восемь 128 битных регистров, названных xmm0-xmm7, и регистр состояния/управления MXCSR.В каждый из регистров XMM помещаются четыре числа в формате с плавающей точкой одинарной точности. Блок позволяет выполнять векторные(реализуют операции сразу над четырьмя комплектами операндов) и скалярные инструкции(работают только с одним комплектом операндов -младшим 32- битным словом).

## Операционные суффиксы

Инструкции по сборке GAS обычно дополнены буквами «b», «s», «w», «l», «q» или «t», чтобы определить, какой размер операнда обрабатывается.

* b= байт (8 бит).
* s= одинарный (32-битный с плавающей точкой).
* w= слово (16 бит).
* l= long (32-битное целое число или 64-битное число с плавающей точкой).
* q= четырехъядерный (64 бит).
* t= десять байт (80-бит с плавающей точкой).

Если суффикс не указан и для инструкции нет операндов памяти, GAS выводит размер операнда из размера операнда регистра назначения (конечного операнда).

## Программная архитектура x86/x86-64:

### 1. Набор регистров:

**Регистры общего назначения (GPR):**

* **32-битные:** EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
* **64-битные:** RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8-R15

**Специальные регистры:**

* **EIP/RIP:** Указатель команд (Instruction Pointer)
* **EFLAGS/RFLAGS:** Регистр флагов
* **CS, DS, SS, ES, FS, GS:** Сегментные регистры

**Регистры для работы с числами с плавающей точкой:**

* **80-битные:** ST0-ST7 (в архитектуре x87)

**Регистры для работы с векторными расширениями:**

* **128-битные:** XMM0-XMM15 (в SSE)
* **256-битные:** YMM0-YMM15 (в AVX)
* **512-битные:** ZMM0-ZMM31 (в AVX-512)

### 2. Основные арифметико-логические команды:

* **Арифметические:** ADD, SUB, MUL, DIV, INC, DEC
* **Логические:** AND, OR, XOR, NOT, SHL, SHR, ROL, ROR
* **Сравнения:** CMP, TEST

### 3. Способы адресации памяти:

* **Непосредственная адресация:** mov eax, 10
* **Регистровая адресация:** mov eax, ebx
* **Прямая адресация:** mov eax, [0x12345678]
* **Косвенная адресация:** mov eax, [ebx]
* **Базовая адресация:** mov eax, [ebx+12]
* **Индексная адресация:** mov eax, [ebx+esi\*4]
* **Базовая индексная адресация:** mov eax, [ebx+esi\*4+12]

### 4. Способы передачи управления:

* **Безусловный переход:** JMP
* **Условный переход:** JE, JNE, JG, JL, JGE, JLE и другие
* **Вызов подпрограммы:** CALL
* **Возврат из подпрограммы:** RET

### 5. Работа со стеком:

* **Помещение в стек:** PUSH
* **Извлечение из стека:** POP
* **Указатель стека:** ESP/RSP

### 6. Вызов подпрограмм:

* **Сохранение адреса возврата:** CALL помещает адрес следующей команды в стек.
* **Возврат из подпрограммы:** RET извлекает адрес возврата из стека и передает управление по этому адресу.

### 7. Передача параметров в подпрограммы и возврат результатов:

* **Через стек:** Параметры помещаются в стек перед вызовом подпрограммы.
* **Через регистры:** Параметры передаются через регистры (например, RDI, RSI, RDX, RCX, R8, R9 в x86-64).
* **Возврат результата:** Обычно через регистр EAX/RAX.

### 8. Работа с арифметическим сопроцессором:

* **Загрузка чисел:** FLD, FILD
* **Сохранение чисел:** FST, FIST
* **Арифметические операции:** FADD, FSUB, FMUL, FDIV
* **Тригонометрические функции:** FSIN, FCOS, FSINCOS
* **Сравнение:** FCOM, FCOMP

### 9. Работа с векторными расширениями (SSE, AVX, AVX-512):

* **Загрузка данных:** MOVAPS, MOVUPS, VMOVAPS, VMOVUPS
* **Арифметические операции:** ADDPS, SUBPS, MULPS, DIVPS, VADDPS, VSUBPS, VMULPS, VDIVPS
* **Логические операции:** ANDPS, ORPS, XORPS, VANDPS, VORPS, VXORPS
* **Тригонометрические функции:** SINPS, COSPS, VSINPS, VCOSPS

**LC0**: Метка, указывающая на место в памяти, где хранится 64-битное значение.

**[rip]**: Регистр RIP (Instruction Pointer) используется для относительной адресации. Он указывает на текущую команду, и адрес вычисляется относительно этой команды.

Указатель на текущую команду. В данном случае, [rip] используется для относительной адресации, что позволяет компилятору размещать данные и код в произвольных местах памяти.

Метка .LC0[rip] используется для обращения к данным в памяти с использованием относительной адресации. Это позволяет компилятору размещать данные и код в произвольных местах памяти, что упрощает создание переносимых программ.

* **RBP (Base Pointer)**: Используется для хранения базового адреса текущего кадра стека.
* **RSP (Stack Pointer)**: Указывает на вершину стека.
* **[rbp-20]**: Адрес памяти, вычисленный как RBP - 20. Этот адрес указывает на область стека, выделенную для локальных переменных.
* Команда mov в ассемблере x86-64 используется для перемещения данных между регистрами и памятью. Когда вы записываете данные в память, адрес, по которому записываются данные, определяется способом адресации.

### Запись в стек:

* **Стек** — это область памяти, которая используется для хранения локальных переменных, аргументов функций, адресов возврата и других данных, связанных с вызовом функций.

main:

push rbp

mov rbp, rsp

sub rsp, 32 ; Выделяем 32 байта в стеке

; Предположим, что edi содержит значение 0x12345678

mov DWORD PTR [rbp-20], edi

; Далее можно использовать значение по адресу [rbp-20]

leave

ret

**Объяснение:**

* **push rbp**: Сохраняет текущее значение RBP в стеке.
* **mov rbp, rsp**: Устанавливает RBP на текущее значение RSP, что создает новый кадр стека для функции.
* **sub rsp, 32**: Выделяет 32 байта в стеке для локальных переменных.
* **mov DWORD PTR [rbp-20], edi**: Записывает 32-битное значение из регистра EDI в память по адресу [rbp-20], который находится в стеке.

**Команды x86-64**

|  |  |
| --- | --- |
| Команда, метки | Применение |
| push | Помещение операнда в стек |
| mov | Копирование второго операнда в первый |
| movsd | Пересылка двойных слов |
| **.LC0** | Метка, указывающая на место в памяти, где хранятся данные. |
| .L3 и т.д. | используются для организации условных переходов и управления потоком выполнения программы. Они позволяют разделить код на логические блоки, что упрощает чтение и понимание программы. |
| jmp | Переход безусловный. Она передает управление по указанной метке, что позволяет изменять поток выполнения программы. |
| pxor | Команда для выполнения побитовой операции XOR между двумя регистрами |
| cvtsi2sd | Преобразует целое число в вещественное |
| mulsd | Умножение двух операндов |
| add | Сложение двух операндов |
| cmp | Сравнение двух операндов  Она вычитает второй операнд из первого и устанавливает флаги в регистре флагов (EFLAGS) в зависимости от результата вычитания. Результат вычитания не сохраняется, а только устанавливаются флаги, которые могут быть использованы для условных переходов.  **Регистр флагов (EFLAGS)**: Содержит флаги, которые устанавливаются в зависимости от результата операции. Важные флаги для сравнения:  **ZF (Zero Flag)**: Устанавливается в 1, если результат вычитания равен нулю (значения равны).  **SF (Sign Flag)**: Устанавливается в 1, если результат вычитания отрицателен (первый операнд меньше второго).  **OF (Overflow Flag)**: Устанавливается в 1, если произошло переполнение при вычитании.  **CF (Carry Flag)**: Устанавливается в 1, если произошел заем при вычитании (первый операнд меньше второго). |
| jle | Используется для условного перехода в программе. Она передает управление по указанной метке, если результат предыдущего сравнения удовлетворяет условию "меньше или равно". |
| jne | перейти, если не нуль или не равно. Условие перехода: ZF = 0 |
| pop | используется для извлечения значения из стека и записи его в указанный регистр или память. Она уменьшает указатель стека (RSP) на размер операнда, тем самым удаляя значение из стека.  **pop eax**: Извлекает значение из стека и записывает его в регистр EAX. Указатель стека (RSP) увеличивается на 4 байта. |
| ret | Используется для возврата управления из подпрограммы (функции) в вызывающую программу. Она извлекает адрес возврата из стека и передает управление по этому адресу. |
| sub | Команда sub rsp используется для вычитания указанного значения из регистра RSP (Stack Pointer). Это действие уменьшает значение указателя стека, тем самым выделяя место в стеке для локальных переменных и других данных. |
| movapd | используется для перемещения 128-битных данных (двойных слов с плавающей точкой) между регистрами XMM и памятью. |
| movq | Команда movq используется для перемещения 64-битных данных между регистрами XMM и GPR(регистры общего назначения), а также с памятью. Она позволяет передавать данные между различными типами регистров и памятью, что полезно для работы с числами с плавающей точкой и векторными данными в ассемблере x86-64. |
| divsd | Используется для деления двух 64-битных чисел с плавающей точкой (двойной точности) в регистрах XMM. Она делит младшее 64-битное значение первого операнда на младшее 64-битное значение второго операнда и сохраняет результат в первом операнде. |
| lea | Используется для вычисления эффективного адреса и записи его в указанный регистр. В отличие от команды mov, которая перемещает данные, lea вычисляет адрес, но не обращается к памяти для получения данных.**Вычисление адреса**: Команда lea вычисляет адрес, но не обращается к памяти для получения данных. Это полезно для выполнения арифметических операций с регистрами.  **Регистры**: Команда lea может работать с регистрами общего назначения (GPR), такими как RAX, RBX, RCX, RDX, и с регистрами XMM для работы с векторными данными. |
| test | Логическое И. Используется для выполнения побитовой операции И между двумя операндами и установки флагов в регистре флагов (EFLAGS) в зависимости от результата операции. Она не сохраняет результат операции, а только устанавливает флаги, которые могут быть использованы для условных переходов. **ZF (Zero Flag)**: Устанавливается в 1, если результат операции равен нулю. |
| call | Вызов процедуры. Передача управления на подпрограмму с запоминанием в стеке адреса точки возврата |
| addsd | Используется для сложения двух 64-битных чисел с плавающей точкой (двойной точности) в регистрах XMM. Она добавляет младшее 64-битное значение из второго операнда к младшему 64-битному значению первого операнда и сохраняет результат в первом операнде. |
| leave | Восстанавливает предыдущий кадр стека |

**Команды x86**

|  |  |
| --- | --- |
| Команда, метки | Применение |
| fld1 | Загрузка константы вещественная единица в вершину стека сопроцессора |
| fstp | Сохранение вещ.значения из стека с последующим выталкиванием |
| fild | Загрузка целого числа, преобразование в вещ.(для того, чтобы в дальнейшем могли работать команды сопроцессора;исходное целочисленное значение только адрес ячеки памяти размером 16, 32 или 64 бита) и загрузка в вершину стека |
| fld | Загрузка вещ.значения из ячейки памяти в вершину стека. |
| fmulp | Умножение с выталкиванием из вершины стека(двух вещ.значений) |
| fdivr | Деление двух вещ.чисел в обратном порядке(делимое вершина стека, делитель в st(1)), результат в st(1)- без операндов. С одним операндом: делимое в st(1), делитель в st(0) |
| faddp | Сложение с выталкиванием из стека(два вещ.) |
| and | Логическое И. Если операнды равны 1, в остальных случаях 0. |
| fdivp | Деление двух вещ.чисел с последующим выталкиванием из стека. Делимое в st(1), делитель в st(0) – 0 или 1 операнд |
| xor | Логическое исключающее ИЛИ. 1, если значения различны, в остальных 0. |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**Инфа к листингам**

В ассемблере x86-64 регистр RDI (и его 32-битная версия EDI) используется для передачи первого аргумента функции. В контексте функции result, регистр RDI содержит значение аргумента N.

* **Базовый адрес**: RBX часто используется для хранения базовых адресов при доступе к данным в памяти.
* **Индексация**: RBX может использоваться как индекс при доступе к элементам массива.
* **Сохранение значений**: RBX может использоваться для временного хранения значений, которые не должны быть изменены во время выполнения функции.
* **Базовый указатель (RBP)**: Регистр, который используется для хранения базового адреса текущего кадра стека. Восстановление предыдущего значения RBP позволяет вернуться к предыдущему кадру стека.
* **Базовый указатель (RBP)**: Регистр, который используется дляя хранения базового адреса текущего кадра стека. Он помогает организовать доступ к локальным переменным и аргументам функций.

ret

### Пример использования:

Предположим, у нас есть следующий код на ассемблере:

assembly

Copy code

main:

; Вызов функции example

call example

; Далее выполняется код после вызова функции

ret

example:

; Тело функции example

; ...

; Возврат из функции

ret

### Объяснение:

1. **call example**: Вызывает функцию example. Адрес следующей команды после call помещается в стек (адрес возврата).
2. **example:**: Метка, обозначающая начало функции example.
3. **ret**: Возвращает управление из функции example. Извлекает адрес возврата из стека и передает управление по этому адресу.
4. **ret**: Возвращает управление из функции main.

* .string — это директива, которая указывает ассемблеру сохранить строку в памяти. Строка указывается внутри кавычек "".
* **Инструкция «add rsp, 8»** **используется для восстановления пространства в стеке**.
* Если перед этим было зарезервировано 8 байт пространства в стеке с помощью команды «sub rsp, 8», то для освобождения этого пространства нужно выполнить «add rsp, 8» перед возвратом из функции.

Предположим, что у нас есть функция, которая принимает аргументы через стек. В x86 архитектуре аргументы функции передаются через стек, и они располагаются в памяти относительно указателя базы (ebp).

### Инструкции

В архитектуре x86 (и x86-64) данные хранятся в памяти в **порядке "от младшего к старшему" (little-endian)**. Это означает, что младшие байты данных хранятся по младшим адресам, а старшие байты - по старшим адресам.

1. **mov eax, DWORD PTR [ebp+12]**
   * **Загружает младшее двойное слово аргумента в eax.**
   * **Объяснение:**
     + DWORD PTR [ebp+12] указывает на 4 байта (двойное слово) в памяти, расположенные на 12 байт выше текущего значения ebp.
     + mov eax, ... копирует значение из этого адреса в регистр eax.
     + Это значение является младшим двойным словом аргумента, переданного функции.
2. **mov edx, DWORD PTR [ebp+16]**
   * **Загружает старшее двойное слово аргумента в edx.**
   * **Объяснение:**
     + DWORD PTR [ebp+16] указывает на 4 байта (двойное слово) в памяти, расположенные на 16 байт выше текущего значения ebp.
     + mov edx, ... копирует значение из этого адреса в регистр edx.
     + Это значение является старшим двойным словом аргумента, переданного функции.
3. **mov DWORD PTR [ebp-32], eax**
   * **Сохраняет младшее двойное слово в [ebp-32].**
   * **Объяснение:**
     + DWORD PTR [ebp-32] указывает на 4 байта (двойное слово) в памяти, расположенные на 32 байта ниже текущего значения ebp.
     + mov ... , eax копирует значение из регистра eax в этот адрес.
     + Это сохраняет младшее двойное слово аргумента в локальную переменную.
4. **mov DWORD PTR [ebp-28], edx**
   * **Сохраняет старшее двойное слово в [ebp-28].**
   * **Объяснение:**
     + DWORD PTR [ebp-28] указывает на 4 байта (двойное слово) в памяти, расположенные на 28 байт ниже текущего значения ebp.
     + mov ... , edx копирует значение из регистра edx в этот адрес.
     + Это сохраняет старшее двойное слово аргумента в локальную переменную.

### Пример

Предположим, что функция принимает аргумент типа double (8 байт), и этот аргумент передается через стек. В x86 архитектура младшие 4 байта аргумента будут расположены по адресу [ebp+12], а старшие 4 байта — по адресу [ebp+16].

assembly

Copy code

push ebp

mov ebp, esp

sub esp, 32 ; Выделяем место для локальных переменных

; Аргументы функции:

; [ebp+8] - первый аргумент (если есть)

; [ebp+12] - младшее двойное слово аргумента типа double

; [ebp+16] - старшее двойное слово аргумента типа double

mov eax, DWORD PTR [ebp+12] ; Загружаем младшее двойное слово в eax

mov edx, DWORD PTR [ebp+16] ; Загружаем старшее двойное слово в edx

mov DWORD PTR [ebp-32], eax ; Сохраняем младшее двойное слово в [ebp-32]

mov DWORD PTR [ebp-28], edx ; Сохраняем старшее двойное слово в [ebp-28]

; Далее код функции

leave

ret

### Результат

* Младшее двойное слово аргумента типа double сохраняется в локальную переменную по адресу [ebp-32].
* Старшее двойное слово аргумента типа double сохраняется в локальную переменную по адресу [ebp-28].

### Что происходит при выполнении команды:

1. **Вычисление адреса**: Сначала вычисляется адрес памяти, который равен значению указателя стека (esp) плюс 36.
2. **Чтение значения**: Затем из памяти по этому адресу читается 32-битное значение.
3. **Помещение на стек**: Это значение помещается на вершину стека. Указатель стека (esp) уменьшается на 4, чтобы указывать на новую вершину стека.

В ассемблерном коде, когда функция вызывается, аргументы функции передаются через стек. Соглашения о вызовах (calling conventions) определяют, как аргументы передаются и как они извлекаются из стека внутри функции. В архитектуре x86, соглашение о вызовах System V AMD64 ABI определяет, что аргументы функции передаются через стек в обратном порядке, то есть первый аргумент функции находится по адресу ebp+8, второй аргумент — по адресу ebp+12, и так далее

Команда lea (Load Effective Address) в ассемблере используется для вычисления эффективного адреса и сохранения его в указанном регистре. В данном случае, команда lea eax, [ebx-1] вычисляет адрес, который равен значению регистра ebx минус 1, и сохраняет результат в регистр eax.

### lea

* **lea** — это команда, которая вычисляет эффективный адрес операнда и сохраняет его в указанном регистре. Она не обращается к памяти, а просто выполняет арифметические операции над регистрами.

### eax

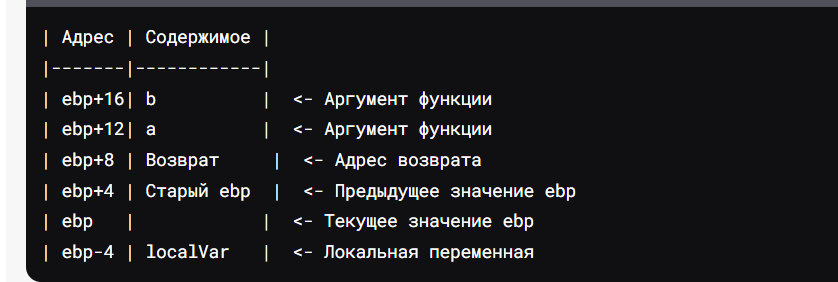
* **eax** — это один из регистров общего назначения в архитектуре x86. В 32-битной архитектуре он имеет размер 32 бита (4 байта).

### [ebp+12]:

* **Адрес:** Базовый адрес кадра стека (ebp) **плюс** 12 байт.
* **Значение:** Данные, расположенные на 12 байт **выше** текущего значения ebp.
* **Обычно используется для:** Доступа к аргументам функции, переданным через стек.

### [ebp-12]:

* **Адрес:** Базовый адрес кадра стека (ebp) **минус** 12 байт.
* **Значение:** Данные, расположенные на 12 байт **ниже** текущего значения ebp.
* **Обычно используется для:** Доступа к локальным переменным функции, которые хранятся в стеке.



rip — это регистр счетчика команд (Instruction Pointer) в архитектуре x86-64. Он указывает на адрес следующей инструкции, которая должна быть выполнена.

В ассемблере rip часто используется вместе с относительной адресацией для доступа к данным, которые находятся в статической памяти (например, глобальные переменные или константы).

В ассемблере x86-64, xmm0 — это один из регистров семейства XMM, которые используются для работы с данными в формате SSE (Streaming SIMD Extensions). XMM-регистры имеют размер 128 бит и могут хранить различные типы данных, такие как числа с плавающей точкой одинарной точности (32 бита), числа с плавающей точкой двойной точности (64 бита), а также целые числа и другие типы данных, которые могут быть обработаны параллельно (SIMD).

В ассемблере ret (сокращение от "return") — это инструкция, которая используется для возврата управления из подпрограммы (функции) в точку вызова. Она завершает выполнение текущей функции и возвращает управление вызывающей функции.

### Основные характеристики ret:

1. **Возврат управления**: Инструкция ret извлекает адрес возврата из стека и передает управление по этому адресу.
2. **Использование**: Обычно используется в конце функции, чтобы вернуть управление вызывающей функции.

**Адрес возврата**: Перед вызовом функции, вызывающая функция обычно помещает адрес возврата в стек. Инструкция ret извлекает этот адрес и передает управление по нему.

Загружает 64-битное значение из памяти по адресу [rbp-8]

QWORD PTR — это указатель на 64-битное (8-байтовое) значение в памяти. QWORD означает "Quad Word", что в контексте архитектуры x86-64 соответствует 64 битам.

DWORD PTR — это указатель на 32-битное (4-байтовое) значение в памяти. DWORD означает "Double Word", что в контексте архитектуры x86 соответствует 32 битам.

Это означает, что вы можете иметь оба типа значений реестра как в 32-битных, так и в 64-битных операционных системах. В этом контексте «слово» означает 16 бит. DWORD, таким образом, означает «двойное слово» или 32 бита (16 X 2). Следуя этой логике, QWORD означает «четверное слово» или 64 бита (16 X 4

Команда **OFFSET FLAT:.**LC2 используется в ассемблерном коде для получения смещения (адреса) метки .LC2 в плоской (flat) модели памяти. Давайте разберем это подробнее.

### Описание:

1. **OFFSET**:
   * OFFSET — это оператор, который используется для получения адреса (смещения) метки или переменной относительно начала сегмента памяти.
2. **FLAT**:
   * FLAT — это модель памяти, которая используется в 32-битных и 64-битных системах, где вся память рассматривается как один непрерывный блок. В плоской модели памяти нет сегментации, и все адреса являются абсолютными.
3. **.LC2**:
   * .LC2 — это метка, которая указывает на определенное место в памяти, где хранится строка или другой тип данных.

### Пример использования:

Предположим, у нас есть следующий ассемблерный код:

assembly

Copy code

.LC2:

.string "Res: %lf"

main:

; Другие инструкции

; ...

; Получаем смещение метки .LC2

mov eax, OFFSET FLAT:.LC2

; Теперь в регистре eax находится адрес строки "Res: %lf"

; Другие инструкции

; ...

### Что происходит:

* **mov eax, OFFSET FLAT:.LC2**:
  + Эта инструкция помещает адрес метки .LC2 в регистр eax. В плоской модели памяти этот адрес является абсолютным, то есть он указывает на конкретное место в памяти, где начинается строка "Res: %lf".

### Зачем это нужно:

* **Передача адреса строки в функцию**:
  + Часто адрес строки нужно передать в функцию, например, в printf для вывода строки на экран. В этом случае адрес строки должен быть известен, и оператор OFFSET FLAT помогает получить этот адрес.

### Пример использования в контексте вызова printf:

assembly

Copy code

extern printf

.LC2:

.string "Res: %lf"

main:

; Другие инструкции

; ...

; Получаем смещение метки .LC2

mov eax, OFFSET FLAT:.LC2

; Передаем адрес строки в функцию printf

push eax

call printf

add esp, 4 ; Очищаем стек

; Другие инструкции

; ...

### Итог:

* **OFFSET FLAT:.LC2** используется для получения абсолютного адреса метки .LC2 в плоской модели памяти.
* Этот адрес может быть использован для передачи строки в функции, такие как printf, для вывода или других операций, требующих адреса строки.

Таким образом, OFFSET FLAT:.LC2 — это инструмент, который помогает получить адрес строки в памяти для дальнейшего использования в программе.

1. **Метки (.LC0 и .LC1)**:
   * .LC0 и .LC1 — это метки, которые обозначают адреса в памяти, где начинаются данные. Метки используются для удобства ссылки на эти данные в других частях программы.
2. **Директива .long**:
   * .long — это директива ассемблера, которая указывает, что следующее значение должно быть сохранено в памяти как 32-битное целое число (long).

### Разбор данных:

#### .LC0:

* **Первое значение (.long 0)**:
  + Это 32-битное целое число, которое сохраняется в памяти как 0.
* **Второе значение (.long 1072693248)**:
  + Это 32-битное целое число, которое сохраняется в памяти как 1072693248.

#### .LC1:

* **Первое значение (.long 0)**:
  + Это 32-битное целое число, которое сохраняется в памяти как 0.
* **Второе значение (.long 1073741824)**:
  + Это 32-битное целое число, которое сохраняется в памяти как 1073741824.

### Что это означает:

* **Двойные значения**:
  + Каждая метка .LC0 и .LC1 содержит два 32-битных значения. Это может указывать на то, что эти данные представляют собой числа с плавающей запятой двойной точности (double), которые хранятся в формате IEEE 754.
* **Формат IEEE 754**:
  + Числа с плавающей запятой двойной точности (double) в формате IEEE 754 занимают 64 бита (8 байт). Они состоят из:
    - 1 бит знака (S)
    - 11 бит экспоненты (E)
    - 52 бита мантиссы (M)

### Интерпретация значений:

#### .LC0:

* **Первое значение (0)**:
  + Это младшая часть числа с плавающей запятой. В данном случае, это 0.
* **Второе значение (1072693248)**:
  + Это старшая часть числа с плавающей запятой. В двоичном виде: 01000000000000000000000000000000.
  + Это соответствует числу 1.0 в формате IEEE 754.

#### .LC1:

* **Первое значение (0)**:
  + Это младшая часть числа с плавающей запятой. В данном случае, это 0.
* **Второе значение (1073741824)**:
  + Это старшая часть числа с плавающей запятой. В двоичном виде: 01000000000000000000000000000000.
  + Это соответствует числу 2.0 в формате IEEE 754.

### Итог:

* **.LC0**:
  + Содержит число 1.0 в формате IEEE 754.
* **.LC1**:
  + Содержит число 2.0 в формате IEEE 754.

Таким образом, данный сегмент ассемблерного кода сохраняет два числа с плавающей запятой двойной точности: 1.0 и 2.0, которые могут быть использованы в программе для различных вычислений или операций.

## Почему команда называется pxor, а не xor:

Команда pxor в архитектуре x86-64 называется именно так, потому что она предназначена для работы с **векторными регистрами** (регистры XMM, YMM, ZMM), которые используются в векторных расширениях, таких как SSE, AVX, AVX-512.

### Различия между xor и pxor:

1. **xor**:
   * **Операнды**: Работает с регистрами общего назначения (GPR) или памятью.
   * **Размер операндов**: 8, 16, 32 или 64 бита.
   * **Пример**: xor eax, ebx — выполняет побитовую операцию XOR между регистрами EAX и EBX.
2. **pxor**:
   * **Операнды**: Работает с векторными регистрами XMM.
   * **Размер операндов**: 128 бит (в SSE), 256 бит (в AVX), 512 бит (в AVX-512).
   * **Пример**: pxor xmm0, xmm1 — выполняет побитовую операцию XOR между регистрами XMM0 и XMM1.

### Префикс p (packed):

* **p** в названии команды означает **packed**, что указывает на то, что операция выполняется над **пакетом данных** (вектором) внутри регистра.
* Например, в случае pxor, операция XOR выполняется одновременно над всеми битами в 128-битном регистре XMM.