Краткие сведения о 32-битных командах ассемблера

**Оглавление**

[1.](#_heading=h.gjdgxs) Регистры 3

[1.1.](#_heading=h.30j0zll) Общего назначения 3

[1.2.](#_heading=h.1fob9te) Специального назначения 3

[2.](#_heading=h.3znysh7) Флаги 4

[3.](#_heading=h.2et92p0) Команды присваивания 5

[4.](#_heading=h.tyjcwt) Арифметические команды 6

[5.](#_heading=h.3dy6vkm) Типы данных 8

[5.1.](#_heading=h.1t3h5sf) Основные типы. 8

[5.2.](#_heading=h.4d34og8) Команды преобразования типов 8

[6.](#_heading=h.2s8eyo1) Битовые команды 9

[7.](#_heading=h.17dp8vu) Условные переходы 11

[7.1.](#_heading=h.3rdcrjn) Команды сравнения 11

[7.2.](#_heading=h.26in1rg) Команды переходов 11

[8.](#_heading=h.lnxbz9) Массивы и адресация 13

[9.](#_heading=h.35nkun2) Циклы 14

[9.1.](#_heading=h.1ksv4uv) Команды петли 14

[9.2.](#_heading=h.44sinio) Стандартные циклы 14

[10.](#_heading=h.2jxsxqh) Стек 16

[10.1.](#_heading=h.z337ya) Команды стека 16

[10.2.](#_heading=h.3j2qqm3) Кадр стека 17

[11.](#_heading=h.1y810tw) Вызов функций 21

[11.1.](#_heading=h.4i7ojhp) Команды работы с функциями 21

[11.2.](#_heading=h.2xcytpi) Основные соглашения вызова 21

[11.3.](#_heading=h.1ci93xb) Выравнивание данных. 22

# Регистры

## Общего назначения

|  |  |  |
| --- | --- | --- |
| *Обозначение* | *Расшифровка* | *Специализация* |
| EAX | accumulator | Хранение промежуточных данных |
| EBX | base | Хранение адреса |
| ECX | count | Работа с циклом |
| EDX | data | Хранение дополнительных данных |
| ESI | source index | Адрес источника |
| EDI | destination index | Адрес приёмника |
| ESP | stack pointer | Указатель на вершину стека |
| EBP | base pointer | Копия указателя на вершину стека |

Разбиение регистров:

EAX/EBX/ECX/EDX

|  |  |  |  |
| --- | --- | --- | --- |
| EzX (32 бита) | | | |
|  | | zX (16 бит) | |
|  |  | zH (8 бит) | zL (8 бит) |

Прочие:

|  |  |
| --- | --- |
| Ezy (32 бита) | |
|  | zy (16 бит) |

## Специального назначения

|  |  |  |
| --- | --- | --- |
| *Обозначение* | *Расшифровка* | *Специализация* |
| EIP | instruction pointer | Хранение адреса следующей команды |
| FLAGS | flags | Хранение флагов |

# Флаги

|  |  |  |  |
| --- | --- | --- | --- |
| *Обозначение* | *VS* | *Расшифровка* | *Специализация* |
| CF | CY | carry flag  (флаг переноса) | Устанавливается, если был перенос из старшего бита результата. |
| AF | AC | auxiliary carry flag  (флаг вспомогательного переноса) | Устанавливается при работе при переносе в старшую тетраду или заёме из старшей тетрады.  **Примечание**: используется неявно в командах нормализации двоично-десятичных чисел. |
| PF | PE | parity flag  (флаг паритета) | Устанавливается, если 8 младших разрядов содержат чётное число единиц.  **Примечание**: в современных программах обычно не используется. |
| ZF | ZR | zero flag  (флаг нуля) | Устанавливается при нулевом результате. |
| SF | PL | sign flag  (флаг знака) | Равен старшему биту результата. |
| OF | OV | overflow flag  (флаг переполнения) | Устанавливается, если был перенос в знаковый бит.  Устанавливается, если был заём из знакового бита. |

# Команды присваивания

R — аргумент-регистр;

M — аргумент-память;

I — аргумент-константа (immediate);

L — аргумент-метка.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *Обозначение* | *Расшифровка* | *Аргументы* | | *Результат* | *Описание* | *Влияние на флаги* |
| Arg1 | Arg2 |
| MOV | move (переместить) | R  M | R/M/I  R/I | Arg1 = Arg2 | Присваивание значения, если размеры Arg1 и Arg2 равны. | нет |
| MOVZX | move with zero-extend (переместить с расширением нулями) | R  M | R/M/I  R/I | Arg1 = Arg2 | Присваивание беззнакового значения, если размер Arg1 больше размера Arg2. | нет |
| MOVSX | move with sign-extend (переместить с расширением знаком) | R  M | R/M/I  R/I | Arg1 = Arg2 | Присваивание знакового значения, если размер Arg1 больше размера Arg2. | нет |
| XCHG | exchange  (обменять) | R  M | R/M  R | swap(&Arg1, &Arg2) | Поменять значения местами. | нет |

# Арифметические команды

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *Обозначение* | *Расшифровка* | *Аргументы* | | | *Результат* | *Описание* | *Влияние на флаги* |
| Arg1 | Arg2 | Arg3 |
| ADD | add  (добавить) | R  M | R/M/I  R/I |  | Arg1 += Arg2 | Прибавление. | OF, SF, ZF, AF, CF, PF согласно результату. |
| ADC | add with carry  (добавить c переносом) | R  M | R/M/I  R/I |  | Arg1 += (Arg2 + CF) | Прибавление с переносом. | OF, SF, ZF, AF, CF, PF согласно результату. |
| INC | increment  (инкремент) | R/M |  |  | ++Arg1 | Прибавить 1. | CF не изменяется.  OF, SF, ZF, AF, PF согласно результату. |
| SUB | subtract  (вычесть) | R  M | R/M/I  R/I |  | Arg1 -= Arg2 | Вычитание. | OF, SF, ZF, AF, CF, PF согласно результату. |
| SBB | subtract with borrow  (вычесть с заёмом) | R  M | R/M/I  R/I |  | Arg1 -= (Arg2 + CF) | Вычитание с заёмом. | OF, SF, ZF, AF, CF, PF согласно результату. |
| DEC | decrement  (декремент) | R/M |  |  | --Arg1 | Вычесть 1. | CF не изменяется.  OF, SF, ZF, AF, PF согласно результату. |
| MUL | unsigned multiply  (беззнаково умножить) | R |  |  | EDX:EAX = AEX \* Arg1 | Беззнаковое умножение. | OF, CF = 1, если используются два регистра для результата.  SF, ZF, AF, PF не определены. |
| IMUL | signed multiply  (знаково умножить) | R  R  R | R/M  R/M | I | EDX:EAX = AEX \* Arg1  Arg1 \*= Arg2  Arg1 = Arg2 \* Arg3 | Знаковое умножение: EDX имеет правильные знаковые биты, если результат занимает один регистр. | OF, CF = 1, если используются два регистра для результата (или не влезает в один регистр).  SF, ZF, AF, PF не определены. |
| DIV | unsigned divide  (беззнаково разделить) | R/M |  |  | eax = edx:eax / Arg1  edx = edx:eax % Arg1 | Беззнаковое целочисленное деление с остатком.  **Внимание**: используйте CDQ для автоматического заполнения EDX. | OF, SF, ZF, AF, CF, PF не определены. |
| IDIV | signed divide  (знаково разделить) | R/M |  |  | eax = edx:eax / Arg1  edx = edx:eax % Arg1 | Знаковое целочисленное деление с остатком. Остаток сохраняет знак делимого.  **Внимание**: используйте CDQ для автоматического заполнения EDX. | OF, SF, ZF, AF, CF, PF не определены. |
| NEG | negate  (отрицать) | R/M |  |  | Arg1 = 0 – Arg1 | Изменение знака. | CF = 0, если Arg1 = 0.  OF, SF, ZF, AF, PF согласно результату. |

Примечание.

Если делитель равен нулю, то вызывается прерывание #DE.

Если частное не помещается в регистр EAX, то вызывается прерывание #DE.

Беззнаковый случай. Пусть делитель равен n, edx = a \* n + b, частное edx:eax = (a \* n + b) \* 2^32 + eax.

Далим на n: a \* 2^32 + (b \* 2^32 + aex)/n.

Если a != 0, то частное будет больше 2^32 и произойдёт исключение.

# Типы данных

## Основные типы.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *Название* | *Сокращение* | *Бит* | *Байт* | *Аналог С* | *unsigned* | *signed* | |
| *max* | *min* | *max* |
| byte  (байт) | db | 8 | 1 | char | 255 | -128 | 127 |
| word  (слово) | dw | 16 | 2 | short | 65 535 | -32 768 | 32 767 |
| dword  (двойное слово) | dd | 32 | 4 | int | 4 294 967 295 | -2 147 483 648 | 2 147 483 647 |
| qword  (четверное слово) | dq | 64 | 8 | long long | 18 446 744 073 709 551 615 | -9 223 372 036 854 775 808 | 9 223 372 036 854 775 807 |

## Команды преобразования типов

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Обозначение* | *Расшифровка* | *Результат* | *Описание* | *Влияние на флаги* |
|
| CBW | convert byte to word  (превратить байт в слово) | AX = AL | Расширяет данные с сохранением знака. | нет |
| CWDE | convert word to double  (превратить слово в двойное слово) | EAX = AX | Расширяет данные с сохранением знака. | нет |
| CWD | convert word to double  (превратить слово в двойное слово) | DX:AX = AX | Расширяет данные с сохранением знака на два регистра. | нет |
| CDQ | convert double to quarter  (превратить двойное слово в четверное) | EDX:EAX = EAX | Расширяет данные с сохранением знака на два регистра. | нет |

# Битовые команды

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *Обозначение* | *Расшифровка* | *Аргументы* | | | *Результат* | *Описание* | *Влияние на флаги* |
| Arg1 | Arg2 | Arg3 |
| AND | AND  (И) | R  M | R/M/I  R/I |  | Arg1 &= Arg2 | Побитовая конъюнкция. | OF, CF = 0.  SF, ZF, PF согласно результату.  AF не определён. |
| OR | OR  (ИЛИ) | R  M | R/M/I  R/I |  | Arg1 |= Arg2 | Побитовая дизъюнкция. | OF, CF = 0.  SF, ZF, PF согласно результату.  AF не определён. |
| XOR | exclusive OR  (ЛИБО) | R  M | R/M/I  R/I |  | Arg1 ^= Arg2 | Побитовое не-равно (исключающее ИЛИ).  **Примечание**:  XOR Arg1, Arg1 — быстрое обнуление. | OF, CF = 0.  SF, ZF, PF согласно результату.  AF не определён. |
| NOT | NOT  (НЕ) | R/M |  |  | Arg1 = ~Arg1 | Побитовое отрицание. | нет |
| SHL | shift logical left  (логический сдвиг влево) | R/M  R/M | I/CL |  | Arg1 << 1  Arg1 << Arg2 | Логический сдвиг влево.  Беззнаковое умножение на степень двойки.  Младшие биты заполняются нулями.  То же, что SAL. | CF содержит последний сдвинутый бит.  OF = 1, если при единичном сдвиге у Arg1 старшие два бита были равны; иначе не определён.  SF, ZF, PF согласно результату, если Arg2 != 0. |
| SHR | shift logical right  (логический сдвиг вправо) | R/M  R/M | I/CL |  | Arg1 >> 1  Arg1 >> Arg2 | Логический сдвиг вправо.  Беззнаковое деление на степень двойки.  Старшие биты заполняются нулями. | CF содержит последний сдвинутый бит.  OF содержит старший бит Arg1 при единичном сдвиге; иначе не определён.  SF, ZF, PF согласно результату, если Arg2 != 0. |
| SAL | shift arithmetic left  (арифметический сдвиг влево) | R/M  R/M | I/CL |  | Arg1 << 1  Arg1 << Arg2 | Логический сдвиг влево.  Беззнаковое умножение на степень двойки.  Младшие биты заполняются нулями.  То же, что SHL. | CF содержит последний сдвинутый бит.  OF = 1, если при единичном сдвиге у Arg1 старшие два бита были равны; иначе не определён.  SF, ZF, PF согласно результату, если Arg2 != 0. |
| SAR | shift arithmetic right  (арифметический сдвиг вправо) | R/M  R/M | I/CL |  |  | Логический сдвиг вправо.  Знаковое деление на степень двойки с округлением вниз.  Старшие биты заполняются нулями, при этом сохраняется знаковый бит. | CF содержит последний сдвинутый бит.  OF = 0 при единичном сдвиге; иначе не определён.  SF, ZF, PF согласно результату, если Arg2 != 0. |
| ROL | rotate left  (повернуть влево) | R/M  R/M | I/CL |  |  | Циклический сдвиг влево.  Сдвигаемые влево биты записываются справа. | CF содержит последний сдвинутый бит. |
| ROR | rotate right  (повернуть вправо) | R/M  R/M | I/CL |  |  | Циклический сдвиг вправо.  Сдвигаемые вправо биты записываются слева. | CF содержит последний сдвинутый бит. |

## Таблица бит

|  |  |  |
| --- | --- | --- |
| *10-ая* | *16-ая* | *2-ая* |
| 0 | 0 | 0000 |
| 1 | 1 | 0001 |
| 2 | 2 | 0010 |
| 3 | 3 | 0011 |
| 4 | 4 | 0100 |
| 5 | 5 | 0101 |
| 6 | 6 | 0110 |
| 7 | 7 | 0111 |
| 8 | 8 | 1000 |
| 9 | 9 | 1001 |
| 10 | A | 1010 |
| 11 | B | 1011 |
| 12 | C | 1100 |
| 13 | D | 1101 |
| 14 | E | 1110 |
| 15 | F | 1111 |

# Условные переходы

## Команды сравнения

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *Обозначение* | *Расшифровка* | *Аргументы* | | *Результат* | *Описание* | *Влияние на флаги* |
| Arg1 | Arg2 |
| CMP | compare  (сравнить) | R  M | R/M/I  R/I | (Arg1 - Arg2) | Делает вычитание (SUB), но не записывает результат, а изменяет только флаги. | OF, SF, ZF, AF, CF, PF согласно результату. |
| TEST | logical compare  (логически сравнить) | R  M | R/M/I  R/I | (Arg1 & Arg2) | Делает конъюнкцию (AND), но не записывает результат, а изменяет только флаги. | OF, CF = 0.  SF, ZF, PF согласно результату.  AF не определён. |

## Команды переходов

|  |  |  |
| --- | --- | --- |
| *Обозначение* | *Расшифровка* | *Значение* |
| A | above (выше) | > (беззнаковое) |
| B | below (ниже) | < (беззнаковое) |
| C | carry (перенос) | CF == 1 |
| CX | count (счётчик) | CX |
| E | equal (равно) | == |
| ECX | count (счётчик) | ECX |
| G | greater (больше) | > (знаковое) |
| J | jump if (прыгнуть если) | if |
| JMP | jump (прыгнуть) | goto |
| L | less (меньше) | < (знаковое) |
| N | not (не) | ! |
| O | overflow (переполнение) | OF == 1 |
| P | parity (паритет) | PF == 1 |
| S | sign (знак) | SZ == 1 |
| Z | zero (нуль) | ZF == 1 |

Примеры:

JNGE = jump if not greater or equal.

JECXZ = jump if ECX is zero.

|  |  |
| --- | --- |
| if (A x B)  {  *// Ветка then.*  }  *// Конец if.* | CMP A, B  JNx EndIf1*; отрицание условия*  *; Ветка then.*  EndIf1:  *; Конец if.* |
| if (A x B)  {  *// Ветка then.*  }  else  {  *// Ветка else.*  }  *// Конец if.* | CMP A, B  JNx Else1*; отрицание условия*  *; Ветка then.*  JMP EndIf1  Else1:  *; Ветка else.*  EndIf1:  *; Конец if.* |

Например,

if (eax <= ebx)

сконвертируется в

CMP eax, ebx

JNLE …; <= - это LE, с отрицанием: NLE.

if (!(eax == 0))

сконвертируется в

CMP eax, ebx

JZ …; == 0 - это Z, ! – это NZ, с отрицанием: Z.

# Массивы и адресация

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *Обозначение* | *Расшифровка* | *Аргументы* | | *Результат* | *Описание* | *Влияние на флаги* |
| Arg1 | Arg2 |
| LEA | load effective address  (загрузить эффективный адрес) | R | M | Arg1 = &Arg2 | Сохраняет указатель на область памяти. | нет |

Разыменование делается при помощи оператора [ ].

Примеры: [eax], ebx[2], edx[ecx], [ecx + edx], eax[ecx \* 4].

Явное приведение типа указателя делается при помощи оператора ptr:

dword ptr [EAX], byte ptr [ecx + edx].

Приведение типов нужно, чтобы избежать неоднозначности, если она есть.

# Циклы

## Команды петли

**Внимание!** Данные команды считаются устаревшими, так как требуют много времени на декодирование и не оптимизируются современными процессорами. Работают почти в два раза медленнее, чем прямая реализация.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *Обозначение* | *Расшифровка* | *Аргументы* | | *Результат* | *Описание* | *Влияние на флаги* |
| Arg1 | Arg2 |
| LOOP | loop  (петля) | Label |  | --ECX;  if (ECX != 0)  goto Arg1; | Уменьшает счётчик.  Если счётчик не равен нулю, повторяет итерацию. | нет |
| LOOPE  LOOPZ | loop while equal/zero  (петля по равенству) | Label |  | --ECX;  if (ECX != 0 && ZF == 1)  goto Arg1; | Уменьшает счётчик.  Если счётчик не равен нулю и аргументы равны (аргумент равен нулю), повторяет итерацию. | нет |
| LOOPNE  LOOPNZ | loop while not equal/zero  (петля по неравенству) | Label |  | --ECX;  if (ECX != 0 && ZF == 0)  goto Arg1; | Уменьшает счётчик.  Если счётчик не равен нулю и аргументы не равны (аргумент не равен нулю), повторяет итерацию. | нет |

## Стандартные циклы

**Внимание!** Требуется обязательно проверить счётчик на нуль (JECXZ) перед входом в петлю, если не гарантируется, что количество итераций положительно.

|  |  |
| --- | --- |
| while (A x B)  {  *// Тело цикла.*  }  *// Конец цикла.* | WhileBegin1:  CMP A, B;  JNx WhileEnd1;  *; Тело цикла*  JMP WhileBegin1;  WhileEnd1:  *; Конец цикла.* |
| do  {  *// Тело цикла.*  } while (A x B);  *// Конец цикла.* | DoWhile1:  *; Тело цикла*  CMP A, B;  Jx DoWhile1;  *; Конец цикла.* |
| for (int i = N; i > 0; --i)  {  *// Тело цикла.*  }  *// Конец цикла.* | MOV ECX, N;  JECXZ ForEnd1*; Случай N == 0.*  ForBegin1:  *; Тело цикла.*  LOOP ForBegin1;  ForEnd1:  *; Конец цикла.* |
| for (int i = 0; i < N; ++i)  {  *// Тело цикла.*  }  *// Конец цикла.* | XOR ecx, ecx*; Быстрое обнуление*  ForBegin1:  CMP ecx, N*; Лучше поместить N в регистр.*  JE ForEnd1;  *; Тело цикла.*  INC ecx;  JMP ForBegin1;  ForEnd1:  *; Конец цикла.* |

# Стек

## Команды стека

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *Обозначение* | *Расшифровка* | *Аргументы* | | *Результат* | *Описание* | *Влияние на флаги* |
| Arg1 | Arg2 |
| PUSH | push  (затолкнуть) | R/M/I |  | SUB ESP, 4  MOV [ESP], Arg1 | Заталкивает аргумент в стек. | нет |
| POP | pop  (вытолкнуть) | R/M |  | MOV Arg1, [ESP]  ADD ESP, 4 | Выталкивает значение из стека и записывает его в Arg1. | нет |

esp *указывает* на текущую вершину стека и последнее значение, затолкнутое в стек.

Пример:

PUSH eax*; AEX == 7*

PUSH ebx*; EBX == 6*

PUSH ecx*; ECX == 5*

В стеке адреса **убывают**:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Свободная память (мусор)  ← | | | Вершина стека | Данные стека  → | | |
| Адреса | … | ESP - 8 | ESP - 4 | esp  ↓ | ESP + 4 | ESP + 8 | … |
| Значения |  |  |  | 5 | 6 | 7 | … |

Значение из стека получается по адресу:

MOV EAX, [ESP + 4]*; EAX == 6*

## Кадр стека

Пусть в функцию переданы некоторые аргументы через стек:

int f(int x, int y, int z) { … }

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Вершина стека | Данные стека  → | | |
| Адреса | esp  ↓ | ESP + 4 | ESP + 8 | ESP + 12 |
| Значения | (адрес возврата) | x | y | z |

При вызове функции всегда передаётся адрес возврата, чтобы после выполнения функции можно было продолжить исполняемый код.

Тогда к аргументам можно обращаться через регистр ESP.

Однако применение команды PUSH меняет адреса аргументов:

PUSH 10;

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Вершина стека | Данные стека  → | | | |
| Адреса | esp  ↓ | ESP + 4 | ESP + 8 | ESP + 12 | ESP + 16 |
| Значения | 10 | (адрес возврата) | x | y | z |

Контроль точного смещения адресов может быть сложен при активном использовании стека в длинных функциях. Поэтому рекомендуется зафиксировать смещение при помощи регистра EBP.

PUSH EBP*; Сохраняем предыдущее значение EBP*

MOV EBP, ESP*; Фиксируем смещение стека*

Теперь мы имеем доступ к аргументам функции через регистр EBP:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Вспомогательные данные | | Аргументы функции | | |
| Адреса | eBp  ↓ | EBP + 4 | EBP + 8 | EBP + 12 | EBP + 16 |
| Значения | EBP  (старое) | (адрес возврата) | x | y | z |

Более того, такой подход позволяет легко выделять память под локальные переменные.

Пусть наша функция выглядит так:

int f(int x, int y, int z)

{

int a, b;

…

}

Локальные переменные требуют 8 байт. Сместим вершину стека на 8 байт:

SUB ESP, 8*; Останутся 8 неиспользованных байт*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Локальные переменные | | Вспомогательные данные | | Аргументы функции | | |
| Адреса | EBP - 8 | EBP - 4 | eBp  ↓ | EBP + 4 | EBP + 8 | EBP + 12 | EBP + 16 |
| esp  ↓ | ESP + 4 | ESP + 8 | ESP + 12 | ESP + 16 | ESP + 20 | ESP + 24 |
| Значения | a | b | EBP  (старое) | (адрес возврата) | x | y | z |

Такое расположение данных называется *кадром стека*.

Инициализация кадра всегда проходит при помощи следующих начальных строчек функции, называемых *прологом*:

PUSH EBP;

MOV EBP, ESP;

SUB ESP, N*; где N — размер локальных переменных в байтах.*

В конце функции идёт уничтожение кадра стека, этот код называется *эпилогом*:

MOV ESP, EBP*; «Очистка» памяти локальных переменных*

POP EBP*; Возврат сохранённого значения EBP.*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *Обозначение* | *Расшифровка* | *Аргументы* | | *Результат* | *Описание* | *Влияние на флаги* |
| Arg1 | Arg2 |
| ENTER | enter  (зайти) | I | I | PUSH EBP;  MOV EBP, ESP;  SUB ESP, Arg1; | Создаёт кадр стека размера Arg1 уровня вложенности Arg2.  **Устаревшее.** Занимает слишком много времени на расшифровку и не оптимизируется современными процессорами. | нет |
| LEAVE | leave  (покинуть) |  |  | MOV ESP, EBP;  POP EBP; | Освобождает кадр стека. | нет |

**Внимание!** Изменение значения EBP разрушает кадр стека. Перед изменением это значение должно быть сохранено. После изменения и до восстановления невозможно пользоваться локальными переменными и аргументами через EBP.

PUSH EBP;

MOV EBP, 1*; Разрушение кадра стека*

… *; Здесь нельзя использовать локальные переменные и аргументы. Считается, что всё нужное сохранено в регистрах.*

POP EBP*; Восстановление кадра стека. Вновь можно использовать локальную память.*

Конечно, всегда можно схитрить и использовать ESP, но не перехитрите сами себя.

**Внимание!** Изменение значения ESP разрушает стек. Вы должны понимать, что делаете! Значение должно быть сохранено в память. После изменения и до восстановления пользоваться всеми командами стека невозможно.

MOV a, ESP*; a — память, например [EBP - 4].*

… *; Команды стека недоступны;*

MOV ESP, a*; Восстановление стека.*

Несмотря на то, что стек разрушен, вы всё ещё можете пользовать локальными переменными и аргументами, если создали кадр стека.

ESP можно использовать как обычный регистр, или же перенаправить на другой участок памяти, чтобы использовать в экзотических случаях.

# Вызов функций

## Команды работы с функциями

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *Обозначение* | *Расшифровка* | *Аргументы* | | *Результат* | *Описание* | *Влияние на флаги* |
| Arg1 | Arg2 |
| CALL | call  (вызвать) | Label |  | PUSH EIP + 2  JMP Arg1  … *; EIP + 2* | Заталкивает адрес возврата в стек и переходит к выполнению заданной функции. | нет |
| RET | return  (вернуть) | I |  | POP EIP  ADD ESP, Arg1 | *То же, что* RET *0.*  Выталкивает значение в EIP и освобождает Arg1 байт стека. | нет |

Вызов функции происходит обычно из пяти этапов, часть из которых необязательна:

1. Передача аргументов
2. Переход на начало функции
3. Возврат
4. Получение результата
5. Очистка стека.

Для согласования работы стека между разными языками, используют *соглашения вызова*.

## Основные соглашения вызова

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Ключевое слово* | *Передача аргументов* | *Очистка стека* | *Хранение результата* | *Сохраняемые значения регистров* |
| \_\_stdcall | Справа налево через стек | Вызываемой функцией | 1 байт — AL  2 байта — AX  4 байта — EAX  8 байт — EAX:EDX | EBX, EDI, ESI, EBP, ESP |
| \_\_cdecl | Справа налево через стек | Вызывающим кодом |
| \_\_fastcall | Первые два — через ECX и EDX.  Остальные — справа налево через стек. | Вызываемой функцией |

## Выравнивание данных.

|  |  |
| --- | --- |
| *Тип* | *Байт в стеке* |
| char | 4 |
| short | 4 |
| int | 4 |
| long long | 8 |

Примеры.

|  |  |  |  |
| --- | --- | --- | --- |
| *Реализация* | | *Вызов* | *Примечания* |
| \_\_stdcall void f1( |  | PUSH w |  |
| int x, | [ESP + 4] | PUSH z |  |
| int y, | [ESP + 8] | PUSH y |  |
| short z, | [ESP + 12] | PUSH x |  |
| short w) | [ESP + 16] | CALL f1 |  |
| { return; } | RET 16 |  | Очистка стека со стороны вызванной функции. |
|  |
| \_\_cdecl void f2( |  | PUSH w |  |
| int x, | [ESP + 4] | PUSH z |  |
| char y, | [ESP + 8] | PUSH y |  |
| long long z, | [ESP + 12] | PUSH x |  |
| char w) | [ESP + 20] | CALL f2 |  |
| { return; } | RET | ADD ESP, 20 | Очистка стека со стороны вызывающего кода. |
|  |
| \_\_fastcall void f3( |  | MOV ECX, x |  |
| int x, | ECX | MOV ECX, y |  |
| int y, | EDX | PUSH w |  |
| char z, | [ESP + 4] | PUSH z |  |
| char w) | [ESP + 8] | CALL f3 |  |
| { return; } | RET 8 |  | Очистка стека со стороны вызванной функции. |

# Цепочечные операции

## Флаги

|  |  |  |
| --- | --- | --- |
| *Обозначение* | *Расшифровка* | *Специализация* |
| DF | direction flag  (флаг направления) | Если DF == 0, то направление цепочечных команд слева направо (восходящий адрес).  Если DF == 1, то направление справа налево (нисходящий адрес). |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Обозначение* | *Расшифровка* | *Результат* | *Описание* | *Влияние на флаги* |
|
| CLD | clear direction flag  (очистить флаг направления) | DF = 0 | Задаёт направление обработки цепочечных операций слева направо. | DF = 0 |
| STD | set direction flag  (установить флаг направления) | DF = 1 | Задаёт направление обработки цепочечных операций справа налево. | DF = 1 |

## Команды

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *Суффикс* | *Расшифровка* | *Смещение* | *Регистры* | | | *Указатель* |
| \*B | byte | 1 | ESI | EDI | AL | byte ptr |
| \*W | word | 2 | ESI | EDI | AX | word ptr |
| \*D | dword | 4 | ESI | EDI | EAX | dword ptr |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Обозначение* | *Расшифровка* | *Результат* | *Описание* | *Влияние на флаги* |
|
| MOVSB  MOVSW  MOVSD | move string  (переместить строку) | sign = (DF == 0)? +1 : -1;  MOV [EDI], x ptr [ESI]  ADD EDI, sign \* 1/2/4  ADD ESI, sign \* 1/2/4 | Перемещает элемент размера 1/2/4 из адреса ESI по адресу EDI.  Смещает адреса ESI и EDI на 1/2/4 вперёд/назад в зависимости от DF. | Нет. |
| CMPSB  CMPSW  CMPSD | compare strings  (сравнить строки) | sign = (DF == 0)? +1 : -1;  CMP [EDI], x ptr [ESI]  ADD EDI, sign \* 1/2/4  ADD ESI, sign \* 1/2/4 | Сравнивает элементы размера 1/2/4 по адресу ESI и по адресу EDI.  Смещает адреса ESI и EDI на 1/2/4 вперёд/назад в зависимости от DF. | OF, SF, ZF, AF, CF, PF согласно результату CMP. |
| LODSB  LODSW  LODSD | load string  (загрузить строку) | sign = (DF == 0)? +1 : -1;  MOV AL/AX/EAX, [ESI]  ADD ESI, sign \* 1/2/4 | Загружает элемент размера 1/2/4 из адреса ESI в регистр AL/AX/EAX.  Смещает адрес ESI на 1/2/4 вперёд/назад в зависимости от DF. | Нет. |
| SCASB  SCASW  SCASD | scan string  (просмотреть строку) | sign = (DF == 0)? +1 : -1;  CMP [EDI], AL/AX/EAX  ADD EDI, sign \* 1/2/4 | Сравнивает элемент размера 1/2/4 по адресу EDI и из регистра AL/AX/EAX.  Смещает адрес EDI на 1/2/4 вперёд/назад в зависимости от DF. | OF, SF, ZF, AF, CF, PF согласно результату CMP. |
| STOSB  STOSW  STOSD | store string  (сохранить строку) | sign = (DF == 0)? +1 : -1;  MOV [EDI], AL/AX/EAX  ADD EDI, sign \* 1/2/4 | Сохраняет элемент размера 1/2/4 из регистра AL/AX/EAX по адресу EDI.  Смещает адрес EDI на 1/2/4 вперёд/назад в зависимости от DF. | Нет. |

## Примечание: обратите внимание, что приведённые команды работают только с одним элементом за раз. Для обработки строки/массива в цикле используйте префиксы.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Обозначение* | *Расшифровка* | *Аргументы* | *Результат* | *Описание* | *Влияние на флаги* |
| Arg1 |
| REP | move string  (переместить строку) | MOVS  STOS | while (ECX != 0)  {  Arg1();  --ECX;  } | Пока счётчик не ноль:   * выполняет Arg1; * уменьшает счётчик; * повторяет итерацию. | Согласно команде Arg1. |
| REPE  REPZ | repeat while equal/zero  (повторять пока равны) | CMPS  SCAS | while (ECX != 0)  {  Arg1();  --ECX;  if (EF == 0) break;  } | Пока счётчик не ноль:   * выполняет Arg1; * уменьшает счётчик; * повторяет итерацию, пока текущие элементы равны. | Согласно команде Arg1. |
| REPNE  REPNZ | repeat while not equal/zero  (повторять пока равны) | CMPS  SCAS | while (ECX != 0)  {  Arg1();  --ECX;  if (EF != 0) break;  } | Пока счётчик не ноль:   * выполняет Arg1; * уменьшает счётчик; * повторяет итерацию, пока текущие элементы не равны. | Согласно команде Arg1. |

Чтобы уточнить причину выхода из цикла, используйте условный переход JECXZ.

**Внимание!** Установите правильное значение DF перед выполнением цикла.

**Внимание!** Выход по условию происходит после сдвига адресов, поэтому для получения элементов, по которому произошёл выход, нужно сдвинуть адреса обратно.