# Disclaimer

Данный справочник составил студент 103 группы (2014 – 2015 год обучения) Аграновский Михаил.

Вы читаете версию от *24.08.2015*. Правка и распространение приветствуются.  
Надеюсь, что этот документ поможет Вам в успешном прохождении курса по Архитектуре ЭВМ и языку ассемблера!

Ахтунг! В тексте могут содержаться ошибки и неточности – будьте осторожны.

Для понимания, что же такое регистр, (если оно не пришло на лекции) советую прочитать следующий .pdf – документ: <http://www.itsusu.ru/pdf/information_technology/work_11.pdf>

(Возможно, Вы его уже скачали под именем “Ассемблер NASM для процессоров с архитектурой IA-32. Адресация. Флаги. Вызов внешних подпрограмм.pdf”)

Желаю удачи!

Оглавление

[Disclaimer 0](#_Toc428219423)

[Оглавление 1](#_Toc428219424)

[Таблица команд 2](#_Toc428219425)

[Сложение и вычитание 2](#_Toc428219426)

[Умножение и деление 3](#_Toc428219427)

[Перемещение данных и приведение типов 5](#_Toc428219428)

[Сдвиги и вращения 6](#_Toc428219429)

[Команды работы с битами 10](#_Toc428219430)

[Команды сканирования битов 10](#_Toc428219431)

[Команды проверки и модификации битов 10](#_Toc428219432)

[Организация стека 11](#_Toc428219433)

[Организация стекового фрейма функции. 12](#_Toc428219434)

[Регистр EFLAGS 14](#_Toc428219435)

[Условные переходы (условная передача управления) 16](#_Toc428219436)

[Условная передача данных 18](#_Toc428219437)

[Строковые инструкции 19](#_Toc428219438)

[Числа с плавающей точкой. Сопроцессор x87. 22](#_Toc428219439)

[Еще команды процессора x86: 24](#_Toc428219440)

[Средства ввода/вывода io.inc 25](#_Toc428219441)

[Источники 27](#_Toc428219442)

# Таблица команд

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **КОМАНДА** | **ОПИСАНИЕ** | | **СИНТАКСИС** | **Комментарий** |
| **XCHG op1, op2** | Обмен значениями. Команда XCHG выполняет обмен значений своих операндов. | | * XCHG r/m 8/16/32, r/m 8/16/32 (память, память нельзя) | Требование: sizeof(op1) == sizeof(op2) |
| **BSWAP op1** | Перевод op1 из одной формы адресации в другую (Big-endian <-> Little-endian).  Иначе говоря, разворот байтов.­­­ | | * DSWP r32 | На флаги не влияет.  <http://www.c-jump.com/CIS77/ASM/DataTypes/T77_0230_bswap.htm> |
| Сложение и вычитание | | | | |
| **NEG op1** | Изменение знака (получение двоичного дополнения к) op1.  Алгоритм работы:   * выполнить вычитание (0 – источник) и поместить результат на место источника; * если источник=0, то его значение не меняется. | |  | Команда используется для формирования двоичного дополнения операнда в памяти или регистре. Операция двоичного дополнения предполагает инвертирование всех разрядов операнда с последующим сложением операнда с двоичной единицей. Если операнд отрицательный, то операция neg над ним означает получение его модуля. |
| **ADD op1, op2** | Сложение | |  | Требование: sizeof(op1) == sizeof(op2) |
| **ADC op1, op2 SBB op1, op2** | *ADC:* Сложение двух целочисленных двоичных операндов с учётом флага переноса CF. op1 = op1 + op2 + CF  *SBB:*  Вычитание двух целочисленных двоичных операндов с учётом флага CF. Состояние флага CF представляет собой заём от предыдущего вычитания.  op1 = op1 - op2 - CF | | Вроде бы:   * r/m 8/16/32, r/m; (sizeof(op2) = sizeof(op1) ) * r/m 8/16/32, imm; (sizeof(imm) <= sizeof(op1) ) | **Флаги:**  (для обеих операций)  OF=r SF=r ZF=r AF=r PF=r CF=r  imm [immediate value]– непосредственная константа (число) |
| **AND op1, op2** | Побитовое "и" | |  | **Неразрушающий аналог:**  *TEST op1, op2* |
| **SUB op1, op2** | Вычитание | |  | **Неразрушающий аналог:**  *CMP op1, op2* |
| Умножение и деление | | | | |
| **MUL op1** | | Команда выполняет целочисленное умножение без учёта знака. Явно задается один из множителей. Второй множитель задается неявно в регистре AL/AX/EAX (это местоположение фиксировано). Местоположение результата умножения определяется кодом операции и размером множителей:   |  |  |  |  | | --- | --- | --- | --- | | Размер операндов | Первый множитель | Второй множитель | Результат | | Байт | AL | r/m8 | AX | | Слово | AX | r/m16 | DX:AX | | Двойное слово | EAX | r/m32 | EDX:EAX | | * r/m 8/16/32 | **Флаги**:  Если старшая половина результата нулевая:  OF=CF=0 SF=? ZF=? AF=? PF=?.  Если старшая половина результата ненулевая:  OF=CF=1 SF=? ZF=? AF=? PF=?. |
| **IMUL** | | Умножение со знаком  Команда имеет три формы, различающиеся количеством операндов:   * С одним операндом — требует явного указания местоположения только одного сомножителя, который может быть расположен в ячейке памяти или регистре. Местоположение произведения зависит от размерности множителей:  |  |  |  |  | | --- | --- | --- | --- | | Размер операндов | Первый множитель | Второй множитель | Результат | | Байт | AL | r/m8 | AX | | Слово | AX | r/m16 | DX:AX | | Двойное слово | EAX | r/m32 | EDX:EAX |  * С двумя операндами — первый операнд определяет местоположение первого сомножителя. На его место впоследствии будет записан результат. Второй операнд определяет местоположение второго сомножителя. * С тремя операндами — первый операнд определяет местоположение результата, второй операнд — местоположение первого сомножителя, третий операнд может быть непосредственно заданным значением размером в байт, слово или двойное слово. | (не совсем совпадает с лекцией)   * r 8/16/32 * r 16/32, r/m 16/32 * r 16/32, r/m 16/32, imm 16/32   imm должен по размеру совпадать с первым множителем или быть размерности до байта. | **Флаги:**   * Для однооперандной команды: OF=1 CF=1 — значимые биты переносятся в верхнюю половину результата; OF=0 CF=0 — результат помещается точно в младшей половине результата. Состояние остальных флагов: SF=? ZF=? AF=? PF=?. * Для двух- и трехоперандной команды: OF=1 CF=1 — результат слишком большой и усекается; OF=0 CF=0 — размер результата точно соответствует операнду назначения. Состояние остальных флагов: SF=? ZF=? AF=? PF=?.   **Двухоперандная форма не рекомендуема к использованию при умножении на imm** – поддерживается не всеми машинами. |
| **DIV op1** | | Деление без знака.  Команда выполняет целочисленное деление без учёта знака. Делимое задается неявно, и его размер зависит от размера делителя, который явно указывается в команде. Местоположения делимого, делителя, частного и остатка — в зависимости от их размерности:   |  |  |  |  |  | | --- | --- | --- | --- | --- | | Размер операнда | Делимое | Делитель | Частное  (a div b) | Остаток  (a mod b) | | Байт | AX | r/m8 | AL | AH | | Слово | DX:AX | r/m16 | AX | DX | | Двойное слово | EDX:EAX | r/m32 | EAX | EDX | | * r/m 8/16/32 |  |
| **IDIV** | | Деление со знаком.  Команда выполняет целочисленное деление со знаком. Делимое задается неявно, и его размер зависит от размера делителя, который явно указывается в команде. Местоположения делимого, делителя, частного и остатка — в зависимости от их размерности:   |  |  |  |  |  | | --- | --- | --- | --- | --- | | Размер операнда | Делимое | Делитель | Частное | Остаток | | Байт | AX | r/m8 | AL | AH | | Слово | DX:AX | r/m16 | AX | DX | | Двойное слово | EDX:EAX | r/m32 | EAX | EDX |   Остаток всегда имеет знак делимого. Знак частного зависит от состояния знаковых битов (старших разрядов) делимого и делителя. | * r/m 8/16/32 |  |
| Перемещение данных и приведение типов | | | | |
| **CBW**  **CWDE**  **CWD**  **CDQ** | | *CBW —* преобразование байта в слово.  Команда копирует знаковый бит регистра AL/AX на все биты регистра АН/ЕАХ (используется при подготовке к знаковому делению):  movsx ax, al  *CWD*, *CWDE* — преобразование слова в двойное слово.  Команда *CWD* копирует значение старшего бита регистра АХ на все биты регистра DX (используется при подготовке к знаковому делению):  mov dx, ax  sar dx, 15  *CWDE* же:  movsx eax, ax  *CDQ* — преобразование двойного слова в учетверённое слово.  Команда CDQ копирует знаковый бит регистра ЕАХ на все биты регистра EDX (используется при подготовке к знаковому делению):  mov edx, eax  sar edx, 31 |  | Convert Byte to Word;  Convert Word to Dword;  Convert Double to Quad. |
| **MOVZX op1, op2**  **MOVSX op1, op2** | | *op2* преобразуется к типу *op1* и записывается в него. При этом:  *MOVZX* выполняет беззнаковое расширение данных (число до нужного размера дополняется слева нулями); *MOVSX*  выполняет знаковое расширение данных (число до нужного размера дополняется слева битами со значением знакового разряда исходного числа). | * MOVZX (r 32), (r/m 16) * MOVSX (r 16/32), (r/m 8) | movsx (**mov**e **s**igned e**x**tension);  movzx (**mov**e **z**ero e**x**tension).  Например: mov bx, 0xA67B ; bx ← 0xA67B  movzx ebx, bx ; ebx ← 0x0000A67B  movsx eax, bx ; eax ← 0xFFFFA67B  movsx ecx, bl ; ecx ← 0x0000007B |
| **LEA op1, op2** | | Получение эффективного адреса (смещения) источника.  При этом **обращение к памяти** (op2) **не производится** (несмотря на применение квадратных скобок), а, вместо этого, в регистр (op1) заносится адрес, вычисленный обычным способом для второго операнда. (В op1 запишутся младшие sizeof(op1) байт вычисленного адреса).  **Общий вид адресного кода при обращении к памяти:** | - r (16) / 32, m  Например: lea eax, [ebx+esi\*2+2] | «Load Effective Address»  Опр. Эффективный адрес — адрес, по которому происходит обращение к памяти; численно совпадает с виртуальным адресом.   * в качестве базы может выступать любой РОН (регистр общего назначения); * индексом может быть любой РОН, кроме ESP; * коэффициент масштабирования может принимать численное значение 1, 2, 4 или 8; * смещение задается константой.   [http://www.konetlasc..ru/cdo/programmes/assembler/lea.html](http://www.kolasc.net.ru/cdo/programmes/assembler/lea.html)  Суть:  Пусть имеются переменные int \*a, \*b и eax = a, ebx = b.  Тогда: mov eax, [ebx + 4]; <=> a = \*(b+1)  lea eax, [ebx + 4]; <=> a = b + 4 |
| Сдвиги и вращения | | | | |
| **SHL op1, op2**  **SHR op1, op2** | | Логический (беззнаковый) сдвиг влево / вправо  *Для SHR:*  Сдвиг всех битов операнда вправо на количество разрядов, указанное операндом *op2*, при этом выдвигаемый справа бит становится значением флага переноса CF. Одновременно слева в операнд вдвигается нулевой бит. | * op1 = r/m 8/16/32 * op2 = CL / imm | **Флаги:**  *SHR:* CF=r OF=?r SF=r ZF=r PF=r AF=?  Флаг CF содержит значение последнего выдвинутого из операнда бита. При сдвиге на 1 бит флаг OF устанавливается равным старшему значащему биту исходного операнда. Если количество сдвигов (значение *op2*) больше 1, то OF=0.  *SHL:*  CF=r OF=?r SF=r ZF=r PF=r AF=?  Флаг CF содержит значение последнего выдвинутого из операнда бита. Флаг OF представляет интерес, если сдвиг осуществляется на один разряд. При сдвиге на несколько разрядов его значение не определено. По его значению можно судить об изменении знакового разряда операнда:   * если OF=1, то значение флага CF и значение старшего бита операнда после выполнения сдвига различны; * если OF=0, то значение флага CF и значение старшего бита операнда после выполнения сдвига совпадают. |
| **ROL op1, number or CL**  **ROR op1, number or CL** | | Циклический сдвиг влево / вправо  *ROR:*  При циклическом сдвиге разрядов операнда вправо его младший выдвигаемый бит вдвигается в op1 слева и становится одновременно значением старшего бита операнда и флага переноса CF.  *ROL:*  При циклическом сдвиге разрядов операнда влево его старший выдвигаемый бит вдвигается в операнд справа и становится одновременно значением младшего бита операнда и флага переноса CF. | * op1 = r/m 8/16/32 * op2 = {число, CL} | **Флаги:**  *ROR:* CF=r OF=?r SF=r ZF=r PF=r AF=? Флаг OF представляет интерес, если сдвиг осуществляется на один разряд. При сдвиге на несколько разрядов его значение не определено. По его значению можно судить об изменении знакового разряда операнда:   * если OF=1, то значение флага CF и значение старшего бита операнда после выполнения сдвига различны; * если OF=0, то значение флага CF и значение старшего бита операнда после выполнения сдвига совпадают.     *ROL:*  CF=r OF=?r SF=r ZF=r PF=r AF=? Флаг OF представляет интерес, если сдвиг осуществляется на один разряд. При сдвиге на несколько разрядов его значение не определено. По его значению можно судить об изменении знакового разряда операнда:   * если OF=1, то значение флага CF и значение старшего бита операнда после выполнения сдвига различны; * если OF=0, то значение флага CF и значение старшего бита операнда после выполнения сдвига совпадают. |
| **RCR, RCL** | | Циклический сдвиг влево / вправо с учетом CF:  CF не только запоминает только что вытолкнутое значение, но и участвует в цикле (похоже на единичную задержку на схеме автомата) |  |  |
| **SAR op1, op2 SAL op1, op2** | | SAR:  Арифметический сдвиг вправо.  Сдвиг всех битов операнда вправо на количество разрядов, указанное операндом количество\_сдвигов, при этом выдвигаемый справа бит становится значением флага переноса CF. По мере сдвига вправо освобождающиеся места заполняются значением знакового разряда.  SAL: - сдвиг всех битов операнда влево на один разряд, при этом выдвигаемый слева бит становится значением флага переноса cf;  - одновременно справа в операнд вдвигается нулевой бит;  - указанные выше два действия повторяются количество раз, равное значению второго операнда. | SAR операнд, количество\_сдвигов | **Флаги**  CF=r OF=?r SF=r ZF=r PF=r AF=?  Флаг CF содержит значение последнего выдвинутого из операнда бита. Команда SAR всегда сбрасывает в ноль флаг OF в операциях сдвига на один разряд. |



## Команды работы с битами

Работать с отдельными битами операндов можно, используя логические операции и сдвиги. Однако кроме них в системе команд x86 существуют специальные команды для работы с битами: это команды сканирования битов и команды проверки (и модификации) битов.

|  |  |  |  |
| --- | --- | --- | --- |
| **КОМАНДА** | **ОПИСАНИЕ** | **СИНТАКСИС** | **Комментарий** |
| Команды сканирования битов | | | |
| **BSF op1, op2**  **BSR op1, op2** | Поиск самого правого (самого левого) бита-единицы.  Команда *BSF*  *(BSR*) просматривает биты *op2* от младшего (т.е. самого правого) к старшему (BSR - от старшего к младшему) и помещает индекс [в обоих случаях отсчитываемый от младшего бита] первого единичного бита в регистр. Биты нумеруются, начиная с нуля. Если единичный бит найден, то флаг нуля сбрасывается (ZF=0). Если все биты нулевые, то флаг нуля устанавливается (ZF=1), а значение первого операнда будет неопределённым (на разных процессорах может быть по-разному). | * op1 = r 16/32 * op2 = r/m 16/32 |  |
| Команды проверки и модификации битов | | | |
| **BT op1, op2** | Копирует из слова *op1*  значение проверяемого бита (номер бита – *op2*) в флаг CF.  После этого можно выполнить условный переход командами JC или JNC, в зависимости от значения бита. | * op1 = r/m 16/32 * op2 = r/imm 16/32 | Bite Test |
| **BTR** | Копирует значение бита в флаг CF и затем сбрасывает его. | Эти команды удобны тем, что можно совместить проверку бита и присвоение ему нового значения. |
| **BTS** | Копирует значение бита в флаг CF и затем устанавливает его в 1 |
| **BTC** | Копирует значение бита в флаг CF и затем инвертирует его |

## Организация стека

|  |  |  |  |
| --- | --- | --- | --- |
| **КОМАНДА** | **ОПИСАНИЕ** | **СИНТАКСИС** | **Комментарий** |
| **PUSH op1** | Помещение в стек. Команда уменьшает значение регистра-указателя стека ESP на 4 и затем записывает значение источника в вершину стека:  sub esp, 4  mov [esp], op1 | - r/m/imm |  |
| **POP op1** | Извлечение из стека.  Команда восстанавливает содержимое вершины стека в регистр, ячейку памяти или сегментный регистр, после чего значение ESP увеличивается на 4:  mov op1, [esp]  add esp, 4 | - r/m | Недопустимо восстановление значения в сегментный регистр CS. |
| **LEAVE** | Эквивалентна применению 2-х команд:  ­­MOV ESP, EBP; (уничтожили фрейм)  POP EBP; вернулись на место стека, откуда функция была вызвана. |  |  |
| **CALL func\_name** | В стек заносится адрес возврата и производится передача управления | -r/m/imm (имя метки / непосредственно адрес в памяти) |  |
| **RET** | Pop'ит с вершины стека 4-байта и записывает их в регистр EIP. Трактует их как адрес возврата (передает управление по этому адресу) |  | return |
| **RET op1** | Относится к соглашению *stdcall*, заключающемуся в том, что вызываемая функция сама снимает со стека свои аргументы. Делается это просто:  ret num; где num = 4 \* число\_аргументов |  | Как следствие, соглашение *stdcall* не позволяет реализовывать функции с переменным числом параметров, передаваемых по значению через стек. |

## Организация стекового фрейма функции.

Опр.: Фрейм – блок памяти на стеке, содержащий данный данного вызова функции.

*ИМХО: Стек стоит представлять в голове как подвешенные к потолку грузики. Каждый новый грузик подвешивается, естественно, за низ самого нижнего из уже висящих.*

*esp* указывает на вершину стека (самый низковисящий грузик).

*esp + smth* – движемся вглубь стека (к потолку).

*esp – smth* – свободное место между стеком и кучей.

*Над стековым фреймом функции:*

|  |  |
| --- | --- |
|  | аргументы, передаваемые фикции f из вызывающей функции |

*Стековый фрейм функции:*

|  |  |
| --- | --- |
| ret | ибо был произведен CALL  (в некоторых учебниках не включается в фрейм) |
| ebp | *Стандартный пролог функции:*  push ebp  mov ebp, esp |
| Сохраненные регистры (ebx, esi, edi) |  |
| Локальные переменные + аргументы для вызова функцией следующей функции |  |
| …. |  |
|  | *Стандартный эпилог функции:*  mov esp, ebp; чтобы вернуться в то положение стека, в котором были при вызове функции  pop ebp; мало ли вызывающей функции это значение пригодится  ret  -------------  эквивалентно:  leave  ret |

\*каждая «строка» в таблице весит 4 байта

//рекурсивные вызовы подчиняются правилу last in - first out => для выделения памяти и работы с ней логично использовать стек.

*Возвращаемое значение по CDECL:*

my\_type f(…)

|  |  |
| --- | --- |
| **sizeof(my\_type)** | **Возвращаемой функцией f значение будет храниться в:** |
| 4 (2, 1) | EAX (AX, AL) |
| 8 | EDX:EAX |
| my\_type – это структура | Первым (неявным) параметром, переданным в функцию f из вызывающей функции, будет адрес начала фрагмента памяти, в который нужно записать структуру |

*fastcall*:

arg1->ecx

arg2->edx;

arg3...argn -> stack

*stdcall*:

Вызываемая функция сама снимает со стека свои аргументы (подробнее – см. операцию ret op1 в табл. выше).

*fomit-frame-pointer* (англ. omit – упускать):

Все обращения к содержимому стека выполняются только через регистр ESP.

С точки зрения эффективности такой подход дает два преимущества:

* исчезает необходимость сохранять и восстанавливать EBP в прологе и эпилоге функции;
* у компилятора появляется дополнительный регистр для организации вычислений.

## Регистр EFLAGS

Выполнение арифметических операций сопровождается выработкой флагов *регистра EFLAGS*:

**Флаги статуса:**

* **SF** – знак результата (иными словами, значение крайнего левого бита) – *sign flag*;
* **ZF** – результат равен нулю – *zero flag*;
* **OF** – переполнение (в знаковой трактовке числа – (перенос *на* знаковый бит) xor CF) – *overflow flag*;
* **CF** – перенос за разрядную сетку (переполнение числа в беззнаковой трактовке – перенос *за* знаковый бит) – *carry flag*;
* **PF** – четность числа единиц в младшем байте результата – *parity flag*.

**Флаг направления:**

* **DF -** флаг направления. Управляет направлением обработки строк данных, поведением цепочных инструкций (MOVS, CMPS, SCAS, LODS, STOS):  
  DF=0 — от младших адресов к старшим,  
  DF=1 — от старших адресов к младшим (для специальных строковых команд).   
  Когда флаг сброшен, при выполнении цепочечной команды происходит автоинкремент адресов источника и приемника. Когда флаг установлен — автодекремент (подробнее – см. ниже табл. "Строковые инструкции").

**Разметка регистра EFLAGS:**



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **OF** | **SF** | **ZF** | **PF** | **CF** |
| **ADD, SUB, NEG** | M | M | M | M | M |
| **INC, DEC** | M | M | M | M |  |
| **IMUL, MUL** | M | – | – | – | M |
| **IDIV, DIV** | – | – | – | – | – |
| **CBW, CWD, CDQ** |  |  |  |  |  |
| **MOV, MOVSX, MOVZX** |  |  |  |  |  |
| **CMP** | M | M | M | M | M |
| **TEST** | 0 | M | M | M | 0 |

«M» инструкция обновляет флаг (сбрасывает или устанавливает); «–» влияние инструкции на флаг не определено ; « » инструкция на флаг не влияет; «0» инструкция сбрасывает флаг.

Некоторые из флагов могут быть изменены специально предназначенными для этой цели инструкциями. Для изменения или проверки группы флагов можно воспользоваться командами:

|  |  |
| --- | --- |
| **LAHF** / **SAHF** | загрузка / сохранение младших 8 битов регистра флагов в регистре **AH**. (Load / Save) |
| **PUSHF** / **POPF** | помещение / извлечение из стека младших 16 битов регистра флагов |
| **PUSHFD** / **POPFD** | помещение / извлечение из стека 32-битного регистра **EFLAGS** |
| **PUSHAD / POPAD** | PUSHAD выгружает все основные регистры на стек, POPAD загружает назад. На значения флагов не влияет. |
| **CLC / STC / CMC** | Сбросить CF в 0 / установить CF в 1 / инвертировать CF |
| **CLD / STD** | Сбросить DF / установить DF |

## Условные переходы (условная передача управления)

•**JMP** –r/m/imm 32 - безусловный переход

•**J*cc*** –imm32 - условный переход. Ниже таблица условий.

|  |  |  |  |
| --- | --- | --- | --- |
| **J*cc*** | **Описание (Переход, если…)** | **Условие перехода** | **Комментарий** |
| **J[буква флага: Z, S, O, C, P]** | Переход, если флаг взведен |  | |  |  |  | | --- | --- | --- | | **Команда** | **Переход, если…** | **Условие перехода** | | **JS** | Отрицательное число (знак равен 1) | SF | | **JNS** | Неотрицательное число | ~SF | | **JNP** |  | ~PF | |
| **JN[буква флага: Z, S, O, C, P]** | Переход, если флаг *не* взведен |  |  |
| **JE / JZ** | Нуль или равно | ZF (т.е. ZF = 1) | *Zero; Equal* |
| **JNE / JNZ** | не нуль или не равно | ~ZF (т.е. ZF = 0) | *Not zero; Not equal* |
| **JG** / JNLE | Больше (знаковые числа) | ~(SF^OF)&~ZF | *Greater* |
| **JGE** / JNL | Больше либо равно (знаковые числа) | ~(SF^OF) | *Greater or equal* |
| **JL** / JNGE | Меньше (знаковые числа) | (SF^OF) | *Lower* |
| **JLE** / JNG | Меньше либо равно (знаковые числа) | (SF^OF)|ZF | *Lower or equal* |
| **JA** / JNBE | Больше / не меньше и не равно (числа без знака) | ~CF&~ZF | *Above / Not (below or equal)* |
| **JNA** / JBE | Не больше / меньше или равно(числа без знака) |  |  |
| **JB** | Меньше (числа без знака) | CF | *Below*   |  |  |  | | --- | --- | --- | | **Команда** | **Переход, если…** | **Условие перехода** | | JC/JNAE/JB | есть переполнение/не выше и не равно/ниже | CF=1 | | JNC/JAE/JNB | нет переполнения/выше или равно/не ниже | CF=0 | |
| **JCXZ / JECXZ** | Содержимое *CX / ECX* равно нулю | CX = 0 / ECX = 0 | jump (if) cx / ecx (is) zero |

|  |  |  |  |
| --- | --- | --- | --- |
| **КОМАНДА** | **ОПИСАНИЕ** | **СИНТАКСИС** | **Комментарий** |
| **LOOP .mark** | Декрементирует регистр регистр ECX и *после этого* сравнивает его с нулем. Если не равен нулю — переход след. итерацию (на метку .mark), если равен нулю — идем по коду дальше. |  | т.е. присваиваются значения True / False  Команды LOOP/LOOP*cc* являются макросам и в этом их главная проблема:  Условный переход может быть только в пределах [ –128, 127] байтов от текущей позиции в коде. Закодировать цикл с достаточно объемным телом не получится |
| **LOOPZ / LOOPE**  **( LOOPNZ / LOOPNE)** | Переход выполняется, если значение регистра ECX не равно нулю и флаг ZF установлен (сброшен). |  |  |

## Условная передача данных

|  |  |  |  |
| --- | --- | --- | --- |
| **КОМАНДА** | **ОПИСАНИЕ** | **СИНТАКСИС** | **Комментарий** |
| **SET*cc*  op1** | Помещает в свой единственный операнд либо 0, либо 1, в зависимости от выполнения условия, указанного в суффиксе кода операции. | - op1 = r/m 8 | т.е. присваиваются значения True / False |
| **CMOV*cc* op1, op2** | if (*cc)*{  mov op1, op2;  } | CMOV*cc* r / m, r / m  (Нельзя константу перемещать. Нельзя память-память) | CMOV - Condition MOV |

## Строковые инструкции

|  |  |  |  |
| --- | --- | --- | --- |
| **КОМАНДА** | **ОПИСАНИЕ** | **СИНТАКСИС** | **Комментарий** |
| **cmps{b, w, d}** | compare [esi] and [edi] |  | eSi – source  eDi - destination |
| **lods{b, w, d}** | {AL, AX, EAX} := [esi]  esi += sizeof( {b, w, d} ) |  |  |
| **movs/movs{b, w, d}** | Пересылка данных память-память.  Конкретнее: пересылка элементов двух последовательностей (цепочек) - строк байтов/слов/двойных слов - в памяти.  [edi] := [esi] | movs приемник, источник  movsb  movsw  movsd | (MOVe String Byte/Word/Double word)  выполнение команды не влияет на флаги |
| **scas{b, w, d}** | compare {AL, AX, EAX} and [edi] |  |  |
| **stos{b, w, d}** | [edi] := {AL, AX, EAX}  edi += sizeof( {b, w, d} ) |  |  |
| **cld**  **std** | cld => DF := 0  std => DF := 1 |  | esi = esi + delta  edi = edi + delta  delta = +/- {1, 2, 4}  DF = 0 => + (просмотр строки слева-направо)  DF = 1 => - (просмотр строки в обратном порядке (справа-налево) ) |
| **rep op1**  **repe (<=> repz) op1**  **repne (<=> repnz) op1** | (**REP**eat string operation) - Повторить цепочечную операцию  Алгоритм работы зависит от конкретного префикса. Префиксы rep, repe и repz на самом деле имеют одинаковый код операции, их действия зависят от той цепочечной команды, которую они предваряют:   * *rep* используется перед следующими цепочечными командами и их краткими эквивалентами: *movs, stos, ins, outs*.  Действия rep: анализ содержимого cx:   while (cx != 0) {  op1  dec cx  }   * + если cx != 0, то     - выполнить цепочечную команду, следующую за данным префиксом;     - dec cx;     - повторить цикл.   + если cx=0, выйти из цикла. * *repe* и *repz* используются перед следующими цепочечными командами и их краткими эквивалентами: *cmps, scas*.  Действия repe и repz: анализ содержимого cx и флага zf:   while (cx != 0 || zf != 0) {  op1  dec cx //не влияет на флаги?  }   * + если cx != 0 или zf != 0, то     - выполнить цепочечную команду, следующую за данным префиксом;     - dec cx;     - повторить цикл.   + если cx=0 или zf=0, то выйти из цикла. * *repne* и *repnz* отличаются от *repe* и *repz* лишь тем,   что проверяют zf на противоположное условие:  while (cx != 0 || zf == 0) {  op1  dec cx //не влияет на флаги?  }   * + если cx != 0 или zf=0, то     - выполнить цепочечную команду, следующую за данным префиксом;     - dec cx;     - повторить цикл.   + если cx=0 или zf != 0, то выйти из цикла. | - op1 – команда NASM | *Состояние флагов после выполнения команды:*  06  ZF  r  *Применение:*  Команды rep, repe, repz, repne и repnz в силу специфики своей работы называются префиксами. Они имеют смысл только при использовании цепочечных операций, заставляя их циклически выполняться и тем самым без организации внешнего цикла обрабатывать последовательности элементов фиксированной длины. Большинство применяемых префиксов являются условными, то есть они прекращают работу цепочечной команды при выполнении определенных условий. |

Алгоритм работы:

* выполнить копирование байта, слова или двойного слова из операнда источника в операнд приемник, при этом адреса элементов предварительно должны быть загружены:
  + адрес источника — в пару регистров ds:esi/si (ds по умолчанию, допускается замена сегмента);
  + адрес приемника — в пару регистров es:edi/di (замена сегмента не допускается);
* в зависимости от состояния флага df изменить значение регистров esi/si и edi/di:
  + если df=0, то увеличить содержимое этих регистров на длину структурного элемента последовательности;
  + если df=1, то уменьшить содержимое этих регистров на длину структурного элемента последовательности;
* если есть префикс повторения, то выполнить определяемые им действия (см. команду rep).

**Применение:**

Команды пересылают элемент из одной ячейки памяти в другую. Размеры пересылаемых элементов зависят от применяемой команды. Команда movs может работать с элементами размером в байт, слово, двойное слово. В качестве операндов в команде указываются идентификаторы последовательностей этих элементов в памяти. *Реально эти идентификаторы используются лишь для получения типов элементов последовательностей, а их адреса должны быть предварительно загружены в указанные выше пары регистров.* Транслятор, обработав команду movs и выяснив тип операндов, генерирует одну из машинных команд movsb, movsw или movsd. Машинного аналога для команды movs нет. Для адресации операнда приемник обязательно должен использоваться регистр es.

Для того чтобы эти команды можно было использовать для пересылки последовательности элементов, имеющих размерность байт, слово, двойное слово, необходимо использовать префикс rep. Префикс rep заставляет циклически выполняться команды пересылки до тех пор, пока содержимое регистра ecx/cx не станет равным нулю.

str1 db 'str1 копируется в str2'

len\_str1=$-str1

a\_str1 dd str1

str2 db len\_str1 dup (' ')

a\_str2 dd str2

...

mov cx,len\_str1

lds si,str1

les di,str2

cld

rep movsb

## Числа с плавающей точкой. Сопроцессор x87.

|  |  |
| --- | --- |
| **Размер числа с плавающей точкой (бит)** | **Соответствующая директива объявления переменной** |
| 32 | dd |
| 64 | dq |
| 80 | dt (TWORD) |

**"Стек" сопроцессора x87:** регистры ST0 ... ST7, где ST0 - вершина "стека".

Все команды x87 имеют префикс F.

Суффикс P указывает на то, что при выполнении команды на верхушке стеке освобождается один регистр и все имена оставшихся регистров «сдвигаются» описанным выше образом.

|  |  |  |  |
| --- | --- | --- | --- |
| **КОМАНДА** | **ОПИСАНИЕ** | **СИНТАКСИС** | **Комментарий** |
| **FINIT** | Инициализация сопроцессора x87 |  | Необходимо выполнить перед началом использования сопроцессора. |
| **FLD op1** | Загружает в стек сопроцессора op1 | - m 32/64/80 | **F**loat **L**oa**D** |
| **FLD1**  **FLDZ**  **FLDPI** | Заносят в стек сопроцессора соответственно числа 1.0, 0.0, pi. |  |  |
| **FST op1**  **FSTP op1** | FST - копирует содержимое верхушки стека в память (op1)  FSTP: FST + pop | - m 32/64/80 |  |
| **FADDP**  **FSUBP**  **FMULP**  **FDIVP**  **FABS** | извлекает из стека операнды и кладет в ST0 результат операции  sub: ST1-ST0 |  |  |
| **fucomi op1** | compare st0 and op1 | - регистр x87 |  |

## Еще команды процессора x86:

|  |  |  |  |
| --- | --- | --- | --- |
| **КОМАНДА** | **ОПИСАНИЕ** | **СИНТАКСИС** | **Комментарий** |
| **int op1** | инициирование прерывания | - op1 - 16-ричный код прерывания. | int - interrupt |
| **iter** (семейство команд) | возврат из обработчика прерывания |  |  |
| **HLT** | Команда приостановки процессора до тех пор, пока не возникнет аппаратное прерывание. Когда процессор находится в стадии исполнения этой команды, он потребляет минимальное количество энергии, и, следовательно, почти что не выделяет тепла. |  | от англ. halt — остановка |
| **NOP** | Пустая команда. Инструкция процессора на языке ассемблера, которая предписывает ничего не делать. |  | сокращение от «No OPeration» |
| **RDTSC** | Ассемблерная инструкция для платформы x86, читающая счётчик TSC (Time Stamp Counter) и возвращающая в регистрах EDX:EAX 64-битное количество тактов с момента последнего сброса процессора. |  | сокращение от «**R**ea**d** **T**ime **S**tamp **C**ounter» |

## Средства ввода/вывода io.inc

|  |  |
| --- | --- |
| **Имя макроса** | **Описание макроса** |
| **Output** | |
| **PRINT**\_UDEC *size, data*  PRINT\_DEC *size, data* | Вывод числовых данных заданных параметром *data* в 10-чном представлении. Параметр *size* – число, указывающее размерность данность в байтах; допускаются значения 1, 2, 4. В качестве параметра *data* может выступать числовая константа, символьная константа, имя переменной, имя регистра или адресное выражение (без спецификатора размера данных в памяти). Если задается регистр большего размера, то берется заданное параметром *size* количество младших разрядов. PRINT\_UDEC интерпретирует число как беззнаковое, PRINT\_DEC — как знаковое. |
| PRINT\_HEX *size, data* | Аналогично предыдущему, но данные выводятся в 16-чном представлении. |
| PRINT\_CHAR *ch* | Печатается символ, заданный параметром *ch*. В качестве параметра может выступать численная константа, символьная константа, имя переменной, имя регистра или адресное выражение (без спецификатора размера данных в памяти). Печатается всегда содержимое 8 младших разрядов. |
| PRINT\_STRING *data* | Печать строки текста, оканчивающейся символом с кодом 0. В качестве параметра можно передавать строковую константу, имя переменной или адресное выражение (без спецификатора размера данных в памяти). В случае печати строковой константы, наличие символа с кодом 0 в конце строки необязательно. |
| **NEWLINE** | Переводит печать на новую строку. |
| **Input** | |
| **GET**\_UDEC *size*, *data*  GET\_DEC *size*, *data* | Ввод числовых данных в 10-чном представлении с клавиатуры. Размер вводимых данных ограничен параметром *size*, который задается числом (**1, 2, 4**). Введенные данные обрезаются соответствующим образом. Параметр *data*– либо имя переменной, либо имя регистра, либо адресное выражение (без спецификатора размера данных в памяти). Если задается регистр большего размера, то старшие разряды заполняются знаковым битом в случае GET\_DEC и нулями в случае GET\_UDEC. GET\_UDEC считывает беззнаковое число, GET\_DEC — знаковое. Запрещается использовать в качестве параметра регистр esp. |
| GET\_HEX *size, data* | Аналогично предыдущему, но данные задаются в 16-чном представлении с префиксом 0x. |
| GET\_CHAR *data* | Аналогично предыдущему, но происходит считывание одного символа, нажатие Enter не требуется. Более того, нажатие Enter будет расцениваться как ввод управляющих символов перевода строки: 0xD 0xA в ОС Windows, 0xA в ОС \*nix. Если параметр – регистр, размер которого больше 1 байта, значение считанного символа будет дополнено нулями. |
| GET\_STRING *data, maxsz* | Ввод последовательности символов длиной не более чем *(maxsz-1)*. Чтение последовательности останавливается на EOF или переводе строки, причем перевод строки сохраняется в буфере. В конец считанной строки добавляется символ с кодом 0. Параметр *data* – либо имя переменной, либо адресное выражение (без спецификатора размера данных в памяти). Параметр *maxsz* –регистр (io.inc версия 1.1) или числовая константа. |
| **Вызов библиотечных функций языка СИ** | |
| CEXTERN cfunc\_name |  |

# Источники

(упорядочены в порядке уменьшения вклада в данный документ)

* Материалы лекций (слайды можно скачать на <http://asmcourse.cs.msu.ru/> ).
* <http://asmworld.ru/spravochnik-komand/>
* <http://www.itsusu.ru/pdf/information_technology/work_11.pdf>
* <http://www.kolasc.net.ru/cdo/programmes/assembler/>
* А. В. Столяров. Программирование на языке ассемблера NASM для ОС Unix. Учебное пособие: <http://www.stolyarov.info/books/asm_unix>
* Сайт Сергея Бугаева (студента 103 группы по состоянию на 2014-2015 учебный год): <https://bugaevc.github.io/asmwall/>   
  Обращаю внимание: названия команд на сайте кликабельны.
* <http://pastebin.com/J7d7R6EF>