01. Регистры общего назначения

Регистры общего назначения

Регистры — специальные ячейки памяти, находящиеся физически внутри процессора, доступ к которым осуществляется не по адресам, а по именам. Поэтому, работают очень быстро.

Существуют регистры, которые могут использоваться без ограничений, для любых целей — регистры общего назначения.

В 8086 регистры 16 битные.

При использовании регистров общего назначения, можно обратится к каждым 8 битам (байту) по-отдельности, используя вместо *X - *H или *L (например, для AX: AH и AL)

AX (аккумулятор)

Регистр часто используется для хранения результата действий, выполняемых над двумя операндами. Например, используется при MUL и DIV (умножение и деление)

Верхние 8 бит (1 байт) — AH

Нижниие 8 бит (1 байт) — AL

BX (база)

Используется для адресации по базе.

Верхние 8 бит (1 байт) — BH

Нижние 8 бит (1 байт) — BL

CX (счётчик)

Используется как счётчик в циклах и строковых операциях.

Верхние 8 бит (1 байт) — CH

Нижние 8 бит (1 байт) — CL

DX (регистр данных)

Если при выполнении действий над двумя операндами, реузльтат не помещается в AX, регистр DX получает старшую часть результата.

Верхние 8 бит (1 байт) — DH

Нижние 8 бит (1 байт) — DL

Этот регистр получает старшую часть данных,

SI (индекс источника) и DI (индекс приемника)

Ещё есть два этих регистра, они называются индексными, то есть используются для индексации в массивах / матрицах и т.д. (другие регистры (кроме BX и BP) не будут там работать (на 8086)).

Могут использоваться в большинстве команд, как регистры общего назначения.

В этих регистрах нельзя обратится к каждому из байтов по-отдельности

02. Сегментные регистры. Адресация в реальном

режиме. Понятие сегментной части адреса и смещения.

Сегментные регистры

- Сегмент кода (CS)
- Сегменты данных (DS, ES, FS, GS)
- Сегмент стека (SS)

Каждый регистр содержит адрес для разных сегментов программы, с помощью них и происходит доступ к этим сегментам.

Мы знаем что регистр IP "указывает" на следующую команду, мы также знаем что регистры у нас размером в 16 бит, таким образом, используя один регистр программист имеет доступ только к 2^16 адресам, это примерно 64 кБ. Это достаточно мало, поэтому адресация в 8086 по 2^20 адресам, то есть примерно 1 МБ памяти. Для этого используют адрес начала сегмента и смещение. Сегментные регистры как раз хранят адрес. Реальный адрес высчитывается так: сегментная_часть × 16 + смещение. (умножение на 16 = сдвиг на четыре бита влево)

При такой адресации адреса 0400h:0001h и 0000h:4001h будут ссылаться на одну и ту же ячейку памяти, так как 400h × 16 + 1 = 0 × 16 + 4001h.

03. Регистры работы со стеком.

Стек

Стек - структура данных, работающая по принципу LIFO (last in, first out) - последним пришёл, первым вышел.

Сегмент стека - область памяти программы, используемая её подпрограммами, а также (вынужденно) обработчиками прерываний.

SP - указатель на вершину стека, ***BP*** - указатель на начало стека. BP используется в подпрограмме для сохранения "начального" значения ***SP***, адресации параметров и локальных переменных.

В ***x86*** стек растёт вниз, в сторону уменьшения адресов. При запуске программы ***SP*** указывает на конец сегмента.

Команды непосредственной работы со стеком

PUSH <источник>

Помещает данные в стек. Уменьшает ***SP*** на размер источника и записывает значение по адресу ***SS:SP***.

POP <приемник>

Считывает данные из стека. Считывает значение с адреса ***SS:SP*** и увеличивает ***SP***.

PUSHA

Помещает в стек регистры ***AX***, ***CX***, ***DX***, ***BX***, ***SP***, ***BP***, ***SI***, ***DI***.

POPA

Загружает регистры из стека (SP игнорируется).

Вызов процедуры и возврат из процедуры

CALL <операнд>

Сохраняет адрес следующей команды в стеке (уменьшает ***SP*** и записывает по его адресу либо IP либо ***CS:IP***, в зависимости от размера аргумента. Передает управление на значение аргумента.

RET/RETN/RETF <число>

Загружает из стека адрес возврата, увеличивает ***SP***. Если указан операнд, его значение будет дополнительно прибавлено к ***SP*** для очистки стека от параметров.

04. Структура программы. Сегменты.

Структура программы

Любая программа состоит из сегментов

/// ! Виды сегментов:

Сегмент кода

Сегмент данных

Сегмент стека

/// ! Описание сегмента в исходном коде:

имя **SEGMENT READONLY** выравнивание тип разряд 'класс'

...

имя **ENDS**

Структура программы на ассемблере (Зубков С. В., Assembler для DOS, Windows, …, глава 3):

- Модули (файлы исходного кода)
- Сегменты (описание блоков памяти)
- Составляющие программного кода:
  - команды процессора
  - инструкции описания структуры, выделения памяти, макроопределения
- Формат строки программы:
  - метка команда/директива операнды ; комментарий

Директива SEGMENT

Каждая программа, написанная на любом языке программирования, состоит из одного или нескольких сегментов. Обычно область памяти, в которой находятся команды, называют сегментом кода, область памяти с данными - сегментом данных и область памяти, отведённую под стек, - сегментом стека.

Выравнивание:

- BYTE
- WORD
- DWORD
- PARA (по умолчанию)
- PAGE

Тип:

- PUBLIC - заставляет компоновщик соединить все сегменты с одинаковым именем. Новый объединенный сегмент будет целым и непрерывным. Все адреса (смещения) объектов, а это могут быть, в зависимости от

типа сегмента, команды или данные, будут вычисляться относительно начала этого нового сегмента;

- STACK - определение сегмента стека. Заставляет компоновщик соединить все одноименные сегменты и вычислять адреса в этих сегментах относительно регистра SS. Комбинированный тип STACK (стек) аналогичен комбинированному типу PUBLIC, за исключением того, что регистр SS является стандартным сегментным регистром для сегментов стека. Регистр SP устанавливается на конец объединенного сегмента стека. Если не указано ни одного сегмента стека, компоновщик выдаст предупреждение, что стековый сегмент не найден. Если сегмент стека создан, а комбинированный тип STACK не используется, программист должен явно загрузить в регистр SS адрес сегмента (подобно тому, как это делается для регистра DS);

- COMMON - располагает все сегменты с одним и тем же именем по одному адресу. Все сегменты с данным именем будут перекрываться и совместно использовать память. Размер полученного в результате сегмента будет равен размеру самого большого сегмента;

- AT - располагает сегмент по абсолютному адресу параграфа (параграф — объем памяти, кратный 16, поэтому последняя шестнадцатеричная цифра адреса параграфа равна 0). Абсолютный адрес параграфа задается выражением xxxx. Компоновщик располагает сегмент по заданному адресу памяти (это можно использовать, например, для доступа к видеопамяти или области ПЗУ), учитывая атрибут комбинирования. Физически это означает, что сегмент при загрузке в память будет расположен, начиная с этого абсолютного адреса параграфа, но для доступа к нему в соответствующий сегментный регистр должно быть загружено заданное в атрибуте значение. Все метки и адреса в определенном таким образом сегменте отсчитываются относительно заданного абсолютного адреса;

- PRIVATE (по умолчанию) - сегмент не будет объединяться с другими сегментами с тем же именем вне данного модуля.

Класс:

Это любая метка, взятая в одинарные кавычки. Сегменты одного класса расположатся в памяти друг за другом.

Директива ASSUME

ASSUME *регистр* : *имя сегмента*

- Не является командой
- Нужна для контроля компилятором правильности обращения к переменным

Модель памяти

.model *модель, язык, модификатор*

- TINY - один сегмент на всё
- SMALL - код в одном сегменте, данные и стек - в другом
- COMPACT - допустимо несколько сегментов данных
- MEDIUM - код в нескольких сегментах, данные - в одном
- LARGE, HUGE
- Язык - C, PASCAL, BASIC, SYSCALL, STDCALL. Для связывания с ЯВУ и вызова подпрограмм.
- Модификатор - NEARSTACK/FARSTACK
- Определение модели позволяет использовать сокращённые формы директив определения сегментов.

Конец программы и точка входа

...

END start

- start - имя метки, объявленной в сегменте кода и указывающее на команду, с которой начнётся исполнение программы.

- Если в программе несколько модулей, только один может содержать начальный адрес.

05. Прерывание 21h. Примеры ввода вывода.

Прерывание 21h

- Аналог системного вызова в современных ОС.
- Используется наподобие вызова подпрограммы.
- Номер функции передается через AH.

функция	назначение	вход	выход
2	Вывод символа в stdout	DL = ASCII-код символа	-
9	Вывод строки в stdout	DS:DX - адрес строки, заканчивающейся символом $	-
1	Считать символ из stdin с эхом	-	AL – ASCII-код символа
6	Считать символ без эха, без ожидания, без проверки на Ctrl+Break	DL=FF	AL – ASCII-код символа
7	Считать символ без эха, с ожиданием и без проверки на Ctrl+Break	-	AL – ASCII-код символа

8	Считать символ без эха	-	AL – ASCII-код символа
10 (0Ah)	Считать строку с stdin в буфер	DS:DX - адрес буфера	Введённая строка помещается в буфер
0Bh	Проверка состояния клавиатуры - AL=0, если клавиша не была нажата, и FF, если была		
OCh	Очистить буфер и считать символ	AL=01, 06, 07, 08, 0Ah	-

Еще одним важным случаем, когда нам требуется прерывание DOS - завершение программы. Чтобы ассемблер перестал читать подряд строки кода нам требуется положить в ah код DOS завершения программы (04Ch) и вызвать прерывание. Код завершения программы (ошибка или нет) кладется и берется из al.

```
; завершить программу с кодом 3
mov al, 03h
mov ah, 04Ch ; оно же 4Ch
int 21h
```

Примеры

```
; ввод символа (результат сохраняется в в al)
mov ah, 01h
int 21h

----------

; вывод символа на экран (содержащегося в dl)
```

```
mov dl, 'x'
mov ah, 02h
int 21h
```

06. Стек. Назначение, примеры использования.

Стек. Назначение, примеры использования.

Стек работает по правилу LIFO / FILO (последним пришёл, последним вышел)

Сегмент стека — область памяти программы, используемая её подпрограммами, а также (вынужденно) обработчиками прерываний.

Используется для временного хранения переменных, передачи параметров для подпрограм, адрес возврата при вызове процедур и прерываний.

Регистр SP — указывает на вершину стека

В x86 стек "растёт вниз", в сторону уменьшения адресов (от максимально возможно адреса). При запуске программы SP указывает на конец сегмента.

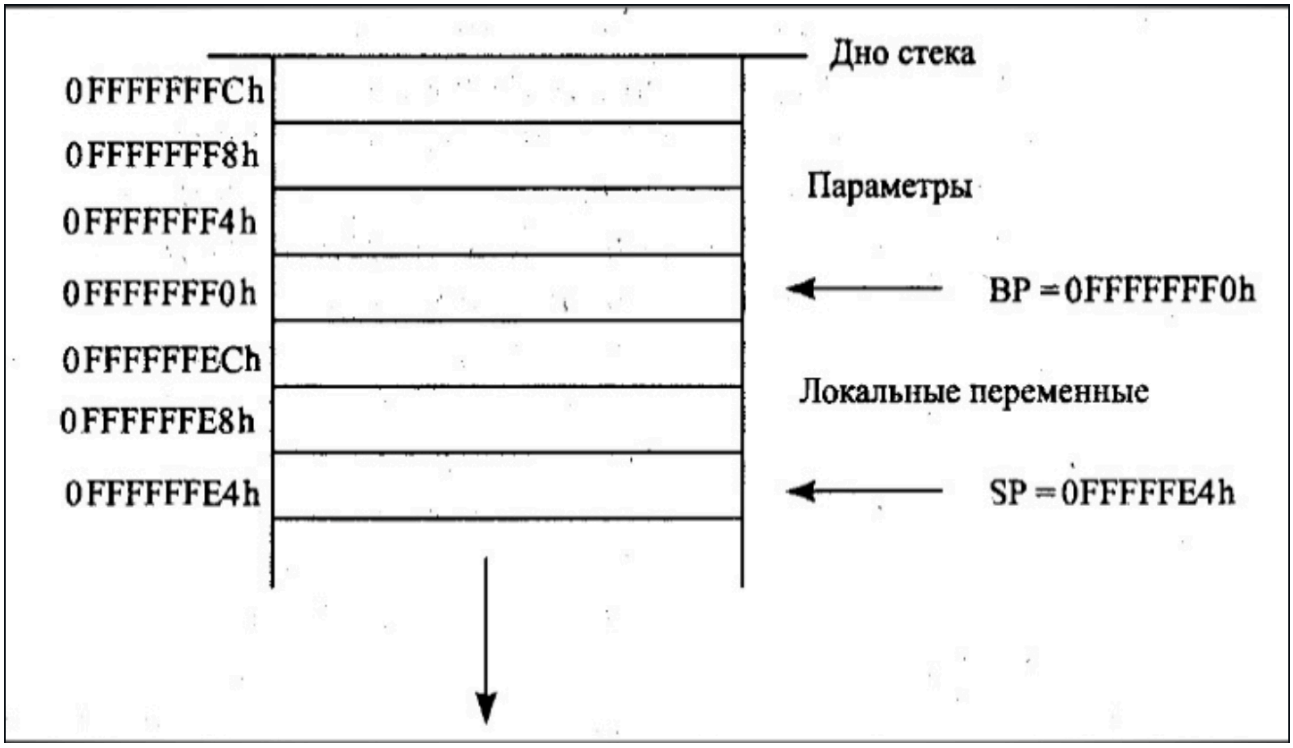
BP (Base Pointer)

Используется в подпрограмме для сохранения "начального" значения SP.

Так же, используется для адресации параметров и локальных переменых.

При вызове подпрограммы параметры кладут на стек, а в BP кладут текущее значение SP. Если программа использует стек для хранения локальных переменных, SP изменится и таким

образом можно будет считывать переменные напрямую из стека (их смещения запишутся как BP + номер параметра)



Команды работы со стеком

PUSH <источник> — поместить данные в стек. Уменьшает SP на размер источника и записывает значение по адресу SS:SP.

POP <приемник> — считать данные из стека. Считывает значение с адреса SS:SP и увеличивает SP.

PUSHA — поместить в стек регистры AX, CX, DX, BX, SP, BP, SI, DI. (регистры общего назначения + SP + BP)

POPA — загрузить регистры из стека (SP игнорируется)

CALL и RET

CALL <операнд> — передает управление на адрес <операнд>

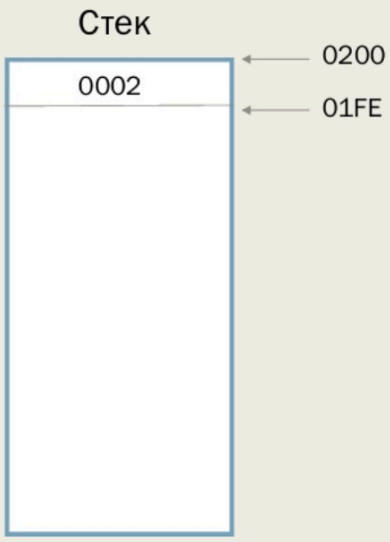
Сохраняет адрес следующей команды в стеке (уменьшает SP и записывает по его адресу IP либо CS:IP, в зависимости от размера аргумента)

RET <число> — загружает из стека адерс возврата, увеличивая SP.

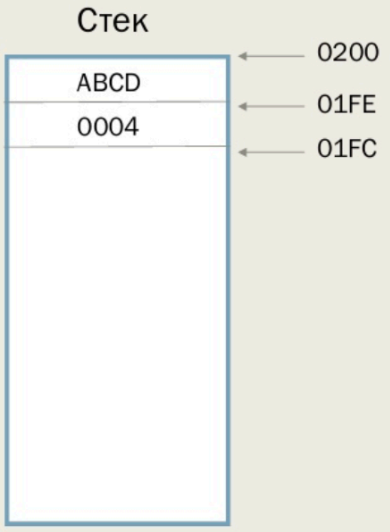
Если указать операнд, то можно очистить стек для очистки стека от параметров (<число> будет прибавлено к SP)

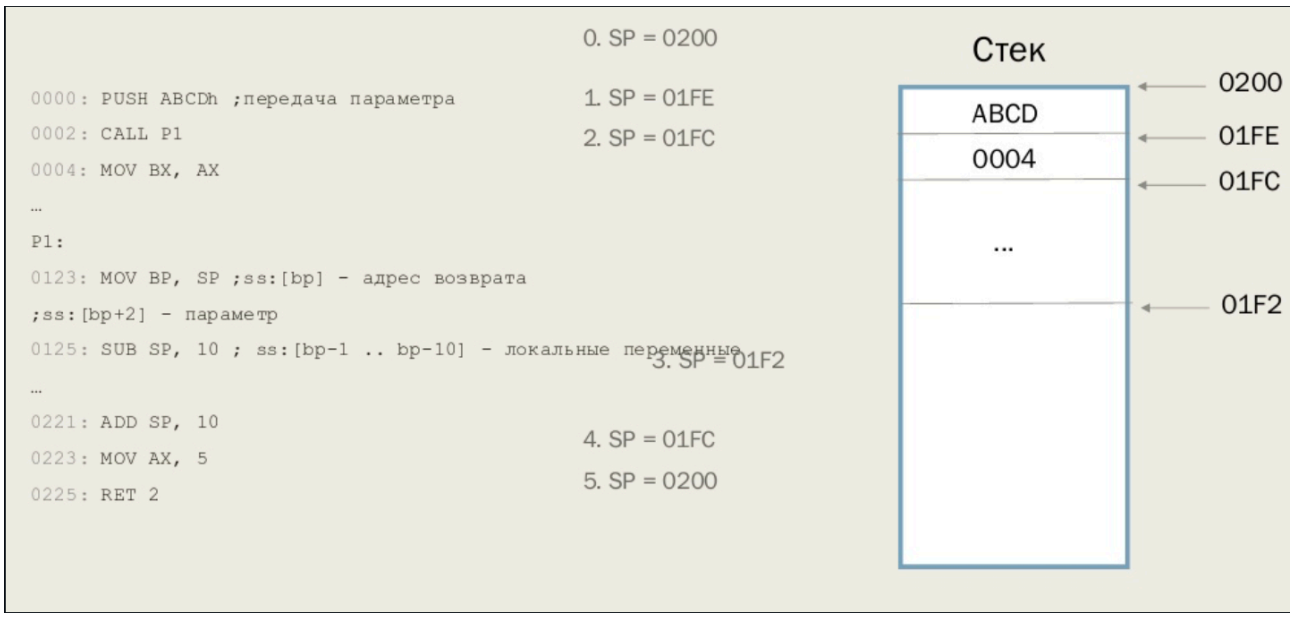
Примеры использования

```
                    0. SP = 0200
0000: CALL P1       1. SP = 01FE
0002: MOV BX, AX
…
P1:
0123: MOV AX, 5
0125: RET           2. SP = 0200
```




```
                                        0. SP = 0200
0000: PUSH ABCDh ;передача параметра    1. SP = 01FE
0002: CALL P1                           2. SP = 01FC
0004: POP DX                            4. SP = 0200
0006: MOV BX, AX
…
P1:
0123: MOV BP, SP ;ss:[bp] - адрес возврата
;ss:[bp+2] - параметр
...
0223: MOV AX, 5
0225: RET                               3. SP = 01FE
```





07. Регистр флагов.

Регистр флагов

Флаги **выставляются при операциях**, но не обязательно все сразу. Например INC и DEC не затрагивают флаг CF, в отличии от ADD и SUB.

Также есть команды рассчитанные на флаги, например CMP, которая выставляет флаги такие, как если бы произошло вычитание аргументов.

Как мы помним, регистры у нас размером в 16 бит.

Вот за что отвечает каждый бит в регистре FLAGS:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
CF	-	PF	-	AF	-	ZF	SF	TF	IF	DF	OF	IOPL	IOPL	NT	-

▪ CF (carry flag) - флаг переноса - устанавливается в 1, если результат предыдущей операции не уместился в приемник и произошел перенос или если требуется заем при вычитании. Иначе 0.

▪ PF (parity flag) - флаг чётности - устанавливается в 1, если младший байт результата предыдущей операции содержит четное количество единиц.

▪ AF (auxiliary carry flag) - вспомогательный флаг переноса - устанавливается в 1, если в результате предыдущей операции произошел перенос из 3 в 4 или заем из 4 в 3 биты.

▪ ZF (zero flag) - флаг нуля - устанавливается в 1, если если результат предыдущей команды равен 0.

▪ SF (sign flag) - флаг знака - всегда равен старшему биту результата.

▪ TF (trap flag) - флаг трассировки - предусмотрен для работы отладчиков в пошаговом режиме. Если поставить в 1, после каждой команды будет происходить передача управления отладчику.

▪ IF (interrupt enable flag) - флаг разрешения прерываний - если 0 процессор перестает обрабатывать прерывания от внешних устройств.

▪ DF (direction flag) - флаг направления - контролирует поведение команд обработки строк. Если 0, строки обрабатываются слева направо, если 1 справа налево.

▪ OF (overflowflag) - флаг переполнения - устанавливается в 1, если результат предыдущей операции над числами со знаком выходит за допустимые для них пределы.

▪ IOPL (I/O privilege level) - уровень приоритета ввода-вывода - а это на 286, на не нужно пока.

▪ NT (nested task) - флаг вложенности задач - а это на 286, на не нужно пока.

08. Команды условной и безусловной передачи управления.

Условный переход - переход, происходящий при выполнении какого-то условия.

Безусловный переход - переход, не зависящий от чего-либо (совершаемый в любом случае).

Виды безусловных переходов

JMP - оператор безусловного перехода.

Вид перехода	Дистанция перехода
short (короткий)	-128..+127 байт
near (ближний)	в том же сегменте (без изменения CS)
far (дальний)	в другой сегмент (со сменой CS)

Команды, выставляющие флаги и использующиеся при переходах к передаче управления

CMP <приемник>, <источник>

Источник - число, регистр или переменная.
Приемник - регистр или переменная; не может быть переменной одновременно с источником.

Вычитает источник из приёмника, результат никуда не сохраняется, выставляются флаги **CF, PF, AF, ZF, SF, OF**.

TEST <приемник>, <источник>

Аналог **AND**, но результат не сохраняется. Выставляются флаги **SF, ZF, PF**.

09. Организация многомодульных программ.

Организация многомодульных программ

Как и на других языках программирования, программа на ассемблере может состоять из нескольких файлов - модулей. При компиляции (трансляции) каждый модуль превращается в объектный файл, далее при компоновке объектные файлы соединяются в единый исполняемый модуль.

Модули обычно состоят из описания сегментов будущей программы с помощью директивы SEGMENT.

Пример:

```
имя SEGMENT [READONLY] выравнивание тип разряд 'класс'
...
имя ENDS
```

Параметры:

• Выравнивание - расположение начала сегмента с адреса, кратного какому-либо значению. Варианты:

BYTE;

WORD (2 байта);

DWORD (4 байта);

PARA (16 байт, по умолчанию);

PAGE (256 байт).

• Тип:

PUBLIC (сегменты с одним именем объединятся в один);

STACK (для стека); COMMON (сегменты будут “наложены” друг на друга по одним и тем же адресам памяти);

AT <начало> - расположение по фиксированному физическому адресу, параметр - сегментная часть этого адреса;

PRIVATE - вариант по умолчанию.

• Класс - метка, позволяющая объединить сегменты (расположить в памяти друг за другом).

10. Подпрограммы. Объявление, вызов.

Описание подпрограммы

```
имя_подпрограммы PROC [NEAR | FAR] ; по умолчанию NEAR, если не указать
    ;тело подпрограммы;
```

```
    ret [кол-во используемых локальный переменных] ; ничего не
указывается, если не использовались локальные переменные на стеке
имя_подпрограммы ENDP
```

Вызов подпрограммы

```
; вызов любой (в плане расстояния) подпрограммы
call имя_подпрограммы
```

CALL - вызов процедуры, RET - возврат из процедуры

CALL <операнд>

- Сохраняет адрес следующей команды в стеке (уменьшает SP и записывает по его адресу IP либо CS:IP, в зависимости от размера аргумента)
- Передаёт управление на значение аргумента.

RET/RETN/RETF <число>

- Загружает из стека адрес возврата, увеличивает SP
- Если указан операнд, его значение будет дополнительно прибавлено к SP для очистки стека от параметров

Отличие RETN и RETF в том, что 1ая команда делает возврат при ближнем переходе, 2ая - при дальнем (различие в кол-ве байт, считываемых из стека при возврате). Если используется RET, то ассемблер сам выберет между RETN и RETF в зависимости от описание подпрограммы (процедуры).

BP – base pointer

- Используется в подпрограмме для сохранения "начального" значения SP
- Адресация параметров
- Адресация локальных переменных (подробнее см. стек)

11. Арифметические команды.

Арифметические команды.

ADD и ADC

ADD <приемник>, <источник> — сложение. Не делает различий между знаковыми и беззнаковыми числами.

ADC <приемник>, <источник> — сложение с переносом. Складывает приёмник, источник и флаг CF.

SUB и SBB

SUB <приемник>, <источник> — вычитание. Не делает различий между знаковыми и беззнаковыми числами.

SBB <приемник>, <источник> — вычитание с займом. Вычитает из приёмника источник и дополнительно - флаг CF.

Флаг **CF** можно рассматривать как дополнительный бит у результата.

$11111111_2 + 00000001_2 = (1)00000000_2$ (флаг установлен)

Можно использовать ADC и SBB для сложения вычитания и больших чисел, которые по частям храним в двух регистрах.

Пример: Сложим два 32-битных числа. Пусть одно из них хранится в паре регистров DX:AX (младшее двойное слово - DX, старшее AX). Другое в паре BX:CX

```
add ax, cx
adc dx, bx
```

Если при сложении двойных слов произошел перенос из старшего разряда, то это будет учтено командой adc.

Эти 4 команды (ADD, ADC, SUB, SBB) меняют флаги: CF, OF, SF, ZF, AF, PF

MUL и IMUL

MUL <источник> — выполняет умножение чисел без знака. <источник> не может быть число (нельзя: MUL 228). Умножает

регистр AX (AL), на <источник>. Результат остается в AX, либо DX:AX, если не помещается в AX.

IMUL — умножение чисел со знаком.

1. **IMUL** <источник>. Работает так же, как и MUL
2. **IMUL** <приёмник>, <источник>. Умножает источник на приемник, результат в приемник.
3. **IMUL** <приёмник>, <источник1>, <источник2>. Умножает источник1 на источник2, результат в приёмник. **Флаги**: OF, CF

DIV и IDIV

DIV <источник> — выполняет деление чисел без знака. <источник> не может быть число (нельзя: DIV 228). Делимое должно быть помещено в AX (или DX:AX, если делитель больше байта). В первом случае частное в AL, остаток в AH, во втором случае частное в AX, остаток в DX.

IDIV <источник> — деление чисел со знаком. Работает так же как и DIV. Округление в сторону нуля, знак остатка совпадает со знаком делимого.

INC, DEC, NOT

INC <приемник> — увеличивает примник на 1.

DEC <приемник> — уменьшает примник на 1.

Меняют флаги: OF, SF, ZF, AF, PF

NEG <примeник> — меняет знак приемника.

12. Команды побитовых операций.

Операции над битами и байтами

BT <база>, <смещение>

Считывает в CF значение бита из битовой строчки.

BTS <база>, <смещение>

Устанавливает бит в 1.

BTR <база>, <смещение>

Сбрасывает бит в 0.

BTC <база>, <смещение>

Инвертирует бит.

BSF <приемник>, <смещение>

Прямой поиск бита (от младшего разряда).

BSR <приемник>, <смещение>

Обратный поиск бита (от старшего разряда).

SETcc <приемник>

Выставляет приемник (1 байт) в 1 или 0 в зависимости от условия, аналогично Jcc.

Логический, арифметический, циклический сдвиг

SAL (SHL)

Арифметический сдвиг налево.

SHR

Логический сдвиг направо, зануляет старший бит.

SAR

Арифметический сдвиг направо, сохраняет знак.

ROR (ROL)

Циклический сдивг вправо (влево).

RCR (RCL)

Циклический сдвиг вправо (влево) через CF.

13. Команды работы со строками.

Строковые операции: копирование, сравнение, сканирование, чтение, запись

Строка-источник - `DS:SI`, строка-приёмник - `ES:DI`.

За один раз обрабатывается один байт (слово).

- `MOVS` / `MOVSB` / `MOVSW` `<приёмник>, <источник>` - копирование
- `CMPS` / `CMPSB` / `CMPSW` `<приёмник>, <источник>` - сравнение
- `SCAS` / `SCASB` / `SCASW` `<приёмник>` - сканирование (сравнение с AL/AX (в зависимости от размеров приемника)
- `LODS` / `LODSB` / `LODSW` `<источник>` - чтение (в `AL/AX`)
- `STOS` / `STOSB` / `STOSW` `<приёмник>` - запись (из `AL/AX`)

- Префиксы: REP / REPE / REPZ / REPNE / REPNZ
- REP - повторить следующую строковую операцию
- REPE - повторить следующую строковую операцию, если равно
- REPZ - Повторить следующую строковую операцию, если нуль
- REPNE - повторить следующую строковую операцию, если не равно
- REPNZ - повторить следующую строковую операцию, если не нуль

Префиксы REP (F3h), REPE (F3h) и REPNE (F2h) применяются со строковыми операциями. Каждый префикс заставляет строковую команду, которая следует за ним, повторяться указанное в регистре счетчике (E)CX (в случае нашей модели процессора 8086 - CX) количество раз или, кроме этого, (для префиксов REPE и REPNE) пока не встретится указанное условие во флаге ZF.

Пример использования: REP LODS AX

Мнемоники REPZ и REPNZ являются синонимами префиксов REPE и REPNE соответственно и имеют одинаковые с ними коды. Префиксы REP и REPE / REPZ также имеют одинаковый код F3h, конкретный тип префикса задается неявно той командой, перед которой он применен.

Все описываемые префиксы могут применяются только к одной строковой команде за один раз. Чтобы повторить блок команд, используется команда LOOP или другие циклические конструкции.

Затрагиваемые флаги: OF, DF, IF, TF, SF, ZF, AF, PF, CF