КОНСТРУИРОВАНИЕ ПРОГРАММ

Лекция № 08 базовые инструкции IA32

+375 17 293 8039 (505a-5)

+375 17 320 7402 (ОИПИ НАНБ)

prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by/

Кафедра ЭВМ, 2022

2022.03.30

Оглавление

Базовые инструкции IA32

- инструкции перемещения данных;
- арифметические инструкции (двоичная и двоично-десятичная арифметика);
- логические инструкции;
- инструкции сдвигов;
- инструкции для работы с битами и байтами;
- инструкции для работы со строками;
- инструкции передачи управления;
- инструкции ввода/вывода;
- инструкции управления флагами;
- инструкции для работы с сегментными регистрами;
- другие инструкции.

Инструкции перемещения данных (основные)

Назначение — перемещение данных между памятью, регистрами общего назначения и сегментными регистрами.

Также выполняют условное перемещение, доступ к стеку и преобразование данных.

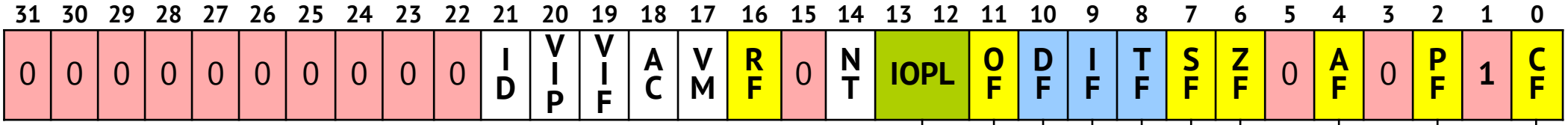
MOV — перемещение данных между памятью, РОН и сегментными регистрами
CMOVcc — условное перемещение данных
XCHG — обмен операндов
BSWAP — перестановка байт
PUSH — поместить в стек
POP — извлечь из стека
PUSHA/PUSHAD — поместить все РОН в стек
POPA/POPAD — извлечь все РОН из стека
CBW/CWD/CDQ — преобразовать
CWDE — преобразовать в EAX
MOVSX/MOVZX — переместить с расширением/без расширения знака

Флагов не изменяют

CMPXCHG — сравнить и обменять
XADD — обменять и сложить

Изменяют флаги

Регистр флагов



IOPL – привилегии ввода/вывода

OF – переполнение

DF – флаг направления (STD, CLD)

IF – маска прерываний (CLI, STI)

TF – пошаговый режим

SF – знак результата

ZF – нулевой результат

AF – BCD перенос

PF – бит четности

CF – перенос (STC, CLC, CMC)

Обозначения в схемах адресации инструкций (325462-074US- V2-3.1.1.3)

r — регистр;
r/m — регистр или память (EA) размером 8, 16, 32, 64 ит;
r/m16 — 16-битный регистр или полуслово в памяти (16 бит);
Sreg — сегментный (селекторный) регистр;
imm — непосредственное значение;
moffsN — операнд задан только смещением без SIB, N — размер данных.

```
MOV     r/m, r         ; 8 ← 8, 16 ← 16, 32 ← 32, 64 ← 64
MOV     r, r/m         ; 8 ← 8, 16 ← 16, 32 ← 32, 64 ← 64
MOV     r/m16, Sreg    ; r/m16 ← Sreg
MOV     Sreg, r/m16    ; Sreg ← r/m16
MOV     r/m64, Sreg    ; r/m64 ← SignExtend(Sreg)
MOV     Sreg, r/m64    ; Sreg ← (r/m64)
MOV     r,  imm        ; 8 ← 8, 16 ← 16, 32 ← 32, 64 ← 64
MOV     r/m, imm       ; 8 ← 8, 16 ← 16, 32 ← 32, 64 ← 64
MOV     AL, moffs8     ; AL ← [byte seg:offset]
MOV64   AL, moffs8     ; AL ← [byte offset]
MOV     AX, moffs16    ; AX ← [word seg:offset]
MOV     EAX, moffs32   ; EAX ← [dword seg:offset]
MOV     RAX, moffs64   ; RAX ← [qword offset]
MOV     moffs8, AL     ; [byte seg:offset] ← AL
MOV64   moffs8, AL     ; [byte offset]← AL
MOV     moffs16, AX    ; [word seg:offset] ← AX
MOV     moffs32, EAX   ; [dword seg:offset] ← EAX
MOV     moffs64, RAX   ; [qword offset] ← RAX
```

Конкретный ассемблер не обязательно будет распознавать и генерировать все из указанных выше режимов.

Даже более, не факт, что компоновщик сможет собрать исполняемый модуль из объектных модулей, сгенерированных ассемблером.

```
 1                                  global _start, code_start
 2
 3                                  section     .data
 4 00000000 717765727479            str1        db 'qwerty'
 5 00000006 0000000000000000                    db 8 dup 0
 6 0000000E 617364666768            str2        db 'asdfgh'
 7
 8                                  section     .text
 9                                  _start:
10                                  code_start:
11 00000000 3EA0[0E000000]              mov     al,  [ds:str2]
12 00000006 3EA0[0E000000]              mov     al,  ds:str2
13 0000000C A0[0E000000]                mov     al,  [str2]
14 00000011 B0[0E]                      mov     al,  str2
15 00000013 3E6667A1[0E00]              mov     ax,  [word ds:str2]
16 00000019 3E66A1[0E000000]            mov     ax,  word ds:str2
17 00000020 3EA1[0E000000]              mov     eax, [dword ds:str2]
18 00000026 3EA1[0E000000]              mov     eax, dword ds:str2
19 0000002C 8EC8                        mov     cs, ax
```

test.asm:14:(.text+0x12): relocation truncated to fit: R_386_8 against `.data'

test.asm:15:(.text+0x17): relocation truncated to fit: R_386_16 against `.data'

`MOV` — перемещение данных

Перемещает данные между регистрами общего назначения, между памятью и регистрами общего назначения или сегментными регистрами, а также пересылают непосредственные операнды в регистры общего назначения.

Синтаксис

```
MOV     r/m, r          ; 8 ← 8, 16 ← 16, 32 ← 32, 64 ← 64
MOV     r, r/m          ; 8 ← 8, 16 ← 16, 32 ← 32, 64 ← 64
MOV     r/m16, Sreg     ; r/m16 ← Sreg
MOV     Sreg, r/m16     ; Sreg ← r/m16
MOV     r/m64, Sreg     ; r/m64 ← SignExtend(Sreg)
MOV     Sreg, r/m64     ; Sreg ← (r/m64)
MOV     r,  imm         ; 8 ← 8, 16 ← 16, 32 ← 32, 64 ← 64
MOV     r/m, imm        ; 8 ← 8, 16 ← 16, 32 ← 32, 64 ← 64
MOV     AL, moffs8      ; AL ← [byte seg:offset]
MOV64   AL, moffs8      ; AL ← [byte offset]
MOV     AX, moffs16     ; AX ← [word seg:offset]
MOV     EAX, moffs32    ; EAX ← [dword seg:offset]
MOV     RAX, moffs64    ; RAX ← [qword offset]
MOV     moffs8, AL      ; [byte seg:offset] ← AL
MOV64   moffs8, AL      ; [byte offset]← AL
MOV     moffs16, AX     ; [word seg:offset] ← AX
MOV     moffs32, EAX    ; [dword seg:offset] ← EAX
MOV     moffs64, RAX    ; [qword offset] ← RAX
```

ЗАМЕЧАНИЕ:

Операнды **`moffs8`**, **`moffs16`**, **`moffs32`** и **`moffs64`** задают простое смещение (без базирования и индексирования) относительно базы сегмента, где 8, 16, 32 и 64 относятся к размеру данных. Размер смещения 16, 32 или 64 бит определяется атрибутом инструкции **`address-size`**.

Описание

Инструкция MOV копирует второй операнд (операнд-источник) в первый операнд (операнд-назначение).

Операндом-источником может быть:

- непосредственное значение;
- регистр общего назначения;
- регистр сегмента;
- ячейка памяти.

Операндом-назначением может быть:

- регистр общего назначения;
- регистр сегмента;
- ячейка памяти.

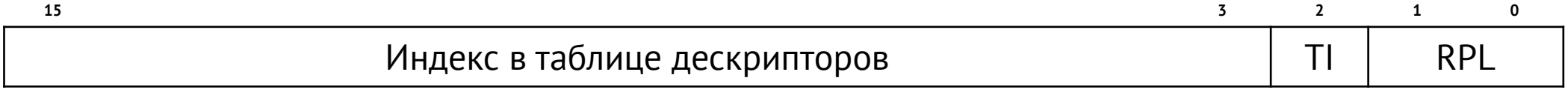
Оба операнда должны иметь одинаковый размер, который может быть байтом, словом, двойным словом или квадрословом.

Загрузка сегментных регистров DS, SS, ES, FS, GS

Инструкция `MOV` не может быть использована для загрузки регистра `CS`.

Попытка сделать это приводит к исключению недопустимого кода операции (**#UD**). Для загрузки регистра CS следует использовать дальнюю инструкцию **JMP**, **CALL** или **RET**.

Если операндом-назначением является регистр сегмента (**DS**, **ES**, **FS**, **GS** или **SS**), операндом-источником должен быть действительный селектор сегмента.



В защищенном режиме перемещение селектора сегмента в регистр сегмента автоматически приводит к загрузке информации дескриптора сегмента, связанной с этим селектором сегмента, в скрытую (теневую) часть регистра сегмента.

Информация о селекторе сегмента и дескрипторе сегмента проверяется при загрузке этой информации (во время исполнения).

Данные дескриптора сегмента получаются из записи **GDT** или **LDT** для указанного селектора сегмента.

Селектор сегмента **NULL** (значения 0000-0003) может быть загружен в регистры **DS**, **ES**, **FS**, **GS** и **SS**, не вызывая исключения защиты. Однако любая последующая попытка сослаться на сегмент, соответствующий регистр сегмента которого загружен со значением **NULL**, вызывает общее исключение защиты (#GP), и ссылки на память не происходит.

Загрузка сегмента стека (переключение стека)

В процессе загрузки регистра SS инструкцией MOV запрещаются все прерывания до тех пор, пока не будет выполнена следующая инструкция (атомарное переключение стека).

Такое поведение позволяет загружать указатель стека в регистр **ESP** с помощью следующей инструкции **MOV ESP, stack-pointer-value** до того, как произойдет прерывание. Однако, в последовательности инструкций, которые загружают регистр **SS**, только первая инструкция гарантированно задерживает прерывание. Например, в следующей последовательности прерывание может случиться раньше **MOV ESP, EBP**:

```
MOV SS, EDX    ; IF <- 0
MOV SS, EAX    ; IF <- 1
MOV ESP, EBP   ; ???
```

Следует иметь в виду, что инструкция **LSS** предлагает более эффективный способ загрузки регистров **SS** и **ESP**.

При работе в 32-битном режиме и перемещении данных между регистром сегмента и регистром общего назначения 32-битные процессоры IA-32 не требуют использования с этой инструкцией 16-битного префикса размера операнда (66H), но большинство ассемблеров ее вставят, если используется стандартная форма инструкции (например, **MOV DS,AX**).

Процессор выполнит эту инструкцию правильно, но обычно для этого потребуется дополнительное время.

Большинство ассемблеров, использующих форму инструкции **MOV DS, EAX** избегает этого ненужного префикса 66H.

```
 1                                  global  _start
 2                                  section .data
 3 00000000 00000000                var0 dd    00000000h
 4 00000004 11111111                var1 dd    11111111h
 5 00000008 22222222                var2 dd    22222222h
 6 0000000C 33333333                var3 dd    33333333h
 7                                  section .text
 8                                  _start:
 9 00000000 B867452301                      mov   eax, 01234567h ; imm
10 00000005 B8[04000000]                    mov   eax, var1      ;  offset
11 0000000A A3[00000000]                    mov   [var0], eax
12 0000000F A1[08000000]                    mov   eax, [var2]
13 00000014 A3[0C000000]                    mov   [var3], eax
14 00000019 66A1[08000000]                  mov   ax, [var2]
15 0000001F 66A3[0C000000]                  mov   [var3], ax
16 00000025 A0[08000000]                    mov   al, [var2]
17 0000002A A2[0C000000]                    mov   [var3], al
18 0000002F 8A25[08000000]                  mov   ah, [var2]
19 00000035 8825[0C000000]                  mov   [var3], ah
20 0000003B 8ED8                            mov   DS, eax
21 0000003D 89D8                            mov   eax, ebx
22 0000003F 8B83[08000000]                  mov   eax, [ebx + var2]
23 00000045 8B84B3[0C000000]                mov   eax, [ebx + var3 + 4*esi]
24 0000004C F7D8                            neg   eax
25 0000004E 8984B3[0C000000]                mov   [ebx + var3 + 4*esi], eax
26 00000055 B910000000                      mov   ecx, 10h
```

MOVSX — переместить с расширением знака (Move & SignExtend)

```
MOVSX    r16, r/m8    ; байт в слово
MOVSX    r32, r/m8    ; байт в двойное слово
MOVSX    r32, r/m16   ; слово в двойное слово
```

Копирует содержимое исходного операнда (регистр или ячейка памяти) в целевой операнд (регистр) и знаково расширяет его значение до 16 или 32 бит.

Размер преобразованного значения зависит от атрибута размер-операнда.

Флагов не изменяет.

MOVZX —переместить с заполнением нулями (Move & ZeroExtend)

```
MOVZX    r16, r/m8  ; байт в слово
MOVZX    r32, r/m8  ; байт в двойное слово
MOVZX    r32, r/m16 ; слово в двойное слово
```

Копирует содержимое исходного операнда (регистр или ячейка памяти) в целевой операнд (регистр) и расширяет нулями его значение до 16 или 32 бит.

Размер преобразованного значения зависит от атрибута размер-операнда.

Флагов не изменяет

CMOVcc — условное перемещение данных

Команды **CMOVcc** проверяют состояние одного или нескольких флагов состояния в регистре **EFLAGS** (**CF**, **OF**, **PF**, **SF** и **ZF**) и выполняют операцию перемещения, если флаги находятся в указанном состоянии (или условие выполнено). Код условия (**cc**) указывает проверяемое условие.

cc	Условие перемещения		
E / Z	**E**qual / **Z**ero	равно / ноль	ZF=1
NE / NZ	**N**ot **E**qual / **N**ot **Z**ero	не равно / не ноль	ZF=0
A / NBE	**A**bove / **N**ot (**B**elow or **E**qual)	выше / не (ниже или равно)	CF=0 & ZF=0
AE / NB	**A**bove or **E**qual / **N**ot **B**elow	выше или равно / не ниже	CF=0
B / NAE	**B**elow / **N**ot (**A**bove or **E**qual)	ниже/ не (выше или равно)	CF=1
BE / NA	**B**elow or **E**qual / **N**ot **A**bove	ниже или равно / не выше	CF=1 | ZF=1
G / NLE	**G**reater / **N**ot (**L**ess or **E**qual)	больше / не (меньше или равно)	ZF=0 & SF=OF
GE / NL	**G**reater or **E**qual / **N**ot **L**ess	больше или равно / не меньше	SF=OF
L / NGE	**L**ess / **N**ot **G**reater or **E**qual	меньше / не (больше или равно)	SF≠OF
LE / NG	**L**ess or **E**qual / **N**ot **G**reater	меньше или равно / не больше	ZF=1 | SF≠OF
C	**C**arry	перенос	CF=1
NC	**N**o **C**arry	отсутствие переноса	CF=0
O	**O**verflow	переполнение	OF=1
NO	**N**o **O**verflow	отсутствие переполнения	OF=0
S	**S**ign (negative)	знак	SF=1
NS	**N**o **S**ign (non-negative)	отсутствие знака	SF=0
P/PE	**P**arity / **P**arity **E**ven	паритет / четно	PF=1
NP/PO	**N**o **P**arity / **P**arity **O**dd	отсутствие паритета / нечетно	PF=0

Если условие не выполняется, перемещение не выполняется, при этом выполнение продолжается с инструкции, следующей за **CMOVcc**.

```
    cmp     esi, edi ; сравним операнды
    cmovz   eax, ebx ; и переместим ebx -> eax если esi == edi
```

До P6

```
    cmp     esi, edi ; сравним операнды
    jne     .L       ; пропустим команду, если не равны
    mov     eax, ebx ; переместим ebx -> eax если esi == edi
.L:
```

Эти инструкции могут перемещать 16-битные, 32-битные или 64-битные значения из памяти в регистр общего назначения или из одного регистра общего назначения в другой.

```
   cmovz   eax, ebx  ; 0F44C3
   cmovz   ebx, [c]  ; 0F441D[06000000]
*  cmovz   [d], ecx  ; error: invalid combination of opcode and operands
```

Условные перемещения 8-битных операндов регистров не поддерживаются.

Условие для каждой мнемоники **CMOVcc** приведено в таблице. Термины «меньше» и «больше» используются для сравнения целых чисел со знаком, а термины «выше» и «ниже» используются для целых чисел без знака.

Поскольку некоторое состояние флагов состояния иногда можно интерпретировать двумя способами, для некоторых кодов операций определены две мнемоники. Код при этом генерирутся один и тот же.

Инструкции **CMOVcc** были введены в процессорах семейства P6, однако эти инструкции могут поддерживаться не всеми процессорами IA-32. Программное обеспечение может определить, поддерживаются ли инструкции **CMOVcc**, путем проверки информации о функциях процессора с помощью инструкции **CPUID**.

Флаги не изменяет.

BSWAP — (Byte swap) Переставляет порядок байт в регистре

```
bswap     r32
```

Описание

Меняет порядок байтов 32-битного (целевого) регистра. Эта инструкция предназначена для преобразования значений с прямым порядком байтов в формат с обратным порядком байтов и наоборот.

Чтобы поменять местами байты в значении слова (16-битный регистр), следует использовать инструкцию XCHG.

При ссылке **BSWAP** на 16-битный регистр, результат не определен.

Аналоги в C/C++ (Linux)

```
#include <arpa/inet.h>
uint32_t htonl(uint32_t hostlong);  // from host byte order to network byte order
uint16_t htons(uint16_t hostshort); // from host byte order to network byte order
uint32_t ntohl(uint32_t netlong);   // from network byte order to host byte order
uint16_t ntohs(uint16_t netshort);  // from network byte order to host byte order
```

На архитектуре i386 host byte order — это Least Significant Byte first (задом наперед) network byte order, как он используется в Internet, — Most Significant Byte first.

Флагов не изменяет.

XCHG — (Exchange) обмен операндов

```
XCHG    r1, r2
XCHG    r/m, r
```

Описание

Обменивает содержимое целевого (первого) и исходного (второго) операндов.

Операндами могут быть два регистра общего назначения или регистр и ячейка памяти.

Если имеется ссылка на операнд памяти, на время операции обмена автоматически реализуется протокол блокировки процессора независимо от наличия или отсутствия префикса **LOCK** или значения **IOPL**.

Инструкция заменяет три инструкции **MOV** и не требует временного сохранения содержимого операнда, пока загружается другое.

Эта инструкция полезна для реализации семафоров или аналогичных информационных объектов для синхронизации процессов.

Инструкция **XCHG** также может использоваться вместо инструкции **BSWAP** для 16-битных операндов.

Флагов не изменяет.

XADD — Обменять и сложить

```
XADD    r/m, r    ; 8, 16, 32
```

Обменивает первый операнд (операнд-назначение) со вторым операндом (операнд-источник), а затем загружает сумму этих двух значений в операнд-назначение.

Операндом-назначением может быть регистр или ячейка памяти.

Операнд-источник — только регистр.

Эта инструкция может использоваться с префиксом LOCK, чтобы позволить инструкции выполняться атомарно.

Флаги: устанавливаются флаги CF, PF, AF, SF, ZF и OF по результату сложения.

CMPXCHG — Сравнить и обменять

```
CMPXCHG    r/m, r    ; 8, 16, 32
```

Сравнивает значение в регистре **AL**, **AX**, **EAX** с первым операндом (операндом назначения). Если два значения равны, второй операнд (исходный операнд) загружается в операнд назначения. В противном случае операнд назначения загружается в регистр **AL**, **AX**, **EAX**.

Эта инструкция может использоваться с префиксом **LOCK**, чтобы позволить ее атомарное выполнение.

Инструкция используется для синхронизации операций в системах, использующих несколько процессоров, а также для работы с семафорами.

Инструкция **CMPXCHG** обычно используется для тестирования и модификации семафоров. Она проверяет, свободен ли семафор. Если семафор свободен, он помечается как занятый; в противном случае инструкция получает идентификатор текущего владельца. Все это делается за одну непрерывную операцию.

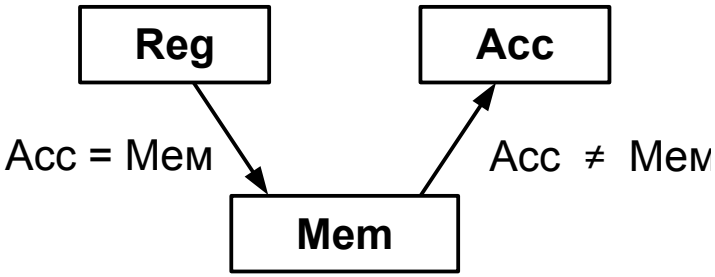
В однопроцессорной системе инструкция CMPXCHG избавляет от необходимости переключаться на уровень защиты 0 (для отключения прерываний) перед выполнением нескольких инструкций для проверки и модификации семафора.

Флаги

Флаг **ZF** устанавливается, если значения операнда назначения и аккумулятора (AL, AX или EAX) равны; в противном случае он очищается.

Флаги **CF**, **PF**, **AF**, **SF** и **OF** устанавливаются по результатам операции сравнения.

CMPXCHG



PUSH — Поместить в стек

```
push    r/m     ; opsize 16, 32, 64
push    r       ; opsize 16, 32, 64
push    imm     ; opsize 16, 32, 64
push    SegR    ; CS, SS, DS, ES, FS, GS (non-64 bit)
```

Уменьшает указатель стека, затем сохраняет операнд-источник на вершине стека.

Размеры адреса и операнда определяются и используются следующим образом:

Размер адреса. Размер адреса по умолчанию определяется флагом **D** в текущем дескрипторе сегмента кода. Умолчание может быть отменено префиксом инструкции (67H).

Размер адреса используется только при обращении к исходному операнду в памяти.

Размер операнда. Размер операнда по умолчанию определяется флагом **D** в текущем дескрипторе сегмента кода. Умолчание может быть отменено префиксом инструкции (67H).

Размер операнда (16, 32 или 64 бита) определяет величину, на которую уменьшается указатель стека (2, 4 или 8 байт).Если непосредственный операнд имеет размер меньше, чем размер операнда, в стек помещается значение, расширенное знаковым битом.

Если исходный операнд является регистром сегмента (16 бит), а размер операнда составляет 64 бита, в стек помещается значение расширенное нулями.

Если же размер операнда составляет 32 бита, в стек помещается либо значение, расширенное нулями, либо селектор сегмента записывается в стек с использованием 16-разрядного перемещения **SP**.

В последнем случае все недавние процессоры Core и Atom выполняют 16-битное перемещение **SP**, оставляя верхнюю часть расположения стека нетронутой.

Размер адреса в стеке. Размер указателя стека (16 или 32 бита) в 32- и 16-битных режимах определяется флагом **B** текущего дескриптора сегмента стека.

В 64-битном режиме размер указателя стека всегда равен 64 битам.

Размер адреса стека определяет ширину указателя стека при записи в стек в памяти, а также при декременте указателя стека. (Как указано выше, величина, на которую уменьшается указатель стека, определяется размером операнда.)

Если размер операнда меньше размера адреса в стеке, инструкция **PUSH** может привести к некорректно выровненному указателю стека (указателю стека, который не выровнен по границе двойного слова или по границе четырех слов).

Инструкция **PUSH ESP** помещает значение регистра **ESP** в том виде, в каком оно существовало до выполнения инструкции. Это имеет место как для архитектуры Intel 64, так и для реальной адресации, а также и режима виртуального 8086 архитектуры IA-32.

Для процессора Intel 8086 инструкция **PUSH SP** помещает в стек новое значение регистра SP, которое стало после декремента на 2.

Если инструкция **PUSH** использует операнд памяти, в котором для вычисления адреса операнда используется регистр **ESP**, адрес операнда вычисляется перед уменьшением регистра **ESP**.

Если регистр **ESP** или **SP** равен 1, когда инструкция **PUSH** выполняется в режиме реальной адресации, генерируется исключение ошибки стека (#SS), поскольку нарушается предел сегмента стека.

Его выдача приводит ко второму исключению ошибки стека по той же причине, вызывая генерацию исключения двойной ошибки (#DF). При выдаче исключения двойной ошибки генерируется третье исключение ошибки стека, и процессор переходит в режим выключения.

POP — Извлечь из стека

```
POP     r/m
POP     r
POP     SegR    ; DS, ES, SS, FS, GS (32-битный режим)
```

Загружает значение из вершины стека в местоположение, указанное с помощью операнда-получателя (или явного кода операции), а затем увеличивает указатель стека.

Операндом-адресатом может быть регистр общего назначения, ячейка памяти или сегментный регистр.

Размеры адреса/операнда инструкций **`push`** и **`pop`** по умолчанию определяются флагом **D** в текущем дескрипторе сегмента кода.

Умолчание может быть отменено префиксом (67h/66h, соотв.).

Размер адреса используется только при обращении к операнду в памяти.

Размер операнда (16, 32 или 64 бита) определяет величину, на которую уменьшается указатель стека (2, 4 или 8 байт).

Размер стекового адреса. Вне 64-битного режима флаг **B** в текущем дескрипторе сегмента стека определяет размер указателя стека (16 или 32 бита). В 64-битном режиме размер указателя стека всегда равен 64 битам.

Размер стекового адреса определяет ширину указателя стека при чтении из стека в память и при увеличении указателя стека. (Как указано выше, величина, на которую увеличивается указатель стека, определяется размером операнда.)

Если операндом-адресатом является один из регистров сегмента **DS**, **ES**, **FS**, **GS** или **SS**, значение, загруженное в регистр, должно быть допустимым селектором сегмента (как для **MOV**).

В защищенном режиме вставка селектора сегмента в регистр сегмента автоматически вызывает загрузку информации дескриптора, связанной с этим селектором сегмента, в скрытую (теневую) часть регистра сегмента и проверку информации селектора и дескриптора.

В регистры **DS**, **ES**, **FS** или **GS** не вызывая общей ошибки защиты может быть занесено значение **NULL** (0000-0003) . Однако любая последующая попытка сослаться на сегмент, соответствующий регистр сегмента которого загружен значением **NULL**, вызывает исключение общей защиты (**#GP**). В этой ситуации ссылки на память не возникает, и сохраненное значение регистра сегмента будет равно **NULL**.

Инструкция **POP** не может вставить значение в регистр **CS**. Чтобы загрузить регистр **CS** из стека, следует использовать инструкцию **RET**.

Инструкция **POP ESP** увеличивает указатель стека (**ESP**) до того, как данные со старой вершины стека будут записаны в место назначения.

Если в качестве базового регистра для адресации операнда назначения в памяти используется регистр **ESP**, инструкция **POP** вычисляет эффективный адрес операнда после того, как он увеличивает регистр **ESP**.

Для случая 16-разрядного стека, если **ESP** оборачивается в **0H** в результате выполнения команды POP, результирующее местоположение записи в память зависит от семейства процессоров.

Инструкция **POP SS** запрещает все прерывания, включая прерывание **NMI**, до выполнения следующей команды.

Это действие позволяет последовательно выполнять команды **POP SS** и **MOV ESP, EBP** без опасности наличия недопустимого стека во время прерывания. Однако более предпочтительным методом загрузки регистров **SS** и **ESP** является использование инструкции **LSS**.

Флагов не изменяет.

PUSHA/PUSHAD — поместить в стек регистры общего назначения

```
PUSHA    ; Push AX, CX, DX, BX, Orig SP, BP, SI, DI
PUSHAD   ; Push EAX, ECX, EDX, EBX, Orig ESP, EBP, ESI, EDI
```

Описание

Помещает содержимое регистров общего назначения в стек. Регистры сохраняются в стеке в следующем порядке:

EAX, **ECX**, **EDX**, **EBX**, **ESP** (исходное значение), **EBP**, **ESI** и **EDI** (если атрибут размера операнда равен 32) и

AX, **CX**, **DX**, **BX**, **SP** (исходное значение), **BP**, **SI** и **DI** (если атрибут размера операнда равен 16).

Эти инструкции выполняют обратную операцию относительно инструкций **POPA**/**POPAD**. Значение, передаваемое для регистра ESP или SP, является его значением до нажатия первого регистра (см. Раздел «Работа» ниже).

Мнемоники **PUSHA** (push all) и **PUSHAD** (push all double) генерируют один и тот же код операции. Инструкция **PUSHA** предназначена для использования, когда атрибут размера операнда равен 16, а инструкция **PUSHAD** для случая, когда атрибут размера операнда равен 32.

Некоторые ассемблеры могут принудительно установить размер операнда до 16 при использовании **PUSHA** и до 32 при использовании **PUSHAD** используя при необходимости префикс переопределения размера операнда [66H]).

Другие могут рассматривать эту мнемонику как синонимы (**PUSHA**/**PUSHAD**) и использовать текущую настройку атрибута размер-операнда, чтобы определить размер значений, помещаемых в стек, независимо от используемой мнемоники.

В режиме реальной адресации если регистр ESP или SP равен 1, 3 или 5, при выполнении **PUSHA**/**PUSHAD** генерируется исключение #SS, но не обрабатывается, поскольку ошибка стека предотвращает обработку #SS. После этого процессор генерирует исключение #DF (двойная ошибка) и входит в состояние выключения.

Эта инструкция выполняется, как описано, в не 64-разрядных режимах.

PUSHA и PUSHAD недопустимы в 64-битном режиме.

Флагов не изменяет.

POPA/POPAD — извлечь из стека регистры общего назначения

```
POPA    ; Pop DI, SI, BP, BX, DX, CX, AX
POPAD   ; Pop EDI, ESI, EBP, EBX, EDX, ECX, EAX
```

Описание

Извлекает двойные слова (**POPAD**) или слова (**POPA**) из стека в регистры общего назначения, обращая действие инструкций **PUSHA** / **PUSHAD**.

Регистры загружаются в следующем порядке:

EDI, **ESI**, **EBP**, **EBX**, **EDX**, **ECX** и **EAX** (если атрибут размера операнда равен 32)
и

DI, **SI**, **BP**, **BX**, **DX**, **CX** и **AX** (если атрибут размера операнда равен 16).

Значение в стеке для регистра **ESP** или **SP** игнорируется. Вместо этого регистр **ESP** или **SP** увеличивается после загрузки каждого регистра.

Мнемоники **POPA** (pop all) и **POPAD** (pop all double) генерируют один и тот же код операции. Инструкция **POPA** предназначена для использования, когда атрибут размера операнда равен 16, а инструкция **POPAD** для случая, когда атрибут размера операнда равен 32.

Некоторые ассемблеры могут принудительно установить размер операнда до 16 при использовании **POPA** и до 32 при использовании **POPAD** используя при необходимости префикс переопределения размера операнда [66H]).

Другие могут рассматривать эту мнемонику как синонимы (**POPA**/**POPAD**) и использовать текущую настройку атрибута размер-операнда, чтобы определить размер значений, извлекаемых из стека, независимо от используемой мнемоники.

(В дескрипторе сегмента текущего сегмента кода есть флаг **D**, котороый определяет атрибут размера операнда.)

Эта инструкция выполняется, как описано выше только в не 64-битных режимах.

POPA и POPAD недопустимы в 64-битном режиме.

Флагов не изменяет.

CBW/CWDE/CDQE — преобразовать в удвоенное с расширением знака

```
CBW    ; AX  ← AL с расширением знака
CWDE   ; EAX ← AX с расширением знака
CDQE   ; RAX ← EAX с расширением знака
```

Удваивают размер исходного операнда с помощью расширения знака.

Инструкция **CBW** (преобразование байта в слово) копирует знак (бит 7) из операнда-источника в AL в каждый бит регистра AH.

Инструкция **CWDE** (преобразование слова в двойное слово) копирует знак (бит 15) из слова в регистре AX в старшие 16 бит регистра EAX.

CBW и **CWDE** ссылаются на один и тот же код операции.

CBW предназначена для использования, когда атрибут размер-операнда равен 16.

CWDE предназначена для использования, когда атрибут размер-операнда равен 32.

Некоторые ассемблеры могут изменять размер операнда.

Другие могут рассматривать эти две мнемоники как синонимы (**CBW**/**CWDE**) и устанавливают атрибут «размер-операнда» для установки размера значений, подлежащих преобразованию.

В 64-битном режиме размер операции по умолчанию равен размеру регистра назначения.

В этом случае CDQE копирует знак (бит 31) двойного слова в регистре EAX в старшие 32 бита RAX.

Флагов не изменяет.

CWD/CDQ/CQO — преобразовать в удвоенное с расширением знака

```
CWD   ; DX:AX   ← AX с расширением знака
CDQ   ; EDX:EAX ← EAX с расширением знака
CQO   ; RDX:RAX ← RAX с расширением знака
```

Удваивают размер операнда, расположенного в регистре AX, EAX или RAX (в зависимости от размера операнда), с помощью расширения знака и сохраняет результат в регистрах DX: AX, EDX: EAX или RDX: RAX соответственно.

CWD копирует знак (бит 15) значения в регистре **AX** в каждую позицию бита в DX.

CDQ копирует знак (бит 31) значения в регистре EAX в каждую битовую позицию в EDX.

Мнемоники CWD и CDQ ссылаются на один и тот же код операции.

CQO (доступна только в 64-битном режиме) копирует знак (бит 63) значения в регистре RAX в каждую битовую позицию в регистре RDX.

Использование

Инструкция CWD может использоваться для получения делимого размером в двойное слово из слова перед делением слова.

Команду CDQ можно использовать для получения делимого размером в четырехбайтное слово из двойного слова перед делением двойного слова.

Инструкция CQO может использоваться для получения делимого размером в пару четырех-байтных слов из четырехбайтных слов перед делением четырехбайтных слов.

Флагов не изменяет.

Двоичные арифметические инструкции

Двоичные арифметические команды выполняют базовые вычисления над двоичными целыми числами в байтах, словах и двойных словах, расположенных в памяти и/или регистрах общего назначения.

ADD — сложение целых;
ADC — сложение целых с переносом;
ADCX — сложение беззнаковых целых с переносом;
ADOX — сложение беззнаковых целых с переполнением;
SUB — вычитание;
SBB — вычитание с заемом (отрицательным переносом);
IMUL — умножение целых со знаком;
MUL — умножение беззнаковых целых;
IDIV — деление целых со знаком;
DIV — деление беззнаковых целых;
INC — инкремент;
DEC — декремент;
NEG — сменить знак;
CMP — арифметическое сравнение.

ADD — сложение целых

```
ADD     AL,     imm8
ADD     AX,     imm16
ADD     EAX,    imm32

ADD     r/m8,   imm8
ADD     r/m16,  imm16
ADD     r/m32,  imm32
ADD     r/m16,  imm8
ADD     r/m32,  imm8
ADD     r/m,    r           ; 8, 16, 32
ADD     r,      r/m         ; 8, 16, 32
```

Операция

```
DEST ← (DEST + SRC)
```

Описание

Выполняет сложение операнда-назначения (первый операнд), операнда-источника (второй операнд) и сохраняет результат в операнде-назначении.

Операндом-назначением может быть регистр или ячейка памяти.

Операндом-источником может быть непосредственное значение, регистр или ячейкой памяти. (Однако два операнда памяти не могут использоваться в одной инструкции.)

В том случае, когда в качестве операнда-источника используется непосредственное значение, оно расширяется до длины формата операнда-назначения.

Инструкция ADD не различает операнды со знаком и без знака.

Вместо этого процессор вычисляет результат для обоих типов данных и устанавливает флаги **CF** и **OF**, указывая наличие переноса и/или переполнения в обоих случаях. Флаг **SF** указывает знак результата для знакового сложения.

Данная инструкция может использоваться с префиксом **LOCK**, чтобы позволить ей выполняться атомарно.

Флаги **OF**, **SF**, **ZF**, **AF**, **CF**, **PF** устанавливаются в соответствии с результатом.

SUB — вычитание

```
SUB     AL, imm8
SUB     AX, imm16
SUB     EAX, imm32

SUB     r/m8, imm8
SUB     r/m16, imm16
SUB     r/m32, imm32
SUB     r/m16, imm8
SUB     r/m32, imm8
SUB     r/m, r       ; 8, 16, 32
SUB     r, r/m       ; 8, 16, 32
```

Операция

```
DEST ← (DEST - SRC)
```

Описание

Выполняет вычитание операнда-источника (второй операнд) из операнда-назначения (первый операнд), и сохраняет результат в операнде-назначении.

Операндом-назначением может быть регистр или ячейка памяти.

Операндом-источником может быть непосредственное значение, регистр или ячейкой памяти. (Однако два операнда памяти не могут использоваться в одной инструкции.)

В том случае, когда в качестве операнда-источника используется непосредственное значение, оно расширяется до длины формата операнда-назначения.

Инструкция **SUB** выполняет целочисленное вычитание, не различая операндов со знаком и без. Вместо этого процессор вычисляет результат для обоих типов данных и устанавливает флаги **CF** и **OF**, указывая наличие переноса и/или переполнения для обоих случаев. Флаг **SF** указывает знак результата для знакового случая.

Данная инструкция может использоваться с префиксом **LOCK**, чтобы позволить ей выполняться атомарно.

Флаги **OF**, **SF**, **ZF**, **AF**, **CF**, **PF** устанавливаются в соответствии с результатом.

MUL — беззнаковое умножение

```
MUL     r/m8          ; AX       ← AL  * r/m8
MUL     r/m16         ; DX:AX    ← AX  * r/m16
MUL     r/m32         ; EDX:EAX  ← EAX * r/m32
```

Описание

Выполняет беззнаковое умножение первого операнда (операнд-приемник) и второго операнда (операнд-источник) и сохраняет результат в операнде-приемнике.

Операндом-приемником является подразумеваемый операнд, расположенный в регистре **AL**, **AX** или **EAX** (в зависимости от размера операнда).

Исходный операнд находится в регистре общего назначения или в ячейке памяти. Действие этой инструкции и расположение результата зависит от кода операции и размера операнда, как показано в таблице.

Размер операнда	операнд-источник 1	операнд-источник 2	операнд-приемник
Байт	AL	r/m8	AX
Слово	AX	r/m16	DX:AX
Двойное слово	EAX	r/m32	EDX:EAX

Старшие биты произведения содержатся в регистре **AH**, **DX** или **EDX** соответственно.

Флаги **CF** и **OF** очищаются если старшие биты произведения равны 0, в противном случае устанавливаются. Флаги **SF**, **ZF**, **AF**, **PF** неопределены.

IMUL — умножение целых со знаком

```
IMUL    r/m8                ; AX ← AL * r/m8
IMUL    r/m16               ; DX:AX ← AX * r/m16
IMUL    r/m32               ; EDX:EAX ← EAX * r/m32

IMUL    r16, r/m16          ; r16 ← r16 * r/m16
IMUL    r32, r/m32          ; r32 ← r32 * r/m32

IMUL    r16, r/m16, imm8    ; r16 ← r/m16 * SignExt(Imm8)
IMUL    r32, r/m32, imm8    ; r32 ← r/m32 * SignExt(Imm8)
IMUL    r16, r/m16, imm16   ; r16 ← r/m16 * Imm16
IMUL    r32, r/m32, imm32   ; r32 ← r/m32 * Imm32
```

Описание

Выполняет знаковое умножение двух операндов. Эта инструкция имеет три формы, в зависимости от количества операндов.

Форма с одним операндом.

Идентична той, которая используется инструкцией **MUL**. Здесь операнд-источник (в регистре общего назначения или в ячейке памяти) умножается на значение в регистре **AL**, **AX** или **EAX** (в зависимости от размера операнда) и произведение (формат которого в два раза больше формата исходного операнда) сохраняется в регистрах **AX**, **DX: AX** или **EDX: EAX**, соответственно.

Форма с двумя операндами.

```
IMUL    r16, r/m16          ; r16 ← r16 * r/m16
IMUL    r32, r/m32          ; r32 ← r32 * r/m32
```

При использовании этой формы операнд-приемник (первый операнд) умножается на операнд-источник (второй операнд).

Операндом- приемником является регистр общего назначения, а операндом-источником является непосредственное значение, регистр общего назначения или ячейка памяти.

Промежуточное произведение (формат которого в общем случае в два раза больше формата исходного операнда) усекается и сохраняется в операнде-приемнике (регистр общего назначения).

Форма с тремя операндами.

```
IMUL    r16, r/m16, imm8  ; r16 ← r/m16 * SignExt(Imm8)
IMUL    r32, r/m32, imm8  ; r32 ← r/m32 * SignExt(Imm8)
IMUL    r16, r/m16, imm16 ; r16 ← r/m16 * Imm16
IMUL    r32, r/m32, imm32 ; r32 ← r/m32 * Imm32
```

Для этой формы требуется операнд-приемник (первый операнд) и два операнда-источника (второй и третий операнды).

Первый операнд-источник (который может быть регистром общего назначения или ячейкой памяти) умножается на второй операнд-источник (непосредственное значение).

Промежуточное произведение, формат которого в общем случае в два раза больше формата первого операнда-источника, усекается и сохраняется в операнде-приемнике (регистр общего назначения).

Когда непосредственное значение используется в качестве операнда, оно расширяется до длины формата операнда-приемника.

Если значение со знаком промежуточного произведения отличается от знакорасширенного произведения, усеченного по размеру операнда, устанавливаются флаги **CF** и **OF**, в противном случае флаги CF и OF очищаются.

Три формы инструкции **IMUL** схожи в том, что в результате длина произведения получается вдвое больше длины исходных операндов.

Произведение сохраняется точным только в форме с одним операндом

В формах с двумя и тремя операндами результат, прежде чем он сохраняется в регистре назначения, усекается до длины операнда-приемника.

Из-за этого усечения необходимо проверять флаг **CF** или **OF**, чтобы быть уверенным, что значимые биты не потеряны.

Формы с двумя и тремя операндами можно использовать и с беззнаковыми операндами, поскольку что младшая половина произведения одинакова вне зависимости от того, являются ли операнды знаковыми или нет.

Однако флаги **CF** и **OF** нельзя использовать для определения того, является ли старшая половина результата ненулевой.

Флаги:

Флаг **SF** обновляется в соответствии с наиболее значимым битом результата усеченного операнда в месте назначения.

Для одноопернадной формы флаги **CF** и **OF** устанавливаются, когда значащие биты переносятся в старшую половину результата, и сбрасываются, когда результат точно совпадает с младшей половиной результата. Для форм с двумя и тремя операндами флаги **CF** и **OF** устанавливаются при усечении результата до размера операнда-приемника, и очищаются, когда результат точно соответствует размеру операнда-приемника.

Флаги **ZF**, **AF** и **PF** не определены.

DIV — беззнаковое деление

```
DIV     r/m8     ; AL  ← Quot, AH  ← Rem (AX/rm8)
DIV     r/m16    ; AX  ← Quot, DX  ← Rem (AX/rm16)
DIV     r/m32    ; EAX ← Quot, EDX ← Rem (EAX/rm32)
```

Описание

Делит значение без знака, содержащееся в **AX**, **DX:AX** или **EDX:EAX** (делимое) на операнд-источник (делитель) и сохраняет результат в регистрах **AX** (**AH:AL**), **DX:AX** или **EDX:EAX**.

Операндом-источником может быть регистр общего назначения или ячейка памяти. Действие этой инструкции зависит от размера операнда (делимое/делитель).

Нецелые результаты усекаются (обрезаются) по направлению к 0.

Остаток всегда меньше делителя по величине.

Переполнение указывается исключением **#DE** (ошибка деления), а не с флагом **CF**.

Размер операнда	Делимое	Делитель	Частное	Остаток	Диапазон частного
Слово/Байт	AX	r/m8	AL	AH	255
Двойное слово/Слово	DX:AX	r/m16	AX	DX	65535
Квадрослово/Двойное слово	EDX:EAX	r/m32	EAX	EDX	2^{32}-1

Флаги **CF**, **OF**, **SF**, **ZF**, **AF**, **PF** не определены

IDIV — знаковое деление целых с остатком

```
IDIV    r/m    ; 8, 16, 32
```

Описание

Делит значение со знаком, содержащееся в **AX**, **DX:AX** или **EDX:EAX** (делимое) на операнд-источник (делитель) и сохраняет результат в регистрах **AX** (**AH:AL**), **DX:AX** или **EDX:EAX**.

Операндом-источником может быть регистр общего назначения или ячейка памяти. Действие этой инструкции зависит от размера операнда (делимое/делитель).

Нецелые результаты усекаются (обрезаются) по направлению к 0.

Остаток всегда меньше делителя по величине.

Переполнение указывается исключением **#DE** (ошибка деления), а не с флагом **CF**.

Размер операнда	Делимое	Делитель	Частное	Остаток	Диапазон частного
Слово/Байт	AX	r/m8	AL	AH	-128 ... +127
Двойное слово/Слово	DX:AX	r/m16	AX	DX	-32768 ... + 32767
Квадрослово/Двойное слово	EDX:EAX	r/m32	EAX	EDX	-2^{31} ... 2^{31}-1

Флаги **CF**, **OF**, **SF**, **ZF**, **AF**, **PF** не определены