Регистры общего назначения:

* **EAX** (Accumulator): для арифметических операций
* **ECX** (Counter): для хранения счетчика цикла
* **EDX** (Data): для арифметических операций и операций ввода-вывода
* **EBX** (Base): указатель на данные
* **ESP** (Stack pointer): указатель на верхушку стека
* **EBP** (Base pointer): указатель на базу стека внутри функции
* **ESI** (Source index): указатель на источник при операциях с массивом
* **EDI** (Destination index): указатель на место назначения в операциях с массивами
* **EIP**: указатель адреса следующей инструкции для выполнения
* **EFLAGS**: регистр флагов, содержит биты состояния процессора
* **eax/ax/ah/al** (Accumulator register) - *аккумулятор*.

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

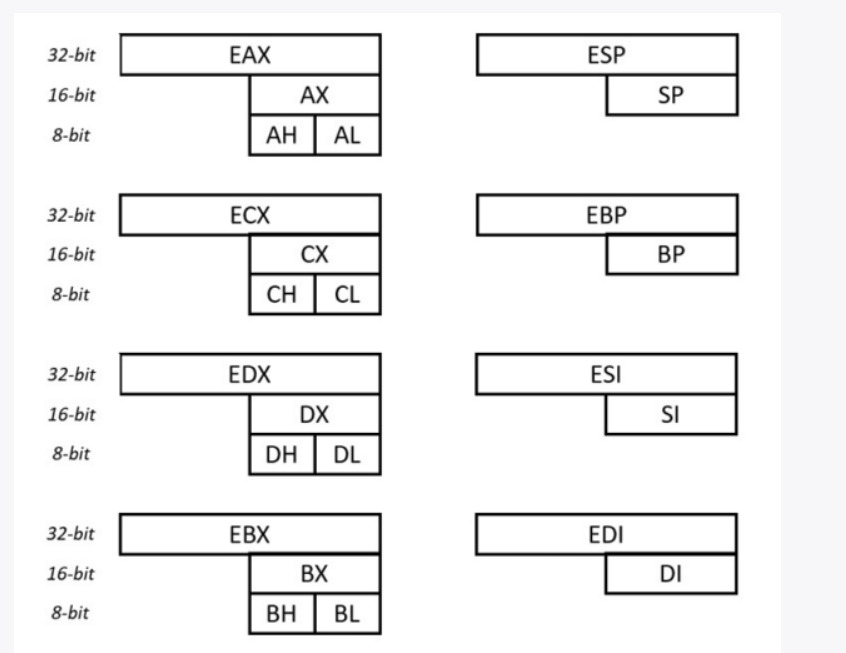
* **ebx/bx/bh/bl** (Base register) - *базовый* регистр.

Часто применяется для хранения базового адреса некоторого объекта в памяти;

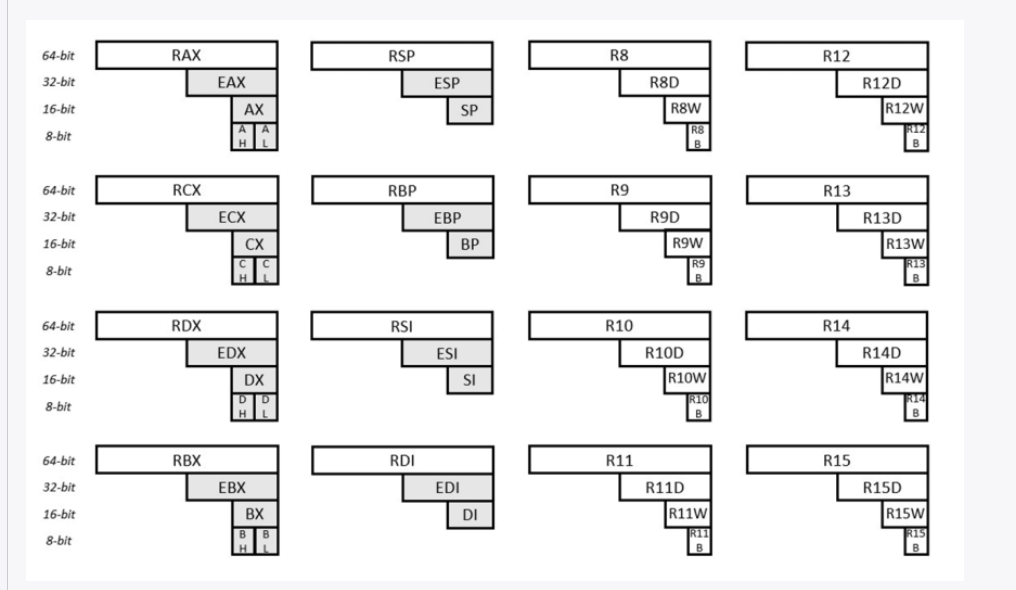
* **ecx/cx/ch/cl** (Count register) - *регистр-счетчик*.

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*;

Можно получить доступ к частям 32-битных регистров с меньшей разрядностью. Например, младшие 16 бит 32-битного регистра EAX обозначаются как AX. К регистру AX можно обращаться как к отдельным байтам, используя имена AH (старший байт) и AL (младший байт).



В архитектуре х64 эти регистры были расширены до 64 бит, а новые расширенные регистры получили имена, которые начинаются с буквы **R**, например, RAX, RBX и т.д. Кроме того, были добавлены 8 новых 64-битных регистров R8 - R15



Для обращения к 32-, 16- и 8-битной части 64-разрядных регистров используются стандартные для архитектуры x86 имена регистров. Для доступа к подрегистрам новых 64-битных регистров R8 - R15 применяется соответствующий суффикс:

* **D**: для получения младших 32 бит регистра, например, R11D
* **W**: для получения младших 16 бит регистра, например, R11W
* **B**: для получения младших 8 бит регистра, например, R11B

Таким образом, в архитектуре x86-64 мы можем использовать следующие регистры общего пользования:

* Шестнадцать 64-разрядных регистров RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8, R9, R10, R11, R12, R13, R14 и R15
* Шестнадцать 32-разрядных регистров EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, R8D, R9D, R10D, R11D, R12D, R13D, R14D и R15D
* Шестнадцать 16-разрядных регистров AX, BX, CX, DX, SI, DI, BP, SP, R8W, R9W, R10W, R11W, R12W, R13W, R14W и R15W
* Шестнадцать 8-разрядных регистров AL, AH, BL, BH, CL, CH, DL, DH, DIL, SIL, BPL, SPL, R8B, R9B, R10B, R11B, R12B, R13B, R14B и R15B

Например, запись в часть 64-битного регистра, например в регистр AL, влияет только на биты этой части. В случае AL загрузка 8-битного значения изменяет младшие 8 битов RAX, оставляя остальные 56 бит без изменений.

Хотя эти регистры и называются общего назначения, но это не значит, что их можно использовать для любых целей. Все регистры x86-64 имеют свое особое назначение, которое ограничивает их использование в определенных контекстах.

В x86\_64 аргументы функции передаются через регистры и стек.

Первые шесть аргументов передаются через регистры:

rdi - первый аргумент

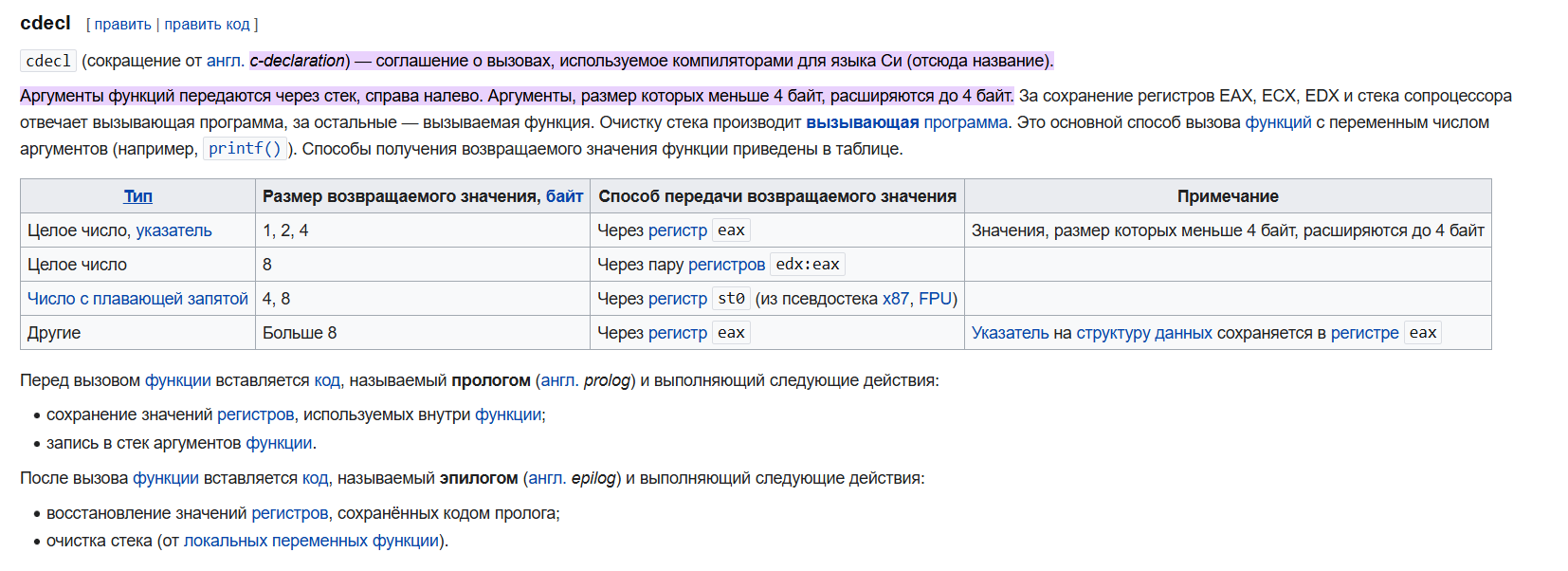
rsi - второй аргумент

rdx - третий аргумент

rcx - четвертый аргумент

r8 - пятый аргумент

r9 - шестой аргумент



Остальные аргументы передаются через стек.

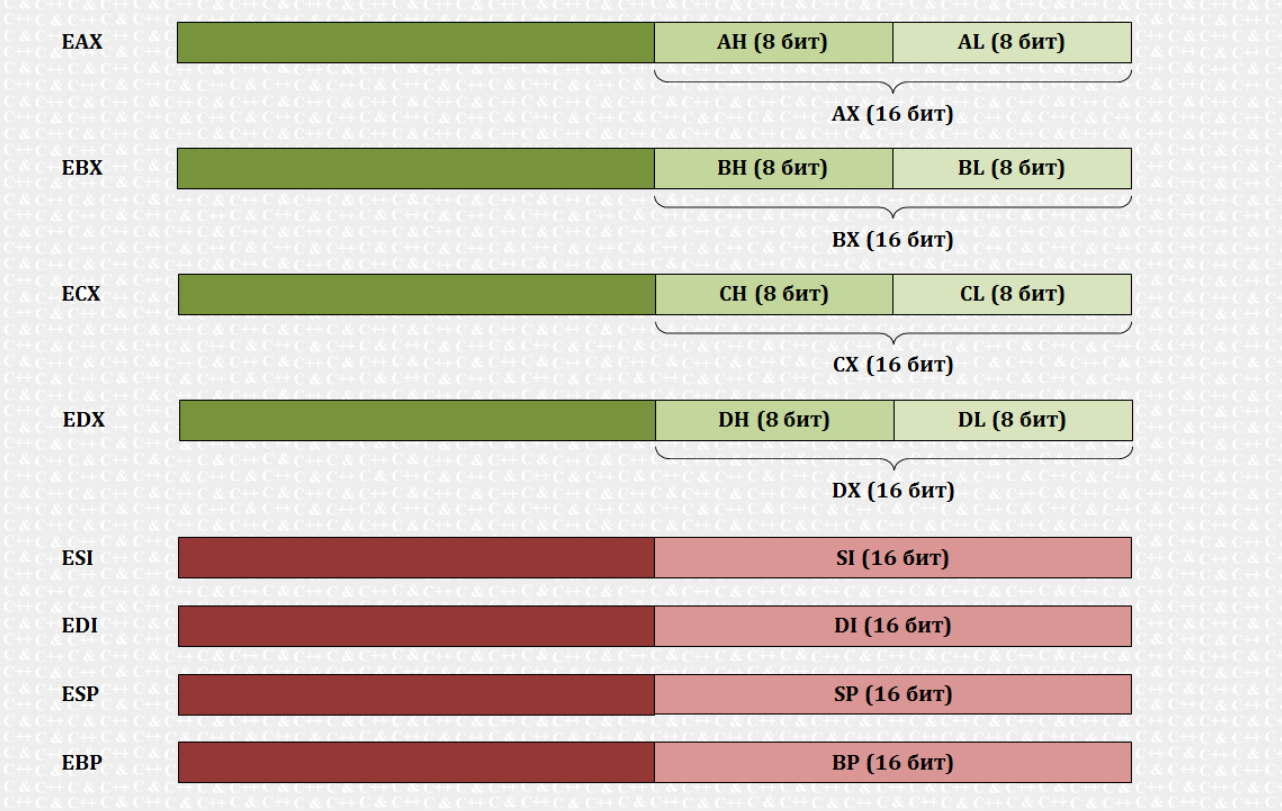
Функция может получить доступ к аргументам через соответствующие регистры или извлечь их из стека.

Функция в x86-64 узнаёт, куда помещены аргументы, через соглашение о вызовах. В нём указывается, как передаются аргументы (в регистрах или на стеке), какие именно регистры используются и где хранится результат.

Например, в 64-amd System V ABI аргументы передаются последовательно в регистрах: %rdi, %rsi, %rdx, %rcx, %r8 и %r9. Если передаётся более 6 аргументов, то оставшиеся — через стек.

Функция знает, где находятся аргументы, исходя из соглашений о вызовах (calling conventions), которые определяют порядок и место передачи аргументов.

Также при запуске программы регистр rsp указывает на вершину стека, где хранится число аргументов, переданных программе, указатели на эти аргументы, а также переменные окружения и другая информация.



Как видно из рисунка, регистры ESI, EDI, ESP и EBP позволяют обращаться к младшим 16 битам по именам SI, DI, SP и BP соответственно, а регистры EAX, EBX, ECX и EDX позволяют обращаться как к младшим 16 битам (по именам AX, BX, CX и DX), так и к двум младшим байтам по отдельности (по именам AH/AL, BH/BL, CH/CL и DH/DL).

Названия регистров происходят от их назначения:

* **EAX/AX/AH/AL** (*accumulator register*) – аккумулятор;
* **EBX/BX/BH/BL** (*base register*) –регистр базы;
* **ECX/CX/CH/CL** (*counter register*) – счётчик;
* **EDX/DX/DH/DL** (*data register*) – регистр данных;
* **ESI/SI** (*source index register*) – индекс источника;
* **EDI/DI** (*destination index register*) – индекс приёмника (получателя);
* **ESP/SP** (*stack pointer register*) – регистр указателя стека;
* **EBP/BP** (*base pointer register*) – регистр указателя базы кадра стека.

Несмотря на существующую специализацию, все регистры можно использовать в любых машинных операциях. Однако надо учитывать тот факт, что некоторые команды работают только с определёнными регистрами. Например, команды умножения и деления используют регистры EAX и EDX для хранения исходных данных и результата операции. Команды управления циклом используют регистр ECX в качестве счётчика цикла.

Ещё один нюанс состоит в использовании регистров в качестве *базы*, т.е. хранилища адреса оперативной памяти. В качестве регистров базы можно использовать любые регистры, но желательно использовать регистры EBX, ESI, EDI или EBP. В этом случае размер машинной команды обычно бывает меньше.

**Можно ли хранить вещественные числа в регистрах общего назначения?**

Да, можно хранить вещественные числа в регистрах общего назначения (GPR), но это не является оптимальным подходом. Вещественные числа обычно хранятся в специальных регистрах с плавающей запятой, таких как XMM, YMM, ZMM в архитектуре x86-64.

**Сопроцессор.** Сопроцессор (FPU – Floating Point Unit) 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. Например: st(0) регистр на вершине стека, st(1) – следующий за ним и т.д

**Для чего нужен "сопроцессор" в архитектуре x86?**

**Как связаны регистры сопроцессора со стеком?**

***Сопроцессор в архитектуре x86 (например, FPU — Floating Point Unit) предназначен для выполнения операций с плавающей запятой. Он предоставляет регистры и инструкции, оптимизированные для работы с вещественными числами, что позволяет выполнять операции с высокой точностью и скоростью.***

***В архитектуре x86 сопроцессор имеет свой собственный стек, который используется для хранения промежуточных результатов и операндов. Регистры сопроцессора (ST0, ST1, ..., ST7) организованы в виде стека, где ST0 — вершина стека. Операции с плавающей запятой обычно выполняются с использованием этого стека.***

***Команды сопроцессора работают с вершиной стека (ST(0)) и могут изменять его содержимое, перемещать данные между регистрами стека или загружать/выгружать данные из памяти.***

1. **Встроенная поддержка:**
   * Современные процессоры, такие как x86-64, имеют встроенный сопроцессор, который работает параллельно с основным процессором. Это позволяет выполнять операции с плавающей точкой без необходимости переключения контекста и с минимальными задержками.
2. **Производительность:**
   * Использование сопроцессора значительно повышает производительность при выполнении вычислений с вещественными числами. Операции с плавающей точкой выполняются быстрее и точнее, чем при использовании GPR.

**Сравнение с GPR**

* **GPR:**
  + Используются для хранения целых чисел и адресов памяти.
  + Операции с плавающей точкой в GPR требуют преобразований и масштабирования, что снижает производительность и точность.
* **Сопроцессор:**
  + Специализирован для операций с плавающей точкой.
  + Предоставляет оптимизированные регистры и инструкции для высокоскоростных и точных вычислений.

### Архитектура x86 и x86-64: Различия в использовании сопроцессора и SSE

#### Архитектура x86

В архитектуре x86 (32-битная) сопроцессор (FPU — Floating Point Unit) используется явно для выполнения операций с плавающей запятой. Основные причины этого:

1. **Исторические причины:**
   * В ранних версиях x86 сопроцессор был отдельным устройством, которое могло быть не установлено на всех процессорах. Поэтому программисты должны были явно проверять наличие сопроцессора и использовать его инструкции.
2. **Специфический набор инструкций:**
   * Сопроцессор имеет свой собственный набор инструкций (например, fld, fadd, fstp), которые работают с регистрами стека сопроцессора (ST0, ST1, ...). Эти инструкции явно управляют стеком сопроцессора.
3. **Организация стека:**
   * Стек сопроцессора организован как стек регистров, где ST0 — вершина стека. Операции с данными в стеке требуют явного управления, такого как загрузка, сохранение и обмен данными между регистрами.

#### Архитектура x86-64

В архитектуре x86-64 (64-битная) сопроцессор интегрирован в основной процессор, и его использование стало менее явным. Основные причины этого:

1. **Интеграция сопроцессора:**
   * В x86-64 сопроцессор (FPU) интегрирован в основной процессор, что упрощает его использование. Программисты могут использовать инструкции сопроцессора, но они не обязаны явно управлять стеком сопроцессора.
2. **Расширение SSE:**
   * В x86-64 широко используется расширение SSE (Streaming SIMD Extensions), которое предоставляет дополнительные регистры (XMM0-XMM15) и инструкции для работы с векторными данными, включая вещественные числа. SSE более удобен для современных вычислений, так как позволяет выполнять параллельные операции над несколькими данными одновременно.
3. **Упрощение программирования:**
   * Использование SSE позволяет упростить программирование, так как регистры XMM не требуют явного управления стеком. Операции с плавающей запятой могут быть выполнены с использованием прямых инструкций, таких как addps, mulps, movaps, которые работают с регистрами XMM.

**Почему вырос объём доступной оперативной памяти при переходе на x86\_64?**

Увеличение разрядности адреса позволяет процессору работать с большими объемами данных и повышает производительность приложения.

Объём доступной оперативной памяти вырос при переходе на x86\_64, потому что 64-разрядные процессоры могут работать с практически неограниченным объёмом ОЗУ.

### Улучшенная производительность

* **Больше регистров:**
  + В 64-битной архитектуре количество регистров общего назначения увеличено с 8 до 16, что позволяет хранить больше данных в регистрах и уменьшать количество обращений к памяти.
  + Это повышает производительность, особенно при работе с большими объемами данных.
* **Больший размер регистров:**
  + Регистры в 64-битной архитектуре имеют размер 64 бита, что позволяет хранить больше данных и выполнять операции с большими числами.

### Что такое "адрес возврата функции"?

Адрес возврата функции в стеке хранится в специально выделенном месте, которое называется **фреймом стека** (stack frame). Фрейм стека создается при вызове функции и содержит информацию, необходимую для возврата управления вызывающей функции после завершения работы вызванной функции.

**Структура фрейма стека**

При вызове функции в архитектуре x86-64 (и x86) фрейм стека обычно содержит следующие элементы:

1. **Адрес возврата (Return Address):**
   * Это адрес команды, которая будет выполняться после завершения работы функции.
   * Адрес возврата сохраняется в стеке с помощью команды call.
2. **Сохраненные регистры (Saved Registers):**
   * Регистры, которые используются функцией, могут быть сохранены в стеке, чтобы их значения могли быть восстановлены после завершения работы функции.
   * Обычно это делается с помощью команд push и pop.
3. **Локальные переменные (Local Variables):**
   * Если функция использует локальные переменные, они также могут быть размещены в стеке.
4. **Аргументы функции (Function Arguments):**
   * Аргументы функции могут быть переданы через стек или регистры.
   * В x86-64 первые шесть аргументов передаются через регистры (RDI, RSI, RDX, RCX, R8, R9), а остальные — через стек.

**Пример создания и использования фрейма стека**

Рассмотрим пример вызова функции и создания фрейма стека.

assembly

Copy

section .data

arg1 dd 10

arg2 dd 20

section .text

global \_start

\_start:

; Вызов функции

mov edi, [arg1] ; Первый аргумент (через регистр RDI)

mov esi, [arg2] ; Второй аргумент (через регистр RSI)

call my\_function

; Завершение программы

mov eax, 60 ; Системный вызов exit

xor edi, edi ; Код возврата 0

syscall

my\_function:

; Создание фрейма стека

push rbp ; Сохранение предыдущего базового указателя

mov rbp, rsp ; Установка нового базового указателя

; Работа функции

; ...

; Восстановление фрейма стека

mov rsp, rbp ; Восстановление указателя стека

pop rbp ; Восстановление предыдущего базового указателя

ret ; Возврат из функции

**Подробное описание шагов**

1. **Вызов функции:**
   * Команда call my\_function сохраняет адрес следующей команды (адрес возврата) в стеке и передает управление функции my\_function.
2. **Создание фрейма стека:**
   * В начале функции my\_function сохраняется предыдущий базовый указатель (rbp) в стеке с помощью команды push rbp.
   * Затем текущий указатель стека (rsp) копируется в базовый указатель (rbp), чтобы установить новый базовый указатель для текущей функции.
3. **Работа функции:**
   * Внутри функции могут использоваться локальные переменные, которые размещаются в стеке.
4. **Восстановление фрейма стека:**
   * Перед возвратом из функции, указатель стека (rsp) восстанавливается до значения базового указателя (rbp) с помощью команды mov rsp, rbp.
   * Затем из стека извлекается сохраненный базовый указатель (rbp) с помощью команды pop rbp.
5. **Возврат из функции:**
   * Команда ret извлекает адрес возврата из стека и передает управление по этому адресу.

**Заключение**

Адрес возврата функции хранится в стеке в рамках фрейма стека, который создается при вызове функции. Фрейм стека также содержит другие важные данные, такие как сохраненные регистры и локальные переменные. Управление фреймом стека осуществляется с помощью базового указателя (rbp) и указателя стека (rsp), что позволяет функции корректно восстановить свое состояние и вернуть управление вызывающей функции.

### 

Адрес возврата функции — это адрес команды, которая будет выполняться после завершения работы функции. Когда функция вызывается, адрес возврата сохраняется в стеке, чтобы процессор знал, куда вернуться после завершения выполнения функции.

В вычислительной технике адрес возврата относится к значению, которое используется для указания того, куда конкретная функция должна вернуть управление после завершения выполнения. Когда вызывается функция, адрес инструкции, следующей за вызовом функции, сохраняется в системном стеке. Это адрес возврата. После завершения выполнения функции управление возвращается по этому адресу, позволяя программе продолжить работу с того места, где она остановилась. Этот механизм необходим для поддержания плавного и логичного потока выполнения программы.

Адрес возврата имеет решающее значение, поскольку он позволяет вашей программе поддерживать ее поток и логику. Когда вы вызываете функцию, программа должна знать, куда вернуться после выполнения этой функции. Без правильно управляемого адреса возврата ваша программа может потеряться в своем собственном выполнении, что приведет к ошибкам, сбоям или неожиданному поведению.

В вычислительной технике стек — это динамическая структура данных, которая хранит адреса возврата. Когда вызывается функция, программа «проталкивает» адрес возврата, т. е. адрес инструкции, следующей за вызовом функции, в стек. После того, как функция завершает выполнение, программа «выталкивает» самый верхний адрес возврата из стека и продолжает выполнение с этой точки. Этот механизм стека, работающий по принципу Last In, First Out (LIFO), имеет решающее значение для поддержания упорядоченного потока инструкций, особенно в программах с несколькими вложенными вызовами функций.

### Как процессор реализует "флаги"?

### Флаги используются для принятия решений в условных переходах.Процессор реализует флаги через специальные регистры — хранилища данных, которые могут принимать различные значения. Каждый флаг в регистре флагов содержит один бит информации о результате выполнения операции. Например:

### Флаг переноса (CF) устанавливается, если произошёл перенос в старший разряд при выполнении операции сложения, вычитания или сдвига.

### Флаг нуля (ZF) устанавливается, если результат операции равен нулю.

### Флаг переполнения (OF) устанавливается, если результат операции не может быть представлен в заданном диапазоне (например, при сложении двух положительных чисел получается отрицательное число).

### Флаг знака (SF) устанавливается, если результат операции отрицательный.

### Регистр флагов RFLAGS( EFLAGS), содержит биты состояния процессора, которые указывают на результаты операций:

*Флаг* – это бит, принимающий значение 1 («флаг установлен»), если выполнено некоторое условие, и значение 0 («флаг сброшен») в противном случае. Процессор имеет *регистр флагов*, содержащий набор флагов, отражающий текущее состояние процессора.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **№ бита** | | **Обозначение** | | **Название** | | **Описание** | **Тип флага** |
| FLAGS | | | | | | | |
| 0 | | CF | Carry Flag | | Флаг переноса | | Состояние |
| 1 | | 1 | | | Зарезервирован | |  |
| 2 | | PF | | Parity Flag | Флаг чётности | | Состояние |
| 3 | | 0 | | | Зарезервирован | |  |
| 4 | | AF | | Auxiliary Carry Flag | Вспомогательный флаг переноса | | Состояние |
| 5 | | 0 | | | Зарезервирован | |  |
| 6 | | ZF | | Zero Flag | Флаг нуля | | Состояние |
| 7 | | SF | | Sign Flag | Флаг знака | | Состояние |
| 8 | | TF | | Trap Flag | Флаг трассировки | | Системный |
| 9 | | IF | | Interrupt Enable Flag | Флаг разрешения прерываний | | Системный |
| 10 | | DF | | Direction Flag | Флаг направления | | Управляющий |
| 11 | | OF | | Overflow Flag | Флаг переполнения | | Состояние |
| 12 | | IOPL | | I/O Privilege Level | Уровень приоритета ввода-вывода | | Системный |
| 13 | |
| 14 | | NT | | Nested Task | Флаг вложенности задач | | Системный |
| 15 | | 0 | | | Зарезервирован | |  |
| EFLAGS | | | | | | | |
| 16 | RF | | | Resume Flag | Флаг возобновления | | Системный |
| 17 | VM | | | Virtual-8086 Mode | Режим виртуального процессора 8086 | | Системный |
| 18 | AC | | | Alignment Check | Проверка выравнивания | | Системный |
| 19 | VIF | | | Virtual Interrupt Flag | Виртуальный флаг разрешения прерывания | | Системный |
| 20 | VIP | | | Virtual Interrupt Pending | Ожидающее виртуальное прерывание | | Системный |
| 21 | ID | | | ID Flag | Проверка на доступность инструкции CPUID | | Системный |
| 22 |  | | | | Зарезервированы | |  |
| **...** |
| 31 |

1. **Бит 0 (CF — Carry Flag):**

CF (Carry Flag) устанавливается, если результат операции не умещается в регистре.

* + Указывает на перенос или заем после арифметических операций.

1. **Бит 1 (Reserved):**
   * Этот бит всегда должен быть установлен в 1. Попытка изменить его значение может привести к неопределенному поведению.
2. **Бит 2 (PF — Parity Flag):**
   * Указывает на четность младшего байта результата (четное или нечетное количество единиц).
3. **Бит 3 (Reserved):**
   * Этот бит зарезервирован и не должен использоваться в пользовательском коде.
4. **Бит 4 (AF — Auxiliary Carry Flag):**
   * Указывает на перенос или заем между младшими тетрадами (полубайтами) в результате арифметических операций.
5. **Бит 5 (Reserved):**
   * Этот бит зарезервирован и не должен использоваться в пользовательском коде.
6. **Бит 6 (ZF — Zero Flag):**

ZF (Zero Flag) устанавливается, если результат операции равен нулю.

* + Указывает на то, что результат операции равен нулю.

1. **Бит 7 (SF — Sign Flag):**

SF (Sign Flag) устанавливается, если результат операции отрицательный.

* + Указывает на знак результата (отрицательный или положительный).

1. **Бит 8 (TF — Trap Flag):**
   * Включает одношаговый режим отладки.
2. **Бит 9 (IF — Interrupt Enable Flag):**
   * Управляет обработкой прерываний.
3. **Бит 10 (DF — Direction Flag):**
   * Управляет направлением обработки строковых операций.
4. **Бит 11 (OF — Overflow Flag):**
   * Указывает на переполнение при арифметических операциях.

### Причины резервирования битов 1, 3 и 5

1. **Бит 1:**
   * Этот бит всегда должен быть установлен в 1. Это связано с историческими причинами и совместимостью с более ранними версиями процессоров. Установка этого бита в 1 обеспечивает совместимость с предыдущими архитектурами и предотвращает неопределенное поведение.
2. **Биты 3 и 5:**
   * Эти биты зарезервированы для будущих использований. Хотя в текущих версиях процессоров они не используются, резервирование этих битов позволяет сохранить возможность их использования в будущих архитектурах без нарушения совместимости с существующим кодом.

Значение флагов CF, DF и IF можно изменять напрямую в регистре флагов с помощью специальных инструкций (например, *CLD* для сброса флага направления), но нет инструкций, которые позволяют обратиться к регистру флагов как к обычному регистру. Однако можно сохранять регистр флагов в стек или регистр AH и восстанавливать регистр флагов из них с помощью инструкций *LAHF*, *SAHF*, *PUSHF*, *PUSHFD*, *POPF* и *POPFD*.

**1.3.1. Флаги состояния**

Флаги состояния (биты 0, 2, 4, 6, 7 и 11) отражают результат выполнения арифметических инструкций, таких как *ADD*, *SUB*, *MUL*, *DIV*.

* *Флаг переноса* CF устанавливается при переносе из старшего значащего бита/заёме в старший значащий бит и показывает наличие переполнения в беззнаковой целочисленной арифметике. Также используется в длинной арифметике.
* *Флаг чётности* PF устанавливается, если младший значащий байт результата содержит чётное число единичных битов. Изначально этот флаг был ориентирован на использование в коммуникационных программах: при передаче данных по линиям связи для контроля мог также передаваться бит чётности и инструкции для проверки флага чётности облегчали проверку целостности данных.
* *Вспомогательный флаг переноса* AF устанавливается при переносе из бита 3-го результата/заёме в 3-ий бит результата. Этот флаг ориентирован на использование в двоично-десятичной (binary coded decimal, BCD) арифметике.
* *Флаг нуля* ZF устанавливается, если результат равен нулю.
* *Флаг знака* SF равен значению старшего значащего бита результата, который является знаковым битом в знаковой арифметике.
* *Флаг переполнения* OF устанавливается, если целочисленный результат слишком длинный для размещения в целевом операнде (регистре или ячейке памяти). Этот флаг показывает наличие переполнения в знаковой целочисленной арифметике.

Из перечисленных флагов только флаг CF можно изменять напрямую с помощью инструкций *STC*, *CLC* и *CMC*.

Флаги состояния позволяют одной и той же арифметической инструкции выдавать результат трёх различных типов: беззнаковое, знаковое и двоично-десятичное (BCD) целое число. Если результат считать беззнаковым числом, то флаг CF показывает условие переполнения (перенос или заём), для знакового результата перенос или заём показывает флаг OF, а для BCD-результата перенос/заём показывает флаг AF. Флаг SF отражает знак знакового результата, флаг ZF отражает и беззнаковый, и знаковый нулевой результат.

В длинной целочисленной арифметике флаг CF используется совместно с инструкциями сложения с переносом (*ADC*) и вычитания с заёмом (*SBB*) для распространения переноса или заёма из одного вычисляемого разряда длинного числа в другой.

Инструкции условного перехода *Jcc* (переход по условию *cc*), *SETcc* (установить значение байта-результата в зависимости от условия *cc*), *LOOPcc* (организация цикла) и *CMOVcc* (условное копирование) используют один или несколько флагов состояния для проверки условия. Например, инструкция перехода *JLE* (*jump if less or equal* – переход, если «меньше или равно») проверяет условие «ZF = 1 или SF ≠ OF».

Флаг PF был введён для совместимости с другими микропроцессорными архитектурами и по прямому назначению используется редко. Более распространено его использование совместно с остальными флагами состояния в арифметике с плавающей запятой: инструкции сравнения (*FCOM*, *FCOMP* и т. п.) в математическом сопроцессоре устанавливают в нём флаги-условия C0, C1, C2 и C3, и эти флаги можно скопировать в регистр флагов. Для этого рекомендуется использовать инструкцию *FSTSW  AX* для сохранения слова состояния сопроцессора в регистре AX и инструкцию *SAHF* для последующего копирования содержимого регистра AH в младшие 8 битов регистра флагов, при этом C0 попадает во флаг CF, C2 – в PF, а C3 – в ZF. Флаг C2 устанавливается, например, в случае несравнимых аргументов (NaN или неподдерживаемый формат) в инструкции сравнения *FUCOM*.

**1.3.2. Управляющий флаг**

*Флаг направления* DF (бит 10 в регистре флагов) управляет строковыми инструкциями (*MOVS*, *CMPS*, *SCAS*, *LODS* и *STOS*) – установка флага заставляет уменьшать адреса (обрабатывать строки от старших адресов к младшим), обнуление заставляет увеличивать адреса. Инструкции *STD* и *CLD* соответственно устанавливают и сбрасывают флаг DF.

Флаги — это биты в регистре состояния процессора (например, EFLAGS в x86), которые указывают на результаты операций. Например, флаг ZF (Zero Flag) устанавливается, если результат операции равен нулю, а флаг CF (Carry Flag) — если произошел перенос. Процессор устанавливает эти флаги автоматически после выполнения арифметических и логических операций.

### Что такое jle и jne?

### 

* **jle (Jump if Less or Equal):** Это команда условного перехода, которая выполняет переход, если результат предыдущего сравнения меньше или равен нулю. Она проверяет флаги SF (Sign Flag), OF (Overflow Flag) и ZF (Zero Flag).

jle (Jump if Less or Equal) - команда условного перехода, которая выполняет переход, если значение флага SF не равно значению флага OF или значение флага ZF установлено.

* **jne (Jump if Not Equal):** Это команда условного перехода, которая выполняет переход, если результат предыдущего сравнения не равен нулю. Она проверяет флаг ZF.

jne (Jump if Not Equal) - команда условного перехода, которая выполняет переход, если значение флага ZF сброшено.

### Какая команда отвечает за безусловный переход?

Команда **jmp** отвечает за безусловный переход. Она передает управление по указанному адресу без каких-либо условий.

jmp - команда безусловного перехода, которая всегда выполняет переход на указанный адрес.

### Как jle узнает о результатах сравнения?

Команда jle использует флаги, установленные предыдущей командой сравнения (например, cmp). Она проверяет флаги SF, OF и ZF, чтобы определить, выполнять ли переход. Если результат сравнения меньше или равен нулю, jle выполняет переход.