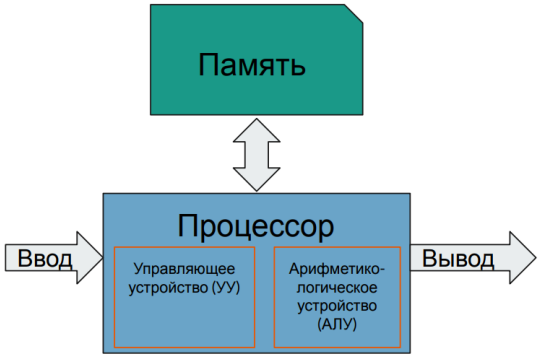
1. Архитектура фон Неймана, принципы фон Неймана.

## Архитектура фон Неймана

От решения частных вычислительных задач - к универсальным системам (теперь программа уже не была постоянной частью машины (как например, у калькулятора))




- Абстрактный Процессор состоит из блоков УУ и АЛУ
    - УУ - дискретный конечный автомат. – интерпретирует, обрабатывает и выполняет команды.
      *Структурно состоит из:*
      *дешифратора команд (операций),*
      *регистра команд,*
      *узла вычислений текущего исполнительного адреса,*
      *счётчика команд (регистр IP).*
    - АЛУ - под управлением УУ производит преобразование над данными (операндами) – выполняет арифметические и логические операции.
      *Разрядность операнда - длина машинного слова. (Машинное слово - машинно-зависимая величина, измеряемая в битах, равная разрядности регистров/шины данных)*
- Кроме того, процессор взаимодействует с памятью из которой он берет значения и программный код и пишет изменяемые значения.

## Принципы фон Неймана

1. Использование двоичной с/с в вычислительных машинах.
2. Программное управление ЭВМ.
3. Память используется не только для хранения данных, но и для программ. *(Принцип однородности памяти)*
4. Ячейки памяти ЭВМ имеют адреса, которые последовательно пронумерованы.
5. Возможность условного перехода в процессе выполнения программы.

2. Машинные команды, машинный код. Понятие языка ассемблера.

- Машинная команда - инструкция (в двоичном коде) из аппаратно определённого набора, которую способен выполнять процессор. *(от 1 байта)*
  *Можно разделить на*
    - *Команды пересылки данных*

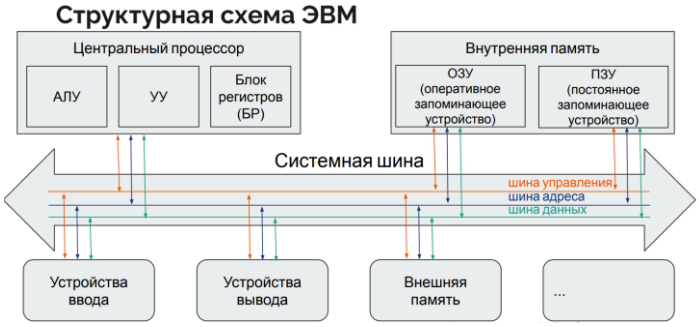
- *Арифметические и логические команды*
- *Команды переходов (условных и безусловных)*
- *Команды работы с подпрограммами*
- *Команды управления процессором*

*Любая команда ЭВМ состоит из двух частей. Операционная сообщает, какое действие необходимо выполнить с информацией. Адресная часть описывает, откуда взять данные, и куда положить результат.*

- Машинный код - система команд конкретной вычислительной машины, которая интерпретируется непосредственно процессором.
- Язык ассемблера - машинно-зависимый язык программирования низкого уровня, команды которого прямо соответствуют машинным командам.

## 3. Виды памяти ЭВМ. Запуск и исполнение программы.

### *Структурная схема ЭВМ*




- *отражает схему современного компьютера.*
- *Центральный элемент – системная шина – некоторая абстракция, которая осуществляет взаимодействие между собой всех устройств. Делится на: шина управления, шина адреса и шина данных. К ней все подключаются – центральной процессор, внутренняя память (оперативная в первую очередь).*
- *Если процессору необходимо считать какие-то данные, то он выставляет на шину адреса номер ячейки памяти (адрес) и по шине управления отправляет сигнал в память – считай значение. Программа останавливает свою работу и ждет ответа. Память находит ячейку памяти по адресу, считывает значение и устанавливает на шину данных и отправляет по шине управления сигнал – готово, значение установлено. ЦП получает данные и программа продолжается. Если же нужно записать, то по аналогии. Адрес, данные, сигнал. Приостановление. Выполнение, сигнал.*
- *В ЦП добавляется Блок регистров – внутренние ячейки памяти процессор, их всего несколько десятков.*
- *Также процессор взаимодействует **с оперативкой** – единственный участок памяти, к которым процессор имеет доступ, туда загружается код программы. Доступ к дисками, флешкам – это как внешняя память.*
- *На практике системная шина – материнская плата. + обычно несколько шин – для оперативки, для другой памяти.*
- *Процессор в соответствии с архитектурой при подаче питания начинает работать с некого фиксированного адреса. Материнские платы настраивают так, чтобы при включении данные с ПЗУ копировались в оперативную память. То*

*есть, чтобы в ОЗУ по фиксированному адресу находилась стартовая программа (BIOS). Она определяет первичную диагностику устройств, определяет, готов ли компьютер функционировать, находит загрузочный диск и загружает операционную систему.*

## Виды памяти ЭВМ

Зачем так разделять – чем более быстрая, тем меньше объем. Регистр – кэш память – оперативная – постоянная.

1. Блок Регистров - внутренние ячейки памяти процессора. Самая быстрая память.
2. Кэш-память процессора - на схеме её нет, но она присутствует во всех современных процессорах. Чуть медленнее регистров, используется для ускорения работы с оперативной памятью.
3. Внутренняя память делится на:
    a. ОЗУ (оперативное запоминающее устройство) – энергозависимая, быстрая, небольшая по объему память для чтения и записи информации. Процессор имеет к нему прямой доступ. В неё загружают данные и программы, с её помощью работает компьютер в целом. Очищается при отключении питания.
    (Random Access Memory, RAM — память с произвольным доступом) — *энергозависимая часть системы компьютерной памяти, в которой во время работы компьютера хранится выполняемый машинный код (программы), а также входные, выходные и промежуточные данные, обрабатываемые процессором. ОЗУ может изготавливаться как отдельный внешний модуль или располагаться на одном кристалле с процессором*

    b. ПЗУ (постоянное запоминающее устройство) – энергонезависимая, только для чтения. В ПЗУ хранится информация, которая записывается туда при изготовлении ЭВМ. Важнейшая микросхема ПЗУ - BIOS.
    ROM (read-only memory, постоянное запоминающее устройство)
    Нужна для запуска компьютера, так как оперативная память очищается. В ПЗУ хранится стартовая программа загрузка компьютера.

4. Внешняя память – различные магнитные и оптические накопители. Используются для долгосрочного хранения данных.

## Запуск и исполнение программ

- **Исполняемый файл** - файл, содержащий программу в виде, в котором она может быть исполнена компьютером (то есть в машинном коде).
- **Получение исполняемых файлов** обычно включает в себя 2 шага: компиляцию и линковку.
    - Компилятор - программа для преобразования исходного текста другой программы на определённом языке в объектный модуль.
    - Компоновщик (линковщик, линкер) - программа для связывания нескольких объектных файлов в исполняемый.
- В DOS и Windows - расширения .EXE и .COM
- **Последовательность запуска** программы операционной системой:

1. Определение формата файла.
2. Чтение и разбор заголовка.
3. Считывание разделов исполняемого модуля (файла) в ОЗУ по необходимым адресам.
4. Подготовка к запуску, если требуется (загрузка библиотек).
5. Передача управления на точку входа.

- ***Отладчик*** - *программа для автоматизации процесса отладки. Может выполнять трассировку, отслеживать, устанавливать или изменять значения переменных в процессе выполнения кода, устанавливать или удалять контрольные точки или условия остановки.*

**.COM (command)**
- простейший формат исполняемых файлов DOS и ранних версий Windows:

- не имеет заголовка;
- состоит из одной секции, не превышающей 64 Кб;
- загружается в ОЗУ без изменений;
- начинает выполняться с 1-го байта (точка входа всегда в начале).
- Последовательность запуска COM-программы:
    1. Система выделяет свободный сегмент памяти нужного размера и заносит его адрес во все сегментные регистры (CS, DS, ES, FS, GS, SS).
    2. В первые 256 (100h) байт этого сегмента записывается служебная структура DOS, описывающая программу - PSP.
    3. Непосредственно за ним загружается содержимое COM-файла без изменений.
    4. Указатель стека (регистр SP) устанавливается на конец сегмента.
    5. В стек записывается 0000h (начало PSP - адрес возврата для возможности завершения командой ret).
    6. Управление передаётся по адресу CS:0100h.

*Программа будет последовательно исполняться до тех пор, пока не будет вызвана инструкция того, что исполнение закончено, иначе программа не закончит работу*

4. Сегментная модель памяти в архитектуре 8086.
5. Процессор 8086. Сегментные регистры. Адресация в реальном режиме. Понятие сегментной части адреса и смещения.

**Реальный режим работы** - режим совместимости современных процессоров с 8086.

- обращение к оперативной памяти происходит по реальным (действительным) адресам, трансляция адресов не используется;
- набор доступных операций не ограничен;
- защита памяти не используется.

*В этой архитектуре 20-разрядная адресация (разрядность шины адреса) (то есть доступно 2^20байт = 1Мб памяти, а не 2^16. Шина данных – 16 разрядная)*

В реальном режиме работы всё адресное пространство делится на сегменты. Под сегментом понимается блок смежных ячеек с максимальным размером 64 Кбайт (2^16 байт) и начальным или базовым адресом, находящимся на 16-байтной границе (такая

граница называется параграфом). *Таким образом, сегменты могут частично перекрывать друг друга.*

Логически память имеет *3 вида* сегментов и соответствующих сегментных регистров:

- Сегмент кода - регистр CS. Командой MOV изменить невозможно, меняется автоматически по мере выполнения команд.
- Сегмент данных. Основной регистр - DS, при необходимости дополнительных сегментов данных задействуются ES, FS, GS.
- Сегмент стека - регистр SS

(*В 8086 регистры 16 битные .Регистры — специальные ячейки памяти, находящиеся физически внутри процессора, доступ к которым осуществляется не по адресам, а по именам. Поэтому, работают очень быстро.*)

**Сегментный регистр** хранит в себе старшие 16 разрядов (из 20) адреса начала сегмента. 4 младших разряда в адресе начала сегмента всегда нулевые. Говорят, что сегментный регистр содержит в себе **номер параграфа (=16 байт) начала сегмента**.

Физический адрес получается сложением адреса начала сегмента (на основе сегментного регистра) и смещения.В реальном режиме для вычисления физического адреса, адрес из сегмента сдвигают влево на 4 разряда (можно сказать, что просто приписывают 0 в конце или умножают на 16) и добавляют смещение. Это делается процессором аппаратно, без участия программиста. Например, логический адрес 7522:F139 дает физический адрес 84359.

Распространённые пары регистров: CS:IP, DS:BX, SS:SP

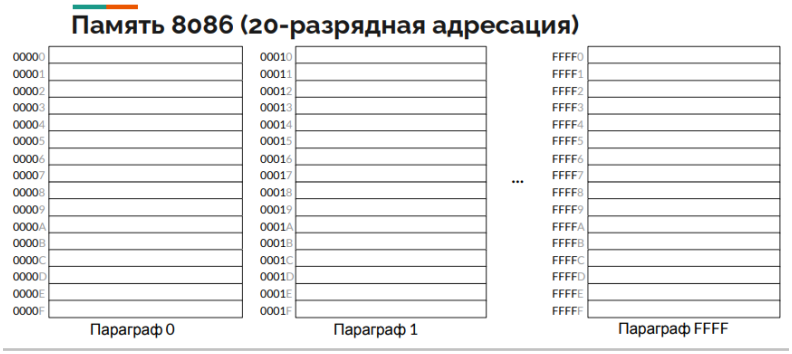
*На шину передается именно физический адрес. Если результат больше, чем 2^20 - 1, то 21 бит отбрасывают.*

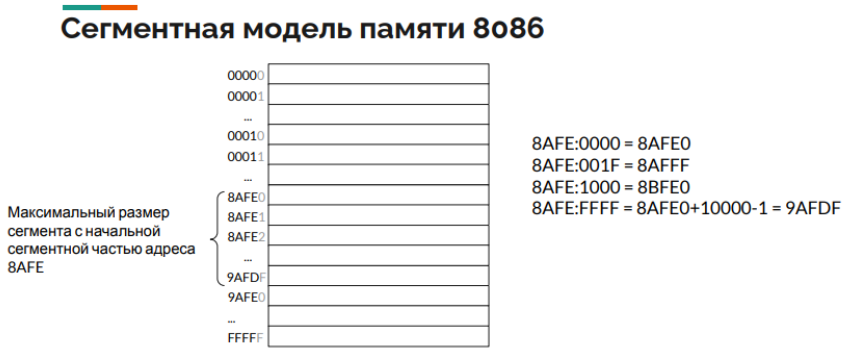
*При такой адресации адреса 0400h:0001h и 0000h:4001h будут ссылаться на одну и ту же ячейку памяти, так как 400h × 16 + 1 = 0 × 16 + 4001h*

*Отметим, что сегментные регистры содержат физические адреса памяти, т.е. значение в каждом сегментном регистре прямо указывает на границу параграфа в адресном пространстве 1 Мбайт.*

*Offset возвращает значение метки в памяти.*

*LEA <приемник>, <источник>*




## *Способы адресации*

- *Регистровая адресация (mov ax, bx) - Операнды могут располагаться в любых регистрах общего назначения и сегментных регистрах.*
- *Непосредственная адресация (mov ax, 2) - Некоторые команды (все арифметические, кроме деления) позволяют указывать один из операндов непосредственно в тексте программы.*
- *Прямая адресация (адресация по смещению) (mov ax, es:0001; mov ax,es:word_var) - Если у операнда, располагающегося в памяти, известен адрес, то его можно использовать. Если селектор сегмента данных находится в DS, то имя сегментного регистра при прямой адресации можно не указывать, DS используется по умолчанию.*
- *Косвенная адресация (mov ax, [bx]). адрес операнда в памяти можно хранить в любом регистре.*
  *в регистр AX слово из ячейки памяти, селектор сегмента которой находится в DS, а смещение - в BX*
  *Как и в случае с прямой адресацией, DS используется по умолчанию, но не всегда: если смещение берут из регистров ESP, EBP или BP, то в качестве сегментного регистра применяется SS.*
  *В 8086 допустимы BX, BP, SI, DI*
- *Адресация по базе со сдвигом (mov ax, [bx]+2; mov ax, 2[bx]). - скомбинируем два предыдущих метода адресации.*
- *Адресация по базе с индексированием (допустимы BX+SI, BX+DI, BP+SI, BP+DI): сумма*

- mov ax, [bx+si+2]
- mov ax, [bx+2][si]
- mov ax, 2[bx][si]
- mov ax, [bx][si]+2
- mov ax, [bx][si+2]

## 6. Процессор 8086. Регистры общего назначения.

В 8086 регистры 16 битные.

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

**Регистры общего назначения. AX, BX, CX, DX**
регистры данных, каждый из которых помимо хранения операндов и результатов операций имеет свое назначение. Большую часть времени используются произвольно, могут использоваться без ограничений, для любых целей

При использовании регистров общего назначения, можно обратится к каждым 8 битам (байту) по-отдельности, используя вместо *X - *H или *L

Верхние 8 бит (1 байт) — AH, BH, CH, DH

Нижние 8 бит (1 байт) — AL, BL, CL, DL

**AX(accumulator)** - используется в качестве операнда для операций *, / и обмена с внешними устройствами (ввод / вывод).

Например, используется при MUL и DIV (умножение и деление)

MUL:

- AX = AL * ЧИСЛО - при mul если число == байт;
- DX:AX = AX * число, если число == слово;

DIV

- AL = AX / ЧИСЛО, остаток в AH при div если число == байт;
- AX = DX:AX / ЧИСЛО, остаток в DX, если число == слово;

часто используется для хранения результата действий, выполняемых над двумя операндами.

**BX(base)** - базовый регистр в вычислениях адреса, часто указывает на начальный адрес (называемый базой) структуры в памяти; DS:BX

**CX(counter)** - счетчик циклов, определяет количество повторов некоторой операции;

**DX(data)** - регистр данных - определение адреса ввода/вывода, также может содержать данные, передаваемые для обработки в подпрограммы.

## 7. Процессор 8086. Регистр флагов.

Регистр флагов – специальный регистр, к которому напрямую обратиться нельзя, и используется он не как число 16 разрядное, а как отдельные биты независимо друг от друга, имеет собственное значение

*Флаги выставляются при выполнение операций, в основном арифметических. С помощью этих флагов можно определить что-нибудь определить, например, было ли переполнение при последней выполненной операции. 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.*

*Хотя разрядность регистра FLAGS 16 бит, реально используют не все 16 (11). Остальные были зарезервированы при разработке процессора, но так и не были использованы.*


Флаги делятся на группы:

1. Флаги состояний (CF, AF, PF, ZF, SF, OF)
2. Системные флаги (TF, IF, IOPL, NT)
3. Флаг направления (DF).

CF, TF, IF, DF - можно изменять командами. Остальные флаги напрямую нельзя изменить

*Управление флагами*

- *STC/CLC/CMC - установить/сбросить/инвертировать CF*
- *STD/CLD - установить/сбросить DF*
- *AHF - загрузка флагов состояния в AH*
- *SAHF - установка флагов состояния из AH*
- *CLI/STI - запрет/разрешение прерываний (сброс/установка IF)*

1. зеленый DF (direction flag) - флаг направления
   Он контролирует поведение команд обработки строк: когда он установлен в 1, строки обрабатываются в сторону уменьшения адресов (справа-налево), когда DF = 0 – наоборот (слева-направо).

черные - флаги состояния 6 - CF, PF, AF, ZF, SF, OF

2. CF (carry flag) - флаг переноса
   устанавливается в 1, если результат предыдущей операции не уместился в приемник и произошел перенос из старшего бита или если требуется заем при вычитании. Иначе 0.
   (Например, после сложения слова OFFFFh и 1, если регистр, в который надо поместить результат, - слово, в него будет записано OOOOh и флаг CF = 1.)
3. PF (parity flag) - флаг четности

Устанавливается в 1, если младший байт результата предыдущей команды содержит четное число битов, равных 1. Это не то же самое, что делимость на два. Число делится на два без остатка, если его самый младший бит равен нулю, и не делится, когда он равен 1. (в логических операциях)

4. 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. Этот флаг используется автоматически командами двоично-десятичной коррекции.
5. ZF (zero flag) - флаг нуля
   устанавливается в 1, если результат предыдущей команды равен 0.
6. SF (sign flag) - флаг знака
   всегда равен старшему биту результата.
7. OF (overflowflag) - флаг переполнения
   Для работы со знаковыми. устанавливается в 1, если результат предыдущей операции над числами со знаком выходит за допустимые для них пределы. Например, если при сложении двух положительных чисел получается число со старшим битом, равным единице, то есть отрицательное, и наоборот.
   100+100=200>127

синие - системные флаги 4 - TF, IF, IOPL, NT

8. TF (trap flag) - флаг трассировки (ловушка)
   предусмотрен для работы отладчиков в пошаговом режиме. Он был предусмотрен для работы отладчиков, не использующих защищенный режим. Установка его в 1 приводит к тому, что после выполнения каждой программной команды управление временно передается отладчику (вызывается прерывание 1 - см. описание команды INT).
9. IF (interrupt enable flag) - флаг разрешения прерываний
   если 0 процессор перестает обрабатывать прерывания от внешних устройств. Когда выполняются критические блоки кода, но надолго нельзя снимать
10. (286)IOPL (I/O privilege level) - уровень приоритета ввода-вывода
11. (286)NT (nested task) - флаг вложенности задач
    Последние 2 применяются в защищенном режиме

## 8. Команды пересылки данных.

*(Краткий список по Зубкову: mov, cmov, xchg,(команды работы со стеком, но тоже относятся к командам пересылки данных) push, pop, pusha, popa, (взаимодействие с портами) in, out, (отдельные) lea, xlat)*

1. MOV <приемник>, <источник> - базовая команда пересылки данных

Копирует содержимое источника в приемник, источник не изменяется

Источник: непосредственный операнд (число - константа, включённая в машинный код), РОН, сегментный регистр, переменная (ячейка памяти).

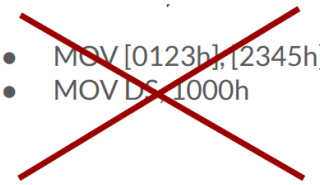
Приёмник: РОН, сегментный регистр (кроме CS), переменная (ячейка памяти).

При работе с памятью самый младший байт сохраняется в наиболее значимом байте: порядок байтов задом на перед.

- MOV AX, 5
- MOV BX, DX
- MOV [1234h], CH
- MOV DS, AX

Ограничения:

- Оба операнда одного размера
- Нельзя переменная-переменная, сегментный регистр-сегментный регистр
- Нельзя в сегментный регистр записать константу



- MOV [0123h], [2345h]
- MOV DS, 1000h

2. ТОЛЬКО С P6!!!CMOVcc <приемник>, <источник> - Условная пересылка данных. Копируют содержимое источника в приемник, если удовлетворяется условие

Источник: РОН, сегментный регистра, переменная (ячейка памяти).

Приёмник: только регистр

Обычно используется после cmp

```
cmp     ax, bx          ; Сравнить ax и bx.
cmovl   ax, bx          ; Если ax < bx, скопировать bx в ax.
```

Условия аналогичны Jcc

- Термины “выше” и “ниже” - при сравнении беззнаковых чисел
- Термины “больше” и “меньше” - при сравнении чисел со знаком

| Команда | Описание | Состояние флагов для выполнения перехода |
|---|---|---|
| JO | Есть переполнение | OF = 1 |
| JNO | Нет переполнение | OF = 0 |
| JS | Есть знак | SF = 1 |
| JNS | Нет знака | SF = 0 |
| JE, JZ | Если равно/если ноль | ZF = 1 |
| JNE, JNZ | Не равно/не ноль | ZF = 0 |
| JP/JPE | Есть чётность / чётное | PF = 1 |
| JNP/JPO | Нет чётности / нечётное | PF = 0 |
| JCXZ | CX = 0 | - |

| Команда | Описание | Состояние флагов для выполнения перехода | Знаковый |
|---|---|---|---|
| JB<br>JNAE<br>JC | Если ниже<br>Если не выше и не равно<br>Если перенос | CF = 1 | нет |
| JNB<br>JAE<br>JNC | Если не ниже<br>Если выше или равно<br>Если нет переноса | CF = 0 | нет |
| JBE<br>JNA | Если ниже или равно<br>Если не выше | CF = 1 или ZF = 1 | нет |
| JA<br>JNBE | Если выше<br>Если не ниже и не равно | CF = 0 и ZF = 0 | нет |

| Команда | Описание | Состояние флагов для выполнения перехода | Знаковый |
|---|---|---|---|
| JL<br>JNGE | Если меньше<br>Если не больше и не равно | SF <> OF | да |
| JGE<br>JNL | Если больше или равно<br>Если не меньше | SF = OF | да |
| JLE<br>JNG | Если меньше или равно<br>Если не больше | ZF = 1 или SF <> OF | да |
| JG<br>JNLE | Если больше<br>Если не меньше и не равно | ZF = 0 и SF = OF | да |

3. XCHG <оп1>, <оп2>

Обмен операндов между собой. Выполняется над двумя регистрами либо регистром и переменной

---

Команда **push** помещает значение источника в стек. Источником может быть регистр, сегментный регистр, непосредственный операнд или переменная. Фактически эта команда уменьшает ESP на размер источника в байтах (2 или 4) и копирует содержимое источника в память по адресу SS:[ESP].

| Команда | Назначение | Процессор |
|---|---|---|
| **PUSH** источник | Поместить данные в стек | 8086 |

Команда **pop** помещает в приемник слово или двойное слово, находящееся в вершине стека, увеличивая ESP на 2 или 4 соответственно.Приемником может быть регистр общего назначения, сегментный регистр, кроме CS (чтобы загрузить CS из стека, надо воспользоваться командой RET), или переменная.

| Команда | Назначение | Процессор |
|---|---|---|
| **POP** приемник | Считать данные из стека | 8086 |

Команды **pusha (pushad)** помещают в стек регистры в порядке: AX (EAX), CX (ECX), DX (EDX), BX (EBX), SP (ESP), BP (EBP), SI (ESI), DI (EDI). **Popa (popad)** их извлекает. Приемников и источников нет, команды добавлены уже в более поздних реализациях процессора.

| Команда | Назначение | Процессор |
|---|---|---|
| **PUSHA** | Поместить в стек | 80186 |
| **PUSHAD** | все регистры общего назначения | 80386 |

Команды **In** и **Out** предназначены для работы с портами ввода-вывода. Взаимодействие происходит через регистр EAX, AX и AL, а порты обозначаются номерами.
Также в команды пересылки данных Зубков включает **XLAT** и **LEA**, но про них написано отдельно в билетах 15-16.

## 9. Команда сравнения.

CMP <приемник>, <источник>

- Источник - число, регистр или переменная
- Приёмник - регистр или переменная; не может быть переменной одновременно с источником
- Вычитает источник из приёмника, результат никуда не сохраняется, выставляются флаги состояния CF, PF, AF, ZF, SF, OF

Обычно команду CMP используют вместе с командами условного перехода (Jcc), условной пересылки данных (CMOVcc) или условной установки байтов (SETcc), которые позволяют применить результат сравнения, не обращая внимания на детальное значение каждого флага

*Из логических команд TEST <приемник>, <источник> - логическое сравнение.*

- *Аналог AND, но результат не сохраняется*
- *Выставляются флаги SF, ZF, PF. OF и CF обнуляются, значение AF не определено*
- *TEST, так же как и CMP, используется в основном в сочетании с командами условного перехода (Jcc), условной пересылки данных (CMOVcc) и условной установки байтов (SETcc).*

1. *CF (carry flag) - флаг переноса*
   *устанавливается в 1, если результат предыдущей операции не уместился в приемник и произошел перенос из старшего бита или если требуется заем при вычитании. Иначе 0.*
   *(Например, после сложения слова OFFFFh и 1, если регистр, в который надо поместить результат, - слово, в него будет записано OOOOh и флаг CF = 1.)*
2. *PF (parity flag) - флаг четности*

*Устанавливается в 1, если младший байт результата предыдущей команды содержит четное число битов, равных 1. Это не то же самое, что делимость на два. Число делится на два без остатка, если его самый младший бит равен нулю, и не делится, когда он равен 1. (в логических операциях)*

3. *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. Этот флаг используется автоматически командами двоично-десятичной коррекции.*
4. *ZF (zero flag) - флаг нуля*
   *устанавливается в 1, если результат предыдущей команды равен 0.*
5. *SF (sign flag) - флаг знака*
   *всегда равен старшему биту результата.*
6. *OF (overflowflag) - флаг переполнения*
   *устанавливается в 1, если результат предыдущей операции над числами со знаком выходит за допустимые для них пределы. Например, если при сложении двух положительных чисел получается число со старшим битом, равным единице, то есть отрицательное, и наоборот.*

## 10. Команды условной и безусловной передачи управления.

### Команда безусловной передачи управления JMP

- Передаёт управление в другую точку программы (на другой адрес памяти), не сохраняя какой-либо информации для возврата.
- Операнд - непосредственный адрес (вставленный в машинный код) (в программах используют имя метки, установленной перед командой, на которую выполняется переход), адрес в регистре или адрес в переменной.
- Виды переходов
  - short (короткий) -128 .. +127 байт
  - near (ближний) в том же сегменте (без изменения регистра CS)
  - far (дальний) в другой сегмент (с изменением значения в регистре CS)
    Дальний переход может выполняться и в тот же самый сегмент при условии, что в сегментной части операнда указано число, совпадающее с текущим значением CS;
    - *переход с переключением задачи - передача управления другой задаче в многозадачной среде. Этот вариант будет рассмотрен в разделе, посвященном защищенному режиму.*
- Для короткого и ближнего переходов просто изменяется значение IP
  - непосредственный операнд (константа) прибавляется к IP (относительный переход)
  - Операнды - регистры и переменные заменяют старое значение в IP (CS:IP) (как mov)

*Выполняя дальний переход в реальном, виртуальном и защищенном режимах (при переходе в сегмент с теми же привилегиями), команда JMP просто загружает новое значение в EIP и новый селектор сегмента кода в CS, используя старшие 16 бит операнда как новое значение для CS и младшие 16 или 32 бит в качестве значений IP или EIP*

### Команды условных переходов J..

- Переход типа short или near, если выполняется соответствующее условие (состояние флагов)
- Обычно используются в паре с CMP

*Операнд для всех команд из набора Jcc - 8-битное или 32-битное смещение относительно текущей команды.*

*Команды Jcc не поддерживают дальних переходов, поэтому, если требуется выполнить условный переход на дальнюю метку, необходимо использовать команду из набора Jcc с обратным условием и дальний JMP, как, например:*

```
        cmp     ax,0
        jne     local_1
        jmp     far_label       ; Переход, если AX = 0.
local_1:
```

- Термины “выше” и “ниже” - при сравнении беззнаковых чисел
- Термины “больше” и “меньше” - при сравнении чисел со знаком

| Команда | Описание | Состояние флагов для выполнения перехода |
|---|---|---|
| JO | Есть переполнение | OF = 1 |
| JNO | Нет переполнение | OF = 0 |
| JS | Есть знак | SF = 1 |
| JNS | Нет знака | SF = 0 |
| JE, JZ | Если равно/если ноль | ZF = 1 |
| JNE, JNZ | Не равно/не ноль | ZF = 0 |
| JP/JPE | Есть чётность / чётное | PF = 1 |
| JNP/JPO | Нет чётности / нечётное | PF = 0 |
| JCXZ | CX = 0 | - |

| Команда | Описание | Состояние флагов для выполнения перехода | Знаковый |
|---|---|---|---|
| JB<br>JNAE<br>JC | Если ниже<br>Если не выше и не равно<br>Если перенос | CF = 1 | нет |
| JNB<br>JAE<br>JNC | Если не ниже<br>Если выше или равно<br>Если нет переноса | CF = 0 | нет |
| JBE<br>JNA | Если ниже или равно<br>Если не выше | CF = 1 или ZF = 1 | нет |
| JA<br>JNBE | Если выше<br>Если не ниже и не равно | CF = 0 и ZF = 0 | нет |

| Команда | Описание | Состояние флагов для выполнения перехода | Знаковый |
|---|---|---|---|
| JL<br>JNGE | Если меньше<br>Если не больше и не равно | SF <> OF | да |
| JGE<br>JNL | Если больше или равно<br>Если не меньше | SF = OF | да |
| JLE<br>JNG | Если меньше или равно<br>Если не больше | ZF = 1 или SF <> OF | да |
| JG<br>JNLE | Если больше<br>Если не меньше и не равно | ZF = 0 и SF = OF | да |

### JCXZ метка - Переход, если CX = 0

Выполняет ближний переход на указанную метку, если регистр CX равен нулю. не могут выполнять дальних переходов. *Проверка равенства CX нулю, например, может потребоваться в начале цикла, организованного командой LOOPNE, - если в него войти с CX = 0, то он будет выполнен 65 535 раз*

### Циклы

Метка не может быть дальше -128..127 байт от команды (short)

1. LOOP <метка> - уменьшает CX и выполняет "короткий" переход на метку, если CX не равен нулю.

   Команда LOOP полностью эквивалентна паре команд

   ```
   dec     ecx
   jnz     метка
   ```

   Но LOOP короче этих двух команд на один байт и не изменяет значения флагов.

2. LOOPE/LOOPZ <метка> - цикл "пока равно"/"пока ноль" (zf=1),
   LOOPNE/LOOPNZ <метка> - цикл "пока не равно"/"пока не ноль" (zf=0)

Декрементируют CX и выполняют переход, если CX не ноль и если выполняется условие (ZF).

Сами команды LOOPcc не изменяют значений флагов, так что ZF должен быть установлен (или сброшен) предшествующей командой.

11. Арифметические команды.
12. Двоично-десятичная арифметика. - ниже

*Все команды этого раздела, кроме команд деления и умножения, изменяют флаги OF, SF, ZF, AF, CF, PF в соответствии с назначением каждого из них*

- Цел, дв ADD <приемник>, <источник>- арифметическое сложение приёмника и источника. Сумма помещается в приёмник, источник не изменяется.
- Цел, дв SUB <приемник>, <источник> - арифметическое вычитание источника из приёмника, помещает разность в приемник.
  То же самое
  ADD, SUB не делают различий между знаковыми и беззнаковыми числами, *но, употребляя значения флагов CF (перенос при сложении чисел без знака), OF (перенос при сложении чисел со знаком) и SF (знак результата), разрешается применять ее и для тех, и для других.*
  Приемник может быть регистром или переменной, источник - числом, регистром или переменной, но нельзя использовать переменную одновременно и для источника, и для приемника.

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

- Дв 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

- Цел, дв MUL <источник>- беззнаковое умножение.
  Выполняет умножение содержимого источника (регистр или переменная) и регистра AL, AX, EAX (в зависимости от размера источника) и помещает результат в AX, DX:AX, EDX.-EAX соответственно. *Если старшая половина результата (AH, DX, EDX) содержит только нули (результат целиком поместился в младшую половину), флаги CF и OF устанавливаются в 0, иначе - в 1. Значение остальных флагов (SF, ZF, AF и PF) не определено.*

- Дв IMUL Умножение чисел со знаком:

  IMUL <источник> - аналогичная команде MUL

  IMUL <приемник>, <источник> - источник (число, регистр или переменная) умножается на приемник (регистр), и результат заносится в приемник

  IMUL <приемник>, <источник1>, <источник2> - источник 1 (регистр или переменная) умножается на источник 2 (число), и результат заносится в приемник (регистр).

  *Во всех трех вариантах считается, что результат может занимать в два раза больше места, чем размер источника. В первом случае приемник автоматически оказывается очень большим, но во втором и третьем случаях существует вероятность переполнения и потери старших битов результата. Флаги OF и CF будут равны единице, если это произошло, и нулю, если результат умножения поместился целиком в приемник (во втором и третьем случаях) или в младшую половину приемника (в первом случае). Значения флагов SF, ZF, AF и PF после команды IMUL не определены*

- Цел, дв DIV <источник>- целочисленное беззнаковое деление.
  Выполняет целочисленное деление без знака AL, AX или EAX (в зависимости от размера источника) на источник (регистр или переменная но не число) и помещает результат в AL, AX или EAX, а остаток - в AH, DX или EDX соответственно. Результат всегда округляется в сторону нуля, абсолютное значение остатка меньше абсолютного значения делителя. Флаги CF, OF, SF, ZF, AF и PF после этой команды не определены, а переполнение или деление на ноль вызывает исключение #DE (ошибка при делении) в защищенном режиме и прерывание 0-в реальном
- Дв IDIV <источник> - Целочисленное деление со знаком:

Результат округляется в сторону нуля, знак остатка совпадает со знаком делимого.

*Если результат деления не помещается в регистр-приемник (такое может произойти при делении больших чисел на маленькие), происходит вызов прерывания 0 в реальном режиме (исключение #DE в защищенном). То же самое происходит и при попытке поделить число на ноль.*

*Выполняет целочисленное деление со знаком AL, AX или EAX (в зависимости от размера источника) на источник (регистр или переменная) и помещает результат в AL, AX или EAX, .а остаток - в AH, DX или EDX соответственно.*

- Цел, дв INC <приемник> DEC <приемник>

Увеличивает/уменьшает приёмник (регистр/переменная) на 1.

В отличие от ADD, не изменяет CF.

Арифметические OF, SF, ZF, AF, PF устанавливаются в соответствии с результатом.

Обе команды работают быстрее ADD и SUB соответственно, потому что занимают 1 байт, а не 3

- дв NEG <приемник> - изменение знака.

Переводит число в дополнительный код (инверсия и прибавление единицы)

*в приемнике (регистр или переменная). Эта операция эквивалентна обращению знака операнда, если рассматривать его как число со знаком. Если приемник равен нулю, флаг CF устанавливается в 0, иначе - в 1. Остальные флаги (OF, SF, ZF, AF, PF) назначаются в соответствии с результатом операции.*

- ***CMP*** *приемник, источник – Сравнение*
  *Сравнивает приемник и источник и устанавливает флаги. Действие осуществляется путем вычитания источника (число, регистр или переменная) из приемника (регистр или переменная; приемник и источник не могут быть переменными одновременно), причем результат вычитания никуда не записывается. Единственным следствием работы этой команды оказывается изменение флагов CF, OF, SF, ZF, AF и PF. Обычно команду CMP используют вместе с командами условного перехода (Jcc), условной пересылки данных (CMOVcc) или условной установки байтов (SETcc),*

## Десятичные

## DAA, DAS, AAA, AAS, AAM, AAD

- Неупакованное двоично-десятичное число - байт от 00h до 09h. (Десятичная цифра, хранящаяся в байте)
- Упакованное двоично-десятичное число - байт от 00h до 99h (цифры A..F не задействуются). (Две десятичные цифры, хранящиеся в полубайтах одного байта)
- При выполнении арифметических операций необходима коррекция:
- 19h + 1 = 1Ah => 20h

*BCD-представление Существует два типа BCD-представления: ● распакованное BCD-представление; ● упакованное BCD-представление. В распакованном BCD-представлении каждый байт хранит двоичный эквивалент каждой десятичной*

*цифры числа. Четыре инструкции настройки ASCII: AAA, AAS, AAM и AAD; также могут использоваться с распакованным BCD-представлением. В упакованном BCD-представлении каждая цифра сохраняется с использованием 4-х бит. Две десятичные цифры упаковываются в 1 байт. Есть две инструкции для обработки этих чисел: DAA (англ. «Decimal Adjust After Addition») — десятичная настройка после добавления; DAS (англ. «Decimal Adjust After Subtraction») — десятичная настройка после вычитания. Обратите внимание, что в упакованном BCD-представлении отсутствует поддержка операций умножения и деления*

1. DAA, DAS – десятичная коррекция после сложения, вычитания (упакованных)

Например, если AL содержит число 19h, последовательность команд inc al; daa приведет к тому, что в AL окажется 20h (а не 1 Ah, как было бы после INC)

Например, если AL содержит число 20h, последовательность команд dec al; das приведет к тому, что в регистре окажется 19h (а не IFh, как было бы после DEC).

*Флаги AF и CF устанавливаются, если в ходе коррекции происходил перенос (заем) из первой или второй цифры. SF, ZF и PF устанавливаются в соответствии с результатом, флаг OF не определен*

*DAA выполняет следующие действия: 1. Если младшие четыре бита AL больше 9 или флаг AF = 1, то AL увеличивается на 6, CF устанавливается, если при этом сложении произошел перенос, и AF устанавливается в 1. 2. Иначе AF = 0. 3. Если теперь старшие четыре бита AL больше 9 или флаг CF = 1, то AL увеличивается на 60h и CF устанавливается в 1. 4. Иначе CF = 0.*

*DAS выполняет следующие действия: 1. Если младшие четыре бита AL больше 9 или флаг AF = 1, то AL уменьшается на 6, CF устанавливается, если при этом вычитании произошел заем, и AF устанавливается в 1. 2. Инача AF = 0. 3. Если теперь старшие четыре бита AL больше 9 или флаг CF — 1, то AL уменьшается на 60h и CF устанавливается в 1. 4. Иначе CF = 0.*

Известный пример необычного использования этой команды ~ самый компактный вариант преобразования шестнадцатеричной цифры в ASCII-код соответствующего символа (более длинный и очевидный вариант этого преобразования рассматривался в описании команды XLAT):

```
cmp al,10
sbb al,69h
das
```

2. AAA, AAS - ASCII коррекция после сложения, вычитания (неупакованных) например, если при сложении 05 и 06 в AX окажется число 000Bh, то команда AAA скорректирует его в 0l0lh (неупакованное десятичное 11).

   *Флаги CF и OF устанавливаются в 1, если произошел перенос (заем) из AL в AH, в противном случае они равны нулю. Значения флагов OF, SF, ZF и PF не определены.*

Для того, чтобы преобразовать содержимое регистра AL к ASCII-формату, необходимо после команды AAA выполнить команду OR AL,0x30h (то есть сделать читаемым числом). Пример:

```
sub AH, AH    ; очистка AH
mov AL, '6'   ; AL = 0x36h
add AL, '8'   ; AL = 0x36h + 0x38h = 0x6Eh
aaa           ; AX = 0x0104h
or AL,30H     ; AL = 0x34h = '4'
```

При положительном результате вычитания это выглядит следующим образом:

```
sub AH,AH    ; очистка AH
mov AL,'9'   ; AL = 0x39h
sub AL,'3'   ; AL = 0x39h - 0x33h = 0x06h
aas          ; AX = 0x0006h
or AL,30H    ; AL = 0x36h = '6'
```

при вычитании с получением результа меньше нуля:

```
sub AH,AH    ; очистка AH
mov AL,'3'   ; AL = 0x33h
sub AL,'9'   ; AL = 0x33h - 0x39h = 0xFAh
aas          ; AX = 0xFF04h
or AL,30H    ; AL = 0x34h = '4' (хз почему)
```

3. AAM - ASCII коррекция после умножения (неупакованных)

```
mov     al,5
mov     bl,5            ; Умножить 5 на 5.
mul     bl              ; Результат в AX - 0019h.
aam                     ; Теперь AX содержит 0205h.
```

4. AAD - ASCII коррекция перед делением (неупакованных)

```
mov     ax,0205h        ; 25 в неупакованном формате.
mov     bl,5
aad                     ; Теперь в AX находится 19h.
div     bl              ; AX = 0005.
```

## 13. Команды побитовых операций. Логические команды.

### Логические операции

*Приёмник (регистр или переменная), источник (число, регистр, переменная) но не переменные одновременно.*

- AND <приемник>, <источник> - побитовое (логическое) И. Часто – для выборочного обнуления отдельных битов: AND al, 00001111b . обнуляет старшие 4 бита al, остальные не изменятся
- OR <приемник>, <источник> - побитовое “ИЛИ”. Часто – для выборочной установки отдельных битов: OR al, 00001111b - младшие 4 бита al 1.
  *Для этих 2 Флаги OF и CF обнуляются, SF, ZF и PF устанавливаются в соответствии с результатом, AF не определен*
- XOR <приемник>, <источник> - побитовое исключающее “ИЛИ”. (0 если равны) XOR AX, AX – обнуление, (xor ax, bx; xor bx.ax; xor ax.bx) Меняет местами содержимое AX и BX.)
- NOT <приемник> - инверсия. Флаги не затрагиваются
- TEST <приемник>, <источник> - логическое сравнение.
  - Аналог AND, но результат не сохраняется
  - Выставляются флаги SF, ZF, PF. OF и CF обнуляются, значение AF не определено

- TEST, так же как и CMP, используется в основном в сочетании с командами условного перехода (Jcc), условной пересылки данных (CMOVcc) и условной установки байтов (SETcc).

## Сдвиговые операции

*Операция сдвига на 1 эквивалентна умножению (сдвиг влево) или делению (сдвиг вправо) на 2.*

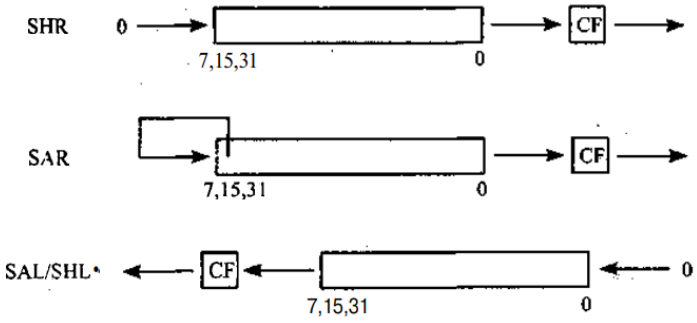
*Логический (старший – обычный), арифметический (старший – знак), циклический сдвиг.*

*Аргументы: приемник (регистр или переменная), счетчик (число или регистр CL, из которого учитываются только младшие 5 бит, принимающие значения от 0 до 31)*

- SAL и SAR (арифметический) побитовый сдвиг числа влево и вправо соответственно.
- SHL и SHR (логический) побитовый сдвиг заданного числа влево или вправо через флаг CF.

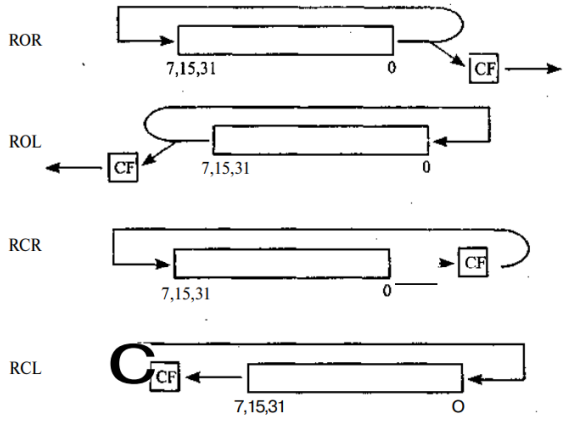
    - SAL=SHL на каждый шаг сдвига старший бит заносится в CF, все биты сдвигаются влево на одну позицию, и младший бит обнуляется.
    - SHR зануляет старший бит, SAR - сохраняет (знак)
    - SHR осуществляет прямо противоположную операцию: младший бит заносится в CF, все биты сдвигаются на 1 вправо, старший бит обнуляется. (=беззнаковому целочисленному делению на 2)
    - SAR действует по аналогии с SHR, только старший бит не обнуляется, а сохраняет предыдущее значение (=знаковому делению на 2, но, в отличие от IDIV, округление происходит не в сторону нуля, а в сторону отрицательной бесконечности. Так, если разделить —9 на 4 с помощью IDIV, получится -2 (и остаток -1), а если выполнить арифметический сдвиг вправо числа -9 на 2, результатом будет -3.

*В процессорах 8086 в качестве второго операнда можно было задавать лишь число 1 и при использовании CL учитывать все биты, а не только младшие 5, но уже начиная с 80186 эти команды приняли свой окончательный вид.*




- ROL и ROR циклический сдвиг вправо/влево  N
- RCL и RCR циклический сдвиг через CF  N+1
    - ROR (ROL) перемещают каждый бит приемника вправо (влево) на одну позицию, за исключением самого младшего (старшего), который записывается в позицию самого старшего (младшего) бита.

- RCR и RCL выполняют аналогичное действие, но включают флаг CF в цикл, как если бы он был дополнительным битом в приемнике




Все эти команды меняют регистр FLAGS.

## Операции над битами и байтами

BT, BTR, BTS, BTC, BSF, BSR, SETcc

Операции над битами и байтами

- BT <база>, <номер>- считать в CF значение бита из битовой строки
  считывает в флаг CF значение бита из битовой строки, определенной первым операндом - битовой базой (регистр или переменная), со смещением, указанным во втором операнде - битовом смещении (число или регистр).
  *Когда первый операнд - регистр, то битовой базой считается бит 0 в названном регистре и смещение не может превышать 15 или 31 (в зависимости от размера регистра); если оно превышает эти границы, в качестве смещения будет использоваться остаток от деления на 16 или 32 соответственно. Если первый операнд - переменная, то в качестве битовой базы нужен бит 0 указанного байта в памяти, а смещение может принимать значения от 0 до 31, если оно установлено непосредственно (старшие биты процессором игнорируются), и от -2^31 до 2^31—1, если оно указано в регистре.*
- BTS <база>, <номер>- установить бит в 1
- BTR <база>, <номер>- - сбросить бит в 0
- BTC <база>, <номер>  - инвертировать бит
- BSF <приемник>, <источник>- прямой поиск бита (от младшего разряда)
- BSR <приемник>, <источник>- обратный поиск бита (от старшего разряда)
  *BSF сканирует источник (регистр или переменная), начиная с самого младшего бита, и записывает в приемник (регистр) номер первого встретившегося бита, равного 1. Команда BSR сканирует источник, начиная с самого старшего бита, и возвращает номер первого встретившегося ненулевого бита, считая от нуля. То есть, если источник равен 0000 0000 0000 0010Ь, то BSF возвратит 1, а BSR - 14.*

- SETcc <приемник> выставляет приёмник (8-битный регистр или переменная размером в 1 байт) в 1 или 0 в зависимости от условия, аналогично Jcc

- Термины “выше” и “ниже” - при сравнении беззнаковых чисел
- Термины “больше” и “меньше” - при сравнении чисел со знаком

| Команда | Описание | Состояние флагов для выполнения перехода |
|---|---|---|
| JO | Есть переполнение | OF = 1 |
| JNO | Нет переполнение | OF = 0 |
| JS | Есть знак | SF = 1 |
| JNS | Нет знака | SF = 0 |
| JE, JZ | Если равно/если ноль | ZF = 1 |
| JNE, JNZ | Не равно/не ноль | ZF = 0 |
| JP/JPE | Есть чётность / чётное | PF = 1 |
| JNP/JPO | Нет чётности / нечётное | PF = 0 |
| JCXZ | CX = 0 | - |

| Команда | Описание | Состояние флагов для выполнения перехода | Знаковый |
|---|---|---|---|
| JB<br>JNAE<br>JC | Если ниже<br>Если не выше и не равно<br>Если перенос | CF = 1 | нет |
| JNB<br>JAE<br>JNC | Если не ниже<br>Если выше или равно<br>Если нет переноса | CF = 0 | нет |
| JBE<br>JNA | Если ниже или равно<br>Если не выше | CF = 1 или ZF = 1 | нет |
| JA<br>JNBE | Если выше<br>Если не ниже и не равно | CF = 0 и ZF = 0 | нет |

| Команда | Описание | Состояние флагов для выполнения перехода | Знаковый |
|---|---|---|---|
| JL<br>JNGE | Если меньше<br>Если не больше и не равно | SF <> OF | да |
| JGE<br>JNL | Если больше или равно<br>Если не меньше | SF = OF | да |
| JLE<br>JNG | Если меньше или равно<br>Если не больше | ZF = 1 или SF <> OF | да |
| JG<br>JNLE | Если больше<br>Если не меньше и не равно | ZF = 0 и SF = OF | да |

## 14. Команды работы со строками.

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

За один раз обрабатывается один байт/слово/двойное слово (SB - байт, SW - слово, SD - двойное слово). Для того чтобы команда выполнялась над всей строкой, необходим один из префиксов повторения операций.

**Префиксы:** REP/REPE/REPZ/REPNE/REPNZ без параметров

Любой из префиксов выполняет следующую за ним команду строковой обработки столько раз, сколько указано в регистре ECX (или CX, в зависимости от разрядности адреса), уменьшая его при каждом выполнении команды на 1. Кроме того, REPZ и REPE прекращают повторения команды, если флаг ZF сброшен в 0, а REPNZ и REPNE прекращают повторения, если флаг ZF установлен в 1.

*Пример использования: REP LODS AX*

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

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

- MOVS <приемник>, <источник> /MOVSB/MOVSW - копирование

При использовании формы записи MOVS ассемблер сам определяет по типу указанных операндов, какую из трех форм этой команды выбрать. Используя MOVS с операндами, разрешается заменить регистр DS другим с помощью префикса замены сегмента (ES:, GS:, FS:, CS:, SS:), регистр ES заменить нельзя. После выполнения команды регистры SI и DI увеличиваются на 1 или 2 (если копируются байты, слова), когда флаг DF = 0, и уменьшаются, когда DF = 1.

- CMPS<приемник>, <источник>/CMPSB/CMPSW - сравнение и установка флагов аналогично CMP
  *REPNE/REPNZ или REPE/REPZ - до первого совпадения в строках, а во втором — до первого несовпадения.*
- SCAS <приемник> /SCASB/SCASW - сканирование (сравнение ES:DI с AL/AX) как выше
- LODS<источник> /LODSB/LODSW - чтение (в AL/AX)
- STOS<приемник> /STOSB/STOSW- запись (из AL/AX)

## 15. Команда трансляции по таблице.

XLAT [адрес] или XLATB - Трансляция в соответствии с таблицей

- Помещает в AL байт из таблицы по адресу DS(по лекции)ES(по Зубкову):BX со смещением относительно начала таблицы, равным AL.
- В качестве аргумента для XLAT в ассемблере можно указать имя таблицы, но эта информация никак не используется процессором и служит только в качестве комментария.
- Если в адресе явно указан сегментный регистр, он будет использоваться вместо DS.

*Короче говоря, XLATB -> AL = DS:[(E)BX + AL]*

**Например**, можно написать следующий вариант преобразования шестнадцатеричного числа в ASCII-код соответствующего ему символа:

```
mov al, 0Ch
mov bx, offset htable
xlatb
```

если в сегменте данных, на который указывает регистр ES, было записано

```
htable db "0123456789ABCDEF"
```

то теперь AL содержит не число 0Ch, а ASCII-код буквы C.

*Применение: XLAT применяется для перекодировки значений. Команду XLAT хорошо использовать при кодировании и декодировании текстовых данных. С помощью этой команды программа может организовать простую замену кодов символов.*

## 16. Команда вычисления эффективного адреса.

LEA <приемник>, <источник> - вычисление эффективного адреса (значение смещения)
Вычисляет эффективный адрес источника (переменная) и помещает его в приёмник (регистр).

- Позволяет вычислить адрес, описанный сложным методом адресации, например, по базе с индексированием
- Иногда используется для быстрых арифметических вычислений: lea bx, [bx+bx*4] lea bx, [ax+12]
- Эти вычисления занимают меньше памяти, чем соответствующие MOV и ADD, и не изменяют флаги.

*По сути, является аналогом операции mov ПРИЕМНИК, offset ИСТОЧНИК*

*Если адрес - 32-битный, а регистр-приемник - 16-битный, старшая половина вычисленного адреса теряется, если наоборот, приемник - 32-битный, а адресация - 16-битная, то вычисленное смещение дополняется нулями.*

*Команду LEA часто используют для быстрых арифметических вычислений, например умножения:*

```
lea     bx,[ebx+ebx*4] ; BX = EBX x 5
```

*или сложения:*

```
lea     ebx,[eax+12]    ;EBX = EAX + 12
```

*(эти команды меньше, чем соответствующие MOV и ADD, и не изменяют флаги)*

## 17. Структура программы на языке ассемблера. Модули. Сегменты.

Программа на языке ассемблера состоит из строк, имеющих следующий вид:
метка команда/директива операнды ; комментарий Причем все эти поля необязательны.

- **Модули** (файлы исходного кода)
  Завершение описания модуля …**END [точка_входа]**
  - ✓ точка_входа - имя метки, объявленной в сегменте кода и указывающее на команду, с которой начнётся исполнение программы.
  - ✓ Если в программе несколько модулей, только один может содержать точку входа.
  - ✓ Глобальные объявления
    - Public – переменная будет доступна из других модулей
    - comm,
    - extrn – подключает эту метку
    - global

  - **Сегменты** (описание блоков памяти)
    Составляющие программного кода

- команды процессора;
- инструкции описания структур данных, выделения памяти для переменных и констант;
- макроопределения.

## Сегменты

- Любая программа состоит из сегментов
- Виды сегментов:
  - сегмент кода
  - сегмент данных
  - сегмент стека
- ассемблер позволяет изменять устройство программы как угодно - помещать данные в сегмент кода, разносить код на множество сегментов, помещать стек в один сегмент с данными или вообще использовать один сегмент для всего.
- Описание сегмента - директивы SEGMENT и ENDS. Все пять операндов директивы SEGMENT необязательны

```
имя SEGMENT [READONLY] [выравнивание] [тип] [разрядность] ['класс']

    …

имя ENDS
```

- … - код: команды, данные – объявление данных с помощью директив выделения памяти

1. Readonly – выдаст ошибку на этапе компиляции при попытке записи в masm
2. Выравнивание – с каких адресов начинается сегмент
   - BYTE, WORD (2), DWORD (4), **PARA**(16, ум.), PAGE(256)
3. Тип
   - **PRIVATE**(ум.) — сегмент не будет объединяться с другими сегментами
   - PUBLIC — все такие сегменты с одинаковым именем, но разными классами будут объединены в один, друг за другом. Новый объединенный сегмент будет целым и непрерывным. Все адреса (смещения) объектов будут вычисляться относительно начала этого нового сегмента;
   - STACK — (как и public) определение сегмента стека. Заставляет компоновщик соединить все одноименные сегменты и вычислять адреса в этих сегментах относительно регистра ss.
   - 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 xxxx(номер подпараграфа) — располагает сегмент по абсолютному адресу параграфа (параграф — объем памяти, кратный 16; поэтому последняя шестнадцатеричная цифра адреса параграфа равна 0). Такие сегменты обычно содержат только метки, указывающие на области памяти, которые могут потребоваться программе; Все метки и адреса в определенном таким образом сегменте отсчитываются относительно заданного абсолютного адреса;

4. Класс – любая метка, взятая в одинарные кавычки. и. Все сегменты с одинаковым классом, даже сегменты типа PRIVATE, будут расположены в исполняемом файле непосредственно друг за другом.
5. *Разрядность. Этот операнд может принимать значения USE16 и USE32. Размер сегмента, описанного как USE16, не может превышать 64 Кб, и все команды и адреса в этом сегменте считаются 16-битными. В этих сегментах все равно можно применять команды, использующие 32-битные регистры или ссылающиеся на данные в 32-битных сегментах, но они будут использовать префикс изменения разрядности операнда или адреса и окажутся длиннее и медленнее. Сегменты USE32 могут занимать до 4 Гб, и все команды и адреса в них по умолчанию 32-битные. Если разрядность сегмента не указана, по умолчанию используется USE 16 при условии, что перед .MODEL не применялась директива задания допустимого набора команд .386 или старше.*

Для обращения к любому сегменту следует сначала загрузить его сегментный адрес (или селектор в защищенном режиме) в какой-нибудь сегментный регистр.

## Сегментный префикс. Директива ASSUME регистр:имя

- Для обращения к переменной процессору необходимо знать обе составляющие адреса: и сегментную, и смещение. Пример полной записи - DS:Var1
- Директива ASSUME сегмента устанавливает значение сегментного регистра по умолчанию

```
Data1 SEGMENT WORD 'DATA'
Var1  DW   0
Data1 ENDS

Data2 SEGMENT WORD 'DATA'
Var2  DW   0
Data2 ENDS

Code  SEGMENT WORD 'CODE'
      ASSUME  CS:Code
ProgramStart:
      mov   ax,Data1
      mov   ds,ax
      ASSUME  DS:Data1
      mov   ax,Data2
      mov   es,ax
      ASSUME  ES:Data2
      mov   ax,[Var2]

      .
      .
      .
Code  ENDS
END   ProgramStart
```

*Директива ASSUME указывает ассемблеру, с каким сегментом или группой сегментов связан тот или иной сегментный регистр. В качестве операнда «имя» могут использоваться имена сегментов, имена групп, выражения с оператором SEG или слово «NOTHING», означающее отмену действия предыдущей ASSUME для данного регистра. Эта директива не изменяет значений сегментных регистров, а только позволяет ассемблеру проверять допустимость ссылок и самостоятельно вставлять при необходимости префиксы переопределения сегментов.*

## Модели памяти

Задаются директивой .model модель, язык, модификатор

- Модели:
  - TINY - один сегмент на всё (.com)
    *код, данные и стек размещаются в одном и том же сегменте размером до 64 Кб. Эта модель памяти чаще всего используется при написании на ассемблере небольших программ;*
  - SMALL - код в одном сегменте, данные и стек - в другом
  - COMPACT - допустимо несколько сегментов данных

*код размещается в одном сегменте, а для хранения данных могут использоваться несколько сегментов, так что для обращения к данным требуется указывать сегмент и смещение (данные дальнего типа);*
    - MEDIUM - код в нескольких сегментах, данные - в одном
      *поэтому для доступа к данным используется только смещение, а вызовы подпрограмм применяют команды дальнего вызова процедуры*
    - LARGE, HUGE – и код, и данные могут занимать несколько сегментов
    - FLAT - то же, что и TINY, но используются 32-битные сегменты, так что максимальный размер сегмента, содержащего и данные, и код, и стек, - 4 Мб.
- Язык - C, PASCAL, BASIC, SYSCALL, STDCALL. Для связывания с ЯВУ и вызова подпрограмм.
- Модификатор - **NEARSTACK**/FARSTACK. Во втором случае сегмент стека не будет объединяться в одну группу с сегментами данных.
- Определение модели позволяет использовать сокращённые формы директив определения сегментов.

*После того как модель памяти установлена, вступают в силу упрощенные директивы определения сегментов, объединяющие действия директив SEGMENT и ASSUME. Кроме того, сегменты, объявленные упрощенными директивами, не требуется закрывать директивой ENDS - они закрываются автоматически, как только ассемблер обнаруживает новую директиву определения сегмента или конец программы.*

*Директива .CODE описывает основной сегмент кода .code имя_сегмента эквивалентно _TEXT segment word public 'CODE'*

*Директива .STACK описывает сегмент стека и эквивалентна директиве STACK segment para public 'stack' Необязательный параметр указывает размер стека. По умолчанию он равен 1 Кб.*

*.data Описывает обычный сегмент данных и соответствует директиве _DATA segment word public 'DATA'*

**Посмотри также про директивы в конце**

## 18. Макроопределения.

Макроопределение
(макрос) - именованный участок программы, который ассемблируется каждый раз, когда его имя встречается в тексте программы.

Роль макросов в ассемблере такая же, как макросов в си. Очень гибкий и мощный инструмент, чтобы писать код общего вида, который во время работы препроцессора будет заменяться на конкретные выражения.

- Определение:
  имя MACRO параметры
  ……
  ENDM
- Пример:
  load_reg MACRO register1, register2
  push register1

pop register2
ENDM
+ может вызываться с параметре и будет проще различать (в отличие от процедур)

### Директивы условного ассемблирования

- IF:
  IF c1 … ELSEIF c2 … ELSE … ENDIF
- IFB параметр - истинно, если параметр не определён
- IFNB параметр - истинно, если параметр определён
- IFIDN s1, s2- истинно, если строки совпадают
- IFDIF s1, s2 - истинно, если строки разные
- IFDEF/IFNDEF имя - истинно, если имя объявлено/не объявлено

### Блоки повторения

Второе название в WASM

- REPT число … ENDM - повтор фиксированное число раз

*Например, если требуется создать массив байтов, проинициализированный значениями от 0 до 0FFh*

```
hexnumber       = 0
hextable        label  byte             ; Имя массива.
                rept    256             ; Начало блока.
                db      hexnumber       ; Эти две строки ассемблируются
hexnumber       = hexnumber+1           ; 256 раз.
                endm
```

- IRP или FOR:

```
IRP form,<fact_1[,fact_2,...]> … ENDM
```

Подстановка фактических параметров по списку на место формального. *Блок, описанный директивой IRP, будет вызываться столько раз, сколько значений указано в списке (в угловых скобках), и при каждом повторении будет определена метка с именем параметр, равная очередному значению из списка. Например, следующий блок повторений сохранит в стек регистры AX, BX, CX*

```
irp     reg,<ax,bx,cx,dx>
push    reg
endm
```

- IRPC или FORC:

```
IRPC form,fact … ENDM
```

Подстановка символов строки на место формального параметра. *при каждом повторении будет определена метка с именем параметр, равная очередному символу из строки. Если строка содержит пробелы или другие символы, отличные от разрешенных для меток, она должна быть заключена в угловые скобки. Например, следующий блок задает строку в памяти, располагая после каждого символа строки атрибут 0Fh (белый символ на черном фоне), так что эту строку впоследствии можно будет скопировать прямо в видеопамять.*

```
irpc    character,<строка символов>
db      '&character&',0Fh
endm
```

- WHILE: WHILE cond … ENDM

**Макрооперации (макрооператоры)**
специальные операторы, которые действуют только внутри макроопределений и блоков повторений.

- % - вычисление выражение перед представлением числа в символьной форме
  *указывает, что находящийся за ним текст является выражением и должен быть вычислен. Обычно это требуется для того, чтобы передавать в качестве параметра в макрос не само выражение, а его результат.*
- <> - подстановка текста без изменений
  *весь текст, заключенный в эти скобки, рассматривается как текстовая строка, даже если он содержит пробелы или другие разделители. Как мы уже видели, этот макрооператор используется при передаче текстовых строк в качестве параметров для макросов. Другое частое применение угловых скобок - передача списка параметров вложенному макроопределению или блоку повторений.*
- & - склейка текста (параметр, переданный в качестве операнда макроопределению или блоку повторений, заменялся значением до обработки строки ассемблером)
  *следующий макрос выполнит команду PUSH EAX, если его вызвать как PUSHREG A:*

```
pushreg  macro    letter
         push     e&letter&x
         endm
```

- ! - считать следующий символ текстом, а не знаком операции
  *используется аналогично угловым скобкам, но действует только на один следующий символ, так что, если этот символ - запятая или угловая скобка, он все равно будет передан макросу как часть параметра*
- ;; - исключение строки из макроса

**Директивы отождествления EQU, TEXTEQU**
Директива для представления текста и чисел:

- Макроимя EQU нечисловой текст и не макроимя ЛИБО число
- Макроимя EQU <операел>
- Макроимя TEXTEQU Операнд

Пример: X EQU [EBP+8] MOV ESI,X

*Директива EQU (как define C) присваивает метке значение, которое определяется как результат целочисленного выражения в правой части. Результатом этого выражения может быть целое число, адрес или любая строка символов:*

```
truth           equ     1
message1        equ     'Try again$'
var2            equ     4[si]
        cmp     ax, truth       ;  cmp ax,1
        db      message1        ;  db 'Try again$'
        mov     ax,var2         ;  mov ax,  4[si]
```

### Директива присваивания =

Директива присваивания служит для создания целочисленной макропеременной или изменения её значения и имеет формат: Макроимя = Макровыражение

- Макровыражение (или Макровыражение, или Константное выражение) - выражение, вычисляемое препроцессором, которое может включать целочисленные константы, макроимена, вызовы макрофункций, знаки операций и круглые скобки, результатом вычисления которого является целое число
- Операции: арифметические (+, -, *, /. MOD), логические, сдвигов, отношения

*Директива = эквивалентна EQU, но определяемая ею метка может принимать только целочисленные значения. Кроме того, метка, указанная этой директивой, может быть переопределена*

*Несмотря на внешнее и функциональное сходство, псевдооператоры EQU, =, TEXTEQU различаются следующим:*

*с помощью псевдооператора EQU идентификатору можно ставить в соответствие как числовые выражения, так и текстовые строки; псевдооператор = может использоваться только с числовыми выражениями; TEXTEQU – только стоки*

*идентификаторы, определенные с помощью псевдооператоров = и TEXTEQU , можно переопределять в исходном тексте программы, определенные с использованием псевдооператора EQU — нельзя.*

*Эти команды одрабатывают на этапе трансляции.*

### Директивы управления листингом

Листинг - файл, формируемый компилятором и содержащий текст ассемблерной программы, список определённых меток, перекрёстных ссылок и сегментов.

- TITLE, SUBTTL - заголовок, подзаголовок на каждой странице
- PAGE высота, ширина
- NAME - имя программы
- .LALL - включение полных макрорасширений, кроме ;;
- .XALL - по умолчанию
- .SALL - не выводить тексты макрорасширений
- .NOLIST - прекратить вывод листинга

### Комментарии

comment @ (или другой символ конца комментария)

… многострочный текст ...

@

## Сравнение макросов с подпрограммами

*Плюсы: Так как текст макрорасширения вставляется на место макрокоманды, то нет затрат времени, как для подпрограмм, на подготовку параметров, передачу управления и выполнение других работ при выполнении программы*

*Минусы: При многочисленных вызовах МО (макроопределения) разрастается объем модуля программы, Фактические значения параметров макрокоманд должны быть известны препроцессору или могли быть вычислены им (нельзя использовать в качестве фактического параметра МО значения переменных или регистров, так как они могут быть известны только при выполнении программы).*

*Замечания. Имена формальных параметров МО-й локализованы в них, т.е. вне определения могут использоваться для обозначения других объектов. Число формальных параметров ограничено лишь длиной строки, обрабатываемой ассемблером. МО-я должны предшествовать обращениям к ним. Нет ограничений, кроме физических, на число предложений в теле МО. В листинге предложениям макрорасширений предшествуют ЦБЗ, указывающие глубину их вложения в макроопределениях.*

19. Подпрограммы. Объявление, вызов.
20. Подпрограммы. Возврат управления.

## Объявление

- Процедурой в ассемблере является все то, что в других языках называют подпрограммами, функциями, процедурами и т. д
- на любой адрес программы можно передать управление командой CALL, и оно вернется к вызвавшей процедуре, как только встретится команда RET.
- Такая свобода выражения легко может приводить к нечитаемым программам, и в язык ассемблера были включены директивы логического оформления процедур.

```
метка          proc    язык тип USES регистры    ; TASM
или
метка          proc    тип язык USES регистры    ; MASM/WASM
...
               ret
метка      ,   endp
```

Все операнды PROC необязательны.

- Тип может принимать значения NEAR и FAR, и если он указан, все команды RET в теле процедуры будут заменены соответственно на RETN и RETF. По умолчанию подразумевается, что процедура имеет тип NEAR в моделях памяти TINY, SMALL и COMPACT.
- Операнд «язык» действует аналогично такому же операнду директивы .MODEL, определяя взаимодействие процедуры с языками высокого уровня.
- USES - список регистров, значения которых изменяет процедура. Ассемблер помещает в начало процедуры набор команд PUSH, а перед командой RET - набор команд POP, так что значения перечисленных регистров будут восстановлены.

## Вызов

CALL <имя> - вызов процедуры,

1. Сохраняет адрес следующей (текущей по учебнику) команды в стеке (уменьшает SP и записывает по его адресу IP либо CS:IP, в зависимости от размера аргумента)
2. Передаёт управление на значение аргумента.
   *Операндом может быть непосредственное значение адреса (метка в ассемблерных программах), регистр или переменная, содержащие адрес перехода. Если операнд CALL - регистр или переменная, то его значение рассматривается как абсолютное смещение, если операнд - ближняя метка в программе, то ассемблер указывает ее относительное смещение.*

   *При вызове подпрограммы параметры в большинстве случаев помещают в стек, а в EBP записывают текущее значение ESP. Если подпрограмма использует стек для хранения локальных переменных, ESP изменится, но EBP можно будет использовать для того, чтобы считывать Значения параметров напрямую из стека*

   ***Указательные регистры SP, BP***

   *SP - указатель на вершину стека*

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

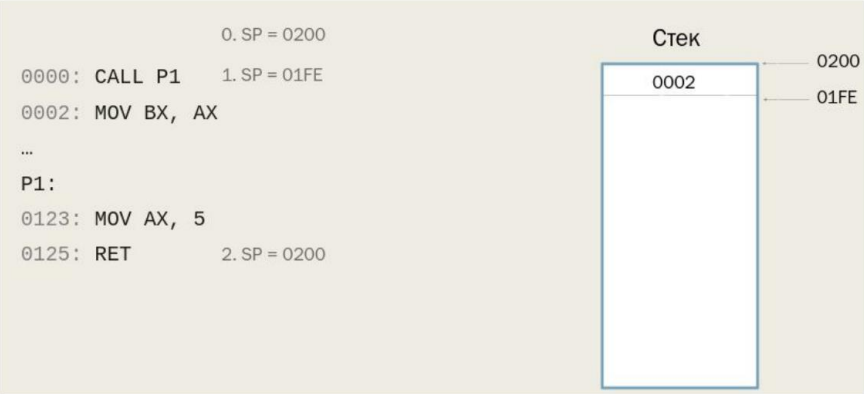
   *BP – base pointer*

   - *Используется в подпрограмме для сохранения "начального" значения SP*
   - *Адресация параметров*
   - *Адресация локальных переменных*

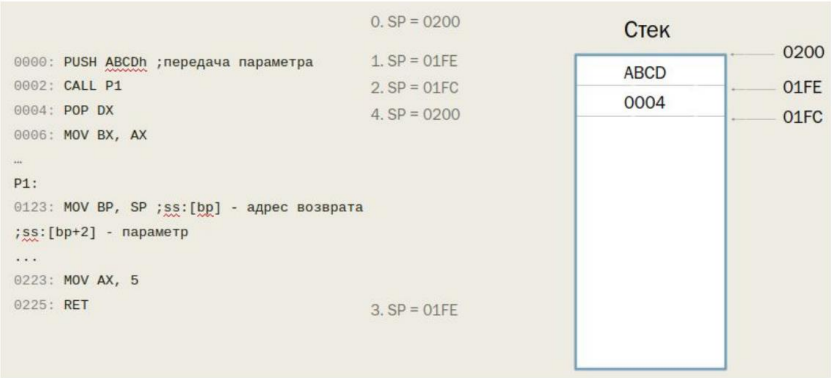
**Возврат**

5. RET/RETN/RETF <число>  (near, far) - возврат из процедуры
   - RETN считывает из стека слово (или двойное слово) и загружает его в IP, увеличивает SP, выполняя тем самым действия, обратные ближнему вызову процедуры командой CALL.
   - Если указан операнд, его значение будет дополнительно прибавлено к SP для очистки стека от параметров
   - Команда RETF загружает из стека IP  и CS, возвращаясь из дальней процедуры.
   - Если в программе указана команда RET, ассемблер заменит ее на RETN или RETF в зависимости от того, как была описана процедура, которую эта команда завершает

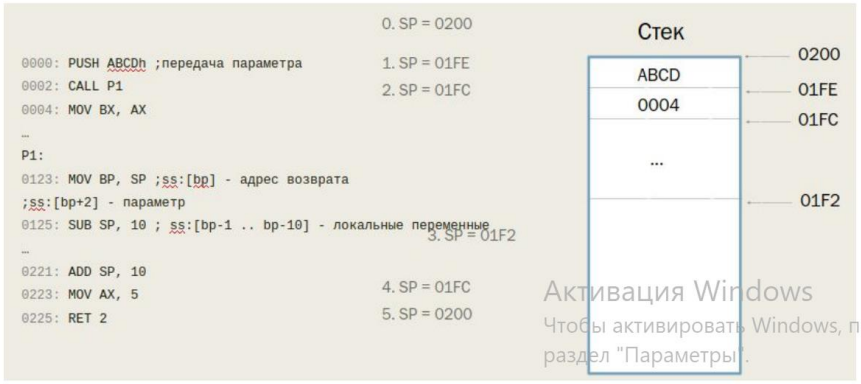
## Пример вызова подпрограммы №1




## Пример вызова подпрограммы №2




## Пример вызова подпрограммы №3




21. Стек. Аппаратная поддержка вызова подпрограмм.

Дополнительно читать про подпрограммы и прерывания

### Указательные регистры SP, BP

SP - указатель на вершину стека

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

BP – base pointer

- Используется в подпрограмме для сохранения "начального" значения SP

- Адресация параметров
- Адресация локальных переменных

## Стек

- LIFO/FILO (last in, first out) - последним пришёл, первым ушёл
- Сегмент стека - область памяти программы, используемая её подпрограммами, а также (вынужденно) обработчиками прерываний
- Использование: 1. Временное хранение переменных 2. Передача параметров вызываемым подпрограммам 3. Сохранение адреса возврата при вызове процедур и прерываний
- *При вызове подпрограммы параметры в большинстве случаев помещают в стек, а в EBP записывают текущее значение ESP. Если подпрограмма использует стек для хранения локальных переменных, ESP изменится, но EBP можно будет использовать для того, чтобы считывать Значения параметров напрямую из стека (их смещения запишутся как EBP + номер параметра).*

## Команды непосредственной работы со стеком

- PUSH <источник> - поместить данные в стек. Уменьшает SP на размер источника и записывает значение по адресу SS:SP. Источник – непосредственный операнд или переменная. Используется вместо 2mov(регистр, регистр) или для временного хранения переменных
- POP <приемник> - считать данные из стека. Считывает значение с адреса SS:SP и увеличивает SP. Приемник – РОН, сегментный регистр (кроме CS – для этого используется команда RET) или переменная
- PUSHA - поместить в стек регистры AX, CX, DX, BX, SP, BP, SI, DI.  (80186)
- POPA - загрузить регистры из стека (SP игнорируется)

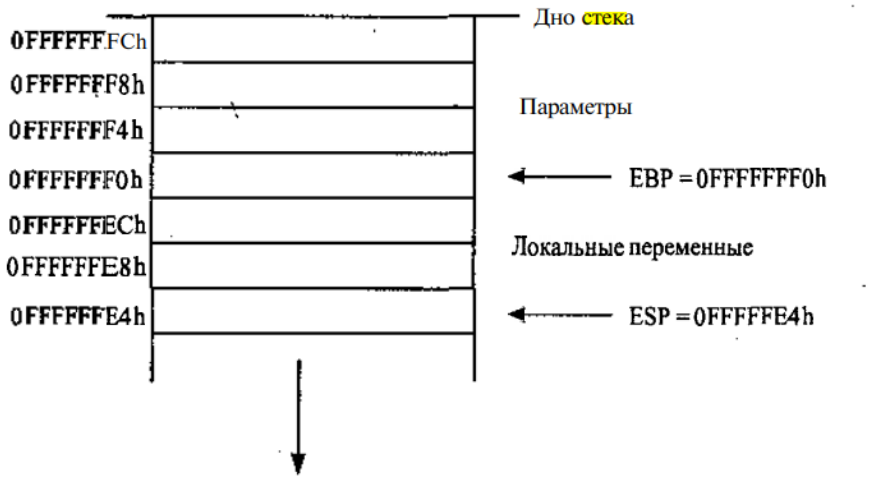
Команда POPA извлекает из стека содержимое регистров общего назначения в следующем порядке: DI, SI, BP, +2, BX, DX, CX, AX. Регистр SP, при чтении из стека с помощью команды POPA, пропускается. Вместо этого регистр стека просто увеличивается на 2 и происходит чтение следующего регистра.

позволяет писать подпрограммы (обычно обработчики прерываний), которые не должны изменять значения регистров по окончании своей работы. В начале такой подпрограммы вызывают команду PUSHA, а в конце - POPA.

дополнительно

*Стек - организованный специальным образом участок памяти, который используется для временного хранения переменных, передачи параметров вызываемым подпрограммам и сохранения адреса возврата при вызове процедур и прерываний*

*Стек располагается в сегменте памяти, описываемом регистром SS, и текущее смещение вершины стека отражено в регистре SP, причем во время записи значение этого смещения уменьшается, то есть он «растет вниз» от максимально возможного адреса*




## 22. Прерывания. Обработка прерываний в реальном режиме работы процессора.

### Прерывания

Прерывания - аппаратный механизм для приостановки выполнения текущей программы и передачи управления специальной программе - обработчику прерывания.

- особая ситуация, когда выполнение текущей программы приостанавливается и управление передаётся программе-обработчику возникшего прерывания.

### Основные виды:

- аппаратные (асинхронные – возникают в случайные моменты времени) (внешние)- события от внешних устройств;

  *Внешние прерывания, в зависимости от возможности запрета, делятся на:*

  - *маскируемые — прерывания, которые можно запрещать установкой соответствующего флага;*
  - *немаскируемые (англ. Non-maskable interrupt, NMI) — обрабатываются всегда, независимо от запретов на другие прерывания*

- внутренние (синхронные) - события в самом процессоре, например, деление на ноль;
- программные - вызванные командой INT.

### Таблица векторов прерываний в реальном режиме работы процессора

- Вектор прерывания — номер, который идентифицирует соответствующий **обработчик прерываний.**
- Векторы прерываний объединяются в таблицу векторов прерываний, содержащую адреса обработчиков прерываний.
- Располагается в самом начале памяти, начиная с адреса 0.
- Доступно 256 прерываний.
- Каждый вектор занимает 4 байта - полный адрес.
- Размер всей таблицы - 1 Кб.

### Срабатывание прерывания

- Сохранение в текущий стек регистра флагов и полного адреса возврата (адреса следующей команды) - 6 байт

- Передача управления по адресу обработчика из таблицы векторов
- Настройка стека *(возможно, обработчику прерываний нужен свой стек, потому что стек остается связан с той программой, которая работала до срабатывания прерывания; если обработчик сложный, то иногда такие обработчики перенастраивают стек)*
- Повторная входимость (реентерабельность), необходимость запрета прерываний? *(Кузнецов: "таймер тикает, срабатывают прерывания. В какой-то момент прерывание тика таймера не успевает отработать до след тика, вызывается еще раз тоже прерывание и нужно обеспечить корректную работу в такой ситуации"; запрет прерывания можно делать только на короткий срок, иначе можно потерять данные (переполнение буфера клавиатуры, например))*
  ---

**Перехват прерывания**

- Сохранение адреса старого обработчика
- Изменение вектора на "свой" адрес
- Вызов старого обработчика до/после отработки своего кода
- При деактивации - восстановление адреса старого обработчика

**IRET - возврат из прерывания**

- Используется для выхода из обработчика прерывания
- Восстанавливает FLAGS, CS:IP
- При необходимости выставить значение флага обработчик меняет его значение непосредственно в стеке

**int<номер>** - вызов (генерация прерывания)

21h - прерывание DOS, предоставляет прикладным программам около 70 различных функций (ввод, вывод, работа с файлами, завершение программы и т.д.)

- Аналог системного вызова в современных ОС
- Используется наподобие вызова подпрограммы
- Номер функции прерыванию 21h передаётся через регистр AH. Параметры для каждой функции передаются собственным способом, он описан в документации. Там же описан способ возврата результата из функции в программу.

Вывод

| Функция | Назначение | Вход | Выход |
|---|---|---|---|
| 02 | Вывод символа в stdout | DL = ASCII-код символа | - |
| 09 | Вывод строки в stdout | DS:DX - адрес строки, заканчивающейся символом $ | - |

Ввод

| Функция | Назначение | Вход | Выход |
|---|---|---|---|
| 01 | Считать символ из stdin с эхом | - | AL - ASCII-код символа |
| 06 | Считать символ без эха, без ожидания, без проверки на Ctrl+Break | DL = FF | AL - ASCII-код символа |
| 07 | Считать символ без эха, с ожиданием и без проверки на Ctrl+Break | - | AL - ASCII-код символа |
| 08 | 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 |
| 0Ch | Очистить буфер и считать символ | AL=01, 06, 07, 08, 0Ah | |

## Установка обработчика прерывания в DOS

- int 21h
  - AH=35h, AL= номер прерывания - возвращает в ES:BX адрес обработчика (в BX 0000:[AL*4], а в ES - 0000:[AL*4+2]. )
  - AH=25h, AL=номер прерывания, DS:DX - адрес обработчика

Некоторые прерывания

- 0 - деление на 0
- 1 - прерывание отладчика, вызывается после каждой команды при флаге TF
- 3 - "отладочное", int 3 занимает 1 байт
- 4 - переполнение при команде INTO (команда проверки переполнения)
- 5 - при невыполнении условия в команде BOUND (команда контроля индексов массива)
- 6 - недопустимая (несуществующая) инструкция
- 7 - отсутствует FPU
- 8 - таймер
- 9 - клавиатура
- 10h - прерывание BIOS

## Резидентные программы

- Резидентная программа - та, которая остаётся в памяти после возврата управления DOS
- Завершение через функцию 31h прерывания 21h / прерывание 27h
- DOS не является многозадачной операционной системой
- Резиденты - частичная реализация многозадачности
- Резидентная программа должна быть составлена так, чтобы минимизировать используемую память

## Завершение с сохранением в памяти

- int 27h
  - DX = адрес первого байта за резидентным участком программы (смещение от PSP)
- int 21h, ah=31h
  - AL - код завершения
  - DX - объём памяти, оставляемой резидентной, в параграфах

## Порты ввода-вывода

- Порты ввода-вывода - механизм взаимодействия программы, выполняемой процессором, с устройствами компьютера.
- IN - команда чтения данных из порта ввода

- OUT - команда записи в порт вывода

Пример:

IN al, 61h

OR al, 3

OUT 61h, al

*IN приемник, источник – считать данные из порта с номером источник (непосредственный операнд или DX) в приемник (AL, AX)*
*OUT приемник, источник – записать данные из источника (AL, AX) в порт номер приемник (число до 255 или DX)*

Прерывания в x86 относятся к исключениям-ловушкам – обнаруживается и обслуживается после выполнения инструкции, его вызывающей. После обслуживания этого исключения управление возвращается на инструкцию, следующую за вызвавшей ловушку.

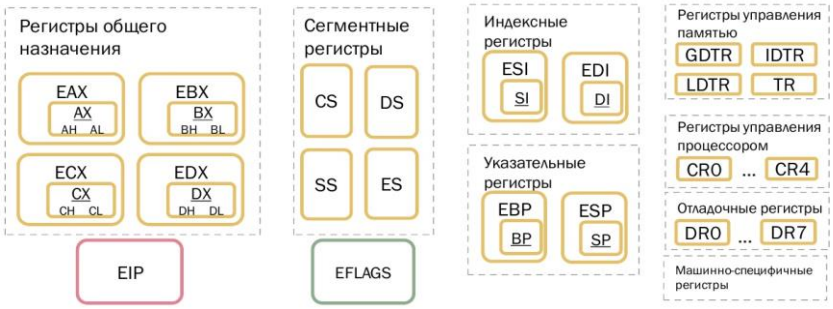
## 23. Процессор 80386. Режимы работы. Регистры.

С 386+ процессоры 32 разрядные. Производство x86 – 1985-2010. Сс 80286 – есть защищенный режим

32-разрядные:

- регистры, кроме сегментных
- шина данных
- шина адреса (2^32 = 4ГБ ОЗУ, до этого 2^20байт = 1Мб памяти)

### Регистры




Добавлены регистры поддержки работы в защищенном режиме (обеспечение разделения доступа программ между собой, между программами и ОС и тд; эти регистры справа на картинке)

### 1. Прежние расширенные

- EDX = Extended DX (обращение к частям остается (DX, DL))
- EIP
- EFLAGS - FLAGS + 5 специфических флагов
- Индексные ESI, EDI

- Указательные EBP, ESP

2. **Сегментные – те же, 16 разрядов**
3. **Регистры управления памятью**

- GDTR: (Global Descriptor Table Register) 6-байтный регистр, содержит 32-битный линейный адрес начала таблицы глобальных дескрипторов (GDT) и 16-битный размер (лимит, уменьшенный на 1). Может быть только 1 GDT
- IDTR: (Interrupt Descriptor Table Descriptor; то есть в защищенном режиме таблица векторов прерываний начинается с некоторого произвольного адреса) 6-байтный регистр, содержит 32-битный линейный адрес начала таблицы глобальных дескрипторов обработчиков прерываний (IDT) и 16-битный размер (лимит, уменьшенный на 1);
- LDTR: (Local Descriptor Table Register – для разделения памяти между программами) 10-байтный регистр, содержит 16-битный селектор для GDT (то есть какой именно базовый адрес брать) и весь 8-байтный дескриптор из GDT, описывающий текущую таблицу локальных дескрипторов;
- TR: (Task Register) 10-байтный регистр, содержит 16-битный селектор для GDT и весь 8-байтный дескриптор из GDT, описывающий TSS текущей задачи (Сегмент состояния задачи TSS (Task State Segment) - это структура данных, которая определяет состояние (т.е. контекст) задачи. В ней хранится содержимое всех регистров общего назначения, сегментных и некоторых системных регистров а также некоторая дополнительная информация.)

### *4.* Регистры управления процессором

- CR0 - флаги управления системой
  - PE - включение защищённого режима (из режима реальной адресации)
  - PG - включение режима страничной адресации. может использоваться только в защищенном режиме работы процессора (CR0.PE = 1).
  - управление отдельными параметрами кеша
  - WP - запрет записи в страницы "только для чтения"
  - NE - ошибки FPU вызывают исключение (внутренний механизм, обеспечивающий генерацию ошибки сопроцессора #MF), а не IRQ13 (в так называемом стиле MS-DOS, который предполагает обработку ошибок сопроцессора через вызов внешнего прерывания)
  - TS - устанавливается процессором после переключения задачи (для исключения исполнения команд над данными, перекочевавшими из другой задачи.)
- CR1 - зарезервирован
- CR2 - регистр адреса ошибки страницы - содержит линейный адрес страницы, при обращении к которой произошло исключение #PF
- CR3 - регистр основной таблицы страниц
  - Иногда CR3 называют регистром базы каталога страниц (PDBR).
  - 20 старших бит физического адреса начала каталога таблиц (при страничном механизме)
  - либо 27 старших бит физического адреса начала таблицы указателей на каталоги страниц (при поддержке расширения физического адреса), в зависимости от бита PAE в CR4
  - младшие биты - 0
  - Управление кешированием и сквозной записью страниц
- CR4 - регистр управления новыми возможностями процессоров (с Pentium)

  Поэтому перед программированием любых битов этого регистра, необходимо с помощью команды CPUID проверить наличие поддержки требуемого режима в конкретной модификации процессора. В случае, если какой-либо из режимов не поддерживается, то соответствующий бит регистра CR4 считается зарезервированным и изменение его значения недопустимо

5. **Отладочные регистры**

- DR0..DR3 - 32-битные линейные адреса четырёх возможных точек останова по доступу к памяти
- DR4, DR5 - зарезервированы
- DR6 (DSR) - регистр состояния отладки. Содержит причину останова
- DR7 (DCR) - регистр управления отладкой. Управляет четырьмя точками останова

6. **Машинно-специфичные регистры**

- Управление кешем
- Дополнительное управление страничной адресацией
- Регистры расширений процессора: MMX и т.д.

## Режимы работы

### 1. Реальный режим

**"Реальный" режим** (режим совместимости с 8086)

- обращение к оперативной памяти происходит по реальным (действительным) адресам, трансляция адресов не используется;
- набор доступных операций не ограничен;
- защита памяти не используется.

*Единственным способом выхода из реального режима является явное переключение в защищенный режим, которое производится установкой специального флага в одном из системных регистров*

### 2. Защищенный режим (с 80286)

- В защищенном режиме обращение к памяти происходит по виртуальным адресам с использованием механизмов защиты памяти.
- Набор доступных операций определяется уровнем привилегий (кольца защиты): системный и пользовательский уровни.
- Колец защит всего 4. Чем ниже уровень защиты, тем больше возможностей. 0 уровень - всё доступно.

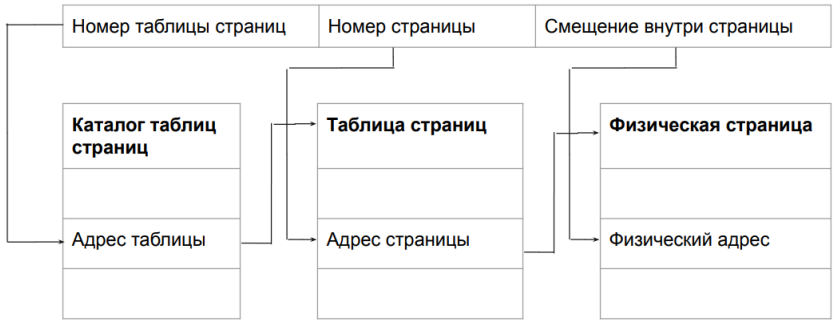
### 3. Режим V86 –

Прикладные программы для 8086 могут исполняться на 32-разрядных процессорах, как в реальном режиме, так и в режиме виртуального 8086 (V86), который является особым состоянием задачи защищенного режима.

## Модели памяти в защищенном режиме

- Плоская - код и данные используют одно и то же пространство.

- Сегментная - сложение сегмента и смещения (используется также в реальном режиме; знакома нам)
- Страничная - виртуальные адреса отображаются на физические постранично
    - виртуальная память - метод управления памятью компьютера, позволяющий выполнять программы, требующие больше оперативной памяти, чем имеется в компьютере, путём автоматического перемещения частей программы между основной памятью и вторичным хранилищем (файл, или раздел подкачки);
    - основной режим для большинства современных ОС;
    - **в x86 минимальный размер страницы - 4096 байт;**
    - основывается на таблице страниц - структуре данных, используемой системой виртуальной памяти в операционной системе компьютера для хранения сопоставления между виртуальным адресом и физическим адресом. Виртуальные адреса используются выполняющимся процессом (программа имеет информацию только о виртуальных адресах), в то время как физические адреса используются аппаратным обеспечением. Таблица страниц является ключевым компонентом преобразования виртуальных адресов, который необходим для доступа к данным в памяти.

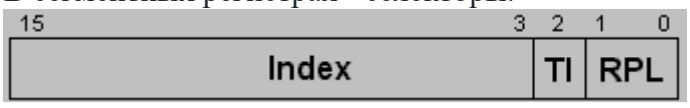



- Программы полностью изолированы друг от друга
- В память можно загрузить больше программ, чем памяти доступно (долго не использующиеся данные загружаются на диск и освобождают место)

## Управление памятью в x86

Сегментные регистры меняют назначение: они внешне выглядят 2 байтными, но на деле они 8 байтные, просто 6 байт - теневые регистры, используются процессором для кеширования дескрипторов страниц

- В сегментных регистрах - селекторы:




    - 13-разрядный номер дескриптора;
    - какую таблицу использовать - глобальную или локальную (таблица текущей программы/задачи);
    - уровень привилегий запроса 0-3
        - 0 - система
        - 3 - прикладная программа
        - 1-2 - где-то не используется, где-то используется, например, для драйверов
- По селектору определяется запись в одной из таблиц дескрипторов сегментов;
- При включённом страничном режиме - по таблице страниц определяется физический адрес страницы либо выявляется, что она выгружена из памяти, срабатывает исключение и операционная система подгружает затребованную страницу из "подкачки" (swap).

*При адресации памяти в защищённом режиме команды ссылаются на сегменты, указывая не их адреса (как в режиме реальных адресов), а описания сегментов (их дескрипторы). Указатель на описание сегмента называется селектор. Другими словами, селектор - это номер дескриптора из таблицы дескрипторов.*

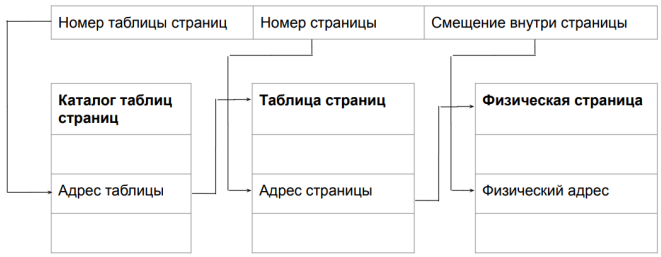
*Адресация производится через пару регистров сегмент:смещение, причём, в качестве сегментного регистра используются обычные CS, SS, DS, ES, FS и GS (последние два появились в 386-м процессоре), но в них указывается не адрес сегмента, а селектор дескриптора.*

*Обращение к дескрипторной таблице процессор производит только в момент загрузки в сегментный регистр нового селектора. После этого содержимое дескриптора копируется в так называемый "теневой регистр", к которому имеет доступ только сам процессор и из которого оно в дальнейшем используется. Любое последующее обращение к сегменту будет происходить с помощью теневого регистра, без обращения к дескрипторной таблице и не потребует лишних тактов на циклы чтения памяти.*

*При загрузке недопустимого значения селектора процессор будет генерировать исключение, даже если вы не обращались через него к памяти.*

## Страничное преобразование

- преобразование линейного адреса в физический




- Линейный адрес - 32:
  - биты 31-22 (10) - номер таблицы страниц в каталоге;
  - биты 21-12 (10) - номер страницы в выбранной таблице;
  - биты 11-0 (12) - смещение от физического адреса начала страницы в памяти.
- Каждое обращение к памяти требует двух дополнительных обращений (проблема, долго);
- Необходим специальный кеш страниц - TLB (решение проблемы выше; внутри процессора);
- Каталог таблиц/таблица страниц:
  - биты 31-12 - биты 31-12 физического адреса таблицы страниц/страницы;
  - младшие биты - атрибуты управления страницей (элементы - таблицы страниц отдельных программ/страницы программы).

## Механизм защиты

Механизм защиты - ограничение доступа к сегментам или страницам в зависимости от уровня привилегий

- К типам сегментов реального режима (код, стек, данные) добавляется TSS (Task State Segment) - сегмент состояния задачи. В нём сохраняется вся информация о задаче на время приостановки выполнения. **Размер - 68h (104) байт.**

- Структура:

- селектор предыдущей задачи
- Регистры стека 0, 1, 2 уровней привилегий
- EIP, EFLAGS, EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI, CS, DS, ES, FS, HS, SS, LDTR
- флаги задачи
- битовая карта ввода-вывода (контроль доступа программы к устройствам
- Используется ОС для диспетчеризации задач, в т. ч. переключения на стек ядра при обработке прерываний и исключений.
- Дескриптор сегмента состояния задачи (task state segment descriptor) должен находиться в GDT
- При переключении задач регистры выполняющейся задачи автоматически сохраняются в её TSS, а регистры новой задачи автоматически подгружаются, но уже из её TSS.

*Система команд*

- *Аналогична системе команд 16-разрядных процессоров*
- *Доступны как прежние команды обработки 8- и 16-разрядных аргументов, так и 32-разрядных регистров и переменных*
- *Пример:*
- *mov eax, 12345678h*
- *xor ebx, ebx*
- *mov bx, 1*

*add eax, ebx ; eax=12345679h*

***Системные и привилегированные команды***

- *Выполнение ограничено, в основном, нулевым кольцом защиты*
- *LGDT, SGDT ззагрузить (сохранить) регистр глобальной таблицы дескрипторов (GDT)*
- *LLDT, SLDT - локальной*
- *LTR, STR - регистр задачи*
- *LIDT, SIDT - регистр таблицы дескрипторов прерываний (IDT)*
- *MOV CR0..CR4 или DR0..DR7,*
- *...*

*Исключения*

*Исключения (Exceptions) подразделяются на отказы, ловушки и аварийные завершения.*

1. *Отказ (fault) — до, подкачка, на ту же*
   *это исключение, которое обнаруживается и обслуживается до выполнения инструкции, вызывающей ошибку. После обслуживания этого исключения управление возвращается снова на ту же инструкцию (включая все префиксы), которая вызвала отказ. Отказы, использующиеся в системе виртуальной памяти, позволяют, например, подкачать с диска в оперативную память затребованную страницу или сегмент.*
2. *Ловушка (trap) — после, прерывания, на следующую*
   *это исключение, которое обнаруживается и обслуживается после выполнения инструкции, его вызывающей. После обслуживания этого исключения управление возвращается на инструкцию, следующую за вызвавшей ловушку. К классу ловушек относятся и программные прерывания.*
3. *Аварийное завершение (abort) — непонятно и серьезно*
   *это исключение, которое не позволяет точно установить инструкцию, его вызвавшую. Оно используется для сообщения о серьезной ошибке, такой как аппаратная ошибка или повреждение системных таблиц*

## 24. Математический сопроцессор. Типы данных.

### Математический сопроцессор

Сопроцессор (FPU – Floating Point Unit)

Изначально - отдельное опциональное устройство на материнской плате, с 80486DX встроен в процессор

### Типы данных.

Операции над 7-ю типами данных

1. целое слово (16 бит)
2. короткое целое (32 бита)
3. длинное слово (64 бита)
4. упакованное десятичное (80 бит)
   Packed BCD (Binary-coded decimal). Каждые 4 бита такого числа могут принимать значения от 0 до 9, бит #79 - знак, 78-72 не имеют значения. При выгрузке NaN'ов, получается число, 18 старших битов которого установлены, а остальные - сброшены.
   Пример: десятичное число -12345 в формате Packed BCD равно 80 00 00 00 00 00 00 01 23 45h. В памяти байты расположены наоборот.

5. короткое вещественное (32 бита)
6. длинное вещественное (64 бита)
7. расширенное вещественное (80 бит)

### Форма представления числа с плавающей запятой в FPU

- Нормализованная форма представления числа (1,...*2exp)
- Экспонента увеличена на константу для хранения в положительном виде
- Бит 31 - знак мантиссы, 30-23 - экспонента, увеличенная на 127, 22-0 - мантисса без первой цифры
- Пример представления 0,625 в коротком вещественном типе:
  1/2+1/8 = 0,101b
  1,01b*2-1
  **0**01111110**01000000000000000000000**
- Все вычисления FPU - в расширенном 80-битном формат

### особые числа:

- Положительная бесконечность: знаковый - 0, матнисса - нули, экспонента – единицы
- Отрицательная бесконечность: знаковый - 1, мантисаа - нули, экспонента – единицы
  возникают при делении бесконечности (или числа) на ноль.
- NaN (Not a Number):
  - qNAN (quiet) - при приведении типов/отдельных сравнениях. не приводит к исключению (арифметической ошибки нет, но получить число без округления нельзя)
  - sNAN (signal) - переполнение в большую/меньшую сторону, прочие ошибочные ситуации

- Денормализованные числа (экспонента = 0): находятся ближе к нуля, чем наименьшее представимое нормальное число

*Перевод в денормализованные числа может производится аппаратно при получении очень малых значений. Их обработка может производится дольше, чем обработка нормализованных чисел.*

***Константы FPU:***

- *FLD1 – 1,0*
- *FLDZ - +0,0*
- *FLDPI - число Пи*
- *FLDL2E - log2e*
- *FLDL2T - log210*
- *FLDLN2 – ln(2)*
- *FLDLG2 – lg(2)*

## 25. Математический сопроцессор. Регистры.

### Математический сопроцессор

Сопроцессор (FPU – Floating Point Unit)

Изначально - отдельное опциональное устройство на материнской плате, с 80486DX встроен в процессор

### Регистры

- В сопроцессоре доступно 8 80-разрядных регистров (R0..R7) адресуются не по именам, а рассматриваются в качестве стека ST. ST соответствует регистру - текущей вершине стека ST(0); ST(1)..ST(7) - прочие регистры
- SR - регистр состояний, содержит слово состояния FPU. Сигнализирует о различных ошибках, переполнениях. *Отдельные биты описывают и состояния регистров и в целом сигнализируют об ошибках (переполнениях и тп) при последней операции.*
- CR - регистр управления. Контроль округления, точности (тоже 16 разрядный). *Через него можно настраивать правила округления чисел и контроль точности (с помощью специальных битов устанавливать параметры, гибкие настройки)*
- TW - 8 пар битов, описывающих состояния регистров: число (00), ноль (01), не-число (10), пусто (11) (изначально все пустые, проинициализированы единичками)
- FIP, FDP - адрес последней выполненной команды и её операнда для обработки исключений

## 26. Математический сопроцессор. Классификация команд.

### Математический сопроцессор

Сопроцессор (FPU – Floating Point Unit)

Изначально - отдельное опциональное устройство на материнской плате, с 80486DX встроен в процессор

### Классификация команд

Все команды сопроцессора оперируют регистрами стека сопроцессора. Если операнд в команде не указывается, то по умолчанию используется вершина стека сопроцессора (логический регистр st(0)). Если команда выполняет действие с двумя операндами по умолчанию, то эти операнды – регистры st(0) и st(1).

### 1. Команды пересылки данных:

команды взаимодействия со стеком (загрузка - выгрузка вещественного числа, целого, BCD (упакованного); смена мест регистров)

- *FLD - загрузить вещественное число из источника (переменная или ST(n)) в стек. Номер вершины в SR увеличивается*
- *FST/FSTP - скопировать/считать число с вершины стека в приёмик*
- *FILD - преобразовать целое число из источника в вещественное и загрузить в стек*
- *FIST/FISTP - преобразовать вершину в целое и скопировать/считать в приёмник*
- *FBLD, FBSTP - загрузить/считать десятичное BCD-число*
- *FXCH - обменять местами два регистра (вершину и источник) стека*

### 2. Арифметические команды:

- *FADD, FADDP, FIADD - сложение, сложение с выталкиванием из стека, сложение целых. Один из операндов - вершина стека*
- *FSUB, FSUBP, FISUB - вычитание*
- *FSUBR, FSUBRP, FISUBR - обратное вычитание (приёмника из источника)*
- *FMUL, FMULP, FIMUL - умножение*
- *FDIV, FDIVP, FIDIV - деление*
- *FDIVR, FDIVRP, FIDIVR - обратное деление (источника на приёмник)*
- *FPREM - найти частичный остаток от деления (делится ST(0) на ST(1)). Остаток ищется цепочкой вычитаний, до 64 раз*
- *FABS - взять модуль числа*
- *FCHS - изменить знак*
- *FRNDINT - округлить до целого*
- *FSCALE - масштабировать по степеням двойки (ST(0) умножается на 2^ST(1))*
- *FXTRACT - извлечь мантиссу и экспоненту. ST(0) разделяется на мантиссу и экспоненту, мантисса дописывается на вершину стека, экспонента - на прошлом месте*
- *FSQRT - вычисляет квадратный корень ST(0*

### 3. Команды сравнений:

Команды сравнения сравнивают значение в вершине стека с операндом. По умолчанию (если операнд не задан) происходит сравнение регистров ST(0) и ST(1). В качестве операнда может быть задана ячейка памяти или регистр. Команда устанавливает флаги (основные и в регистре SR), биты C0, C2, C3 регистра swr в соответствии с таблицей. Сбрасывает в 0 признак C1 при пустом стеке после выполнения команды.

- *COM, FCOMP, FCOMPP - сравнить и вытолкнуть из стека*
- *FUCOM, FUCOMP, FUCOMPP - сравнить без учёта порядков и вытолкнуть*
- *FICOM, FICOMP, FICOMP - сравнить целые*
- *FCOMI, FCOMIP, FUCOMI, FUCOMIP (P6)*
  *устанавливает биты ZF, PF, CF регистра EFLAGS в соответствии с таблицей.*
- *FTST - сравнивает с нулём*
- *FXAM - выставляет флаги в соответствии с типом числа*

### 4. Трансцедентные операции

- *FSIN*
- *FCOS*
- *FSINCOS*
- *FPTAN*

- *FPATAN*
- *F2XM1 – 2^x -1*
- *FYL2X, FYL2XP1 – y*log2 x, y*log2 (x+1)*

*sin (fsin), cos(fsin) - принимают значение в радианах в некотором диапазоне. tg, arctg, 2^x - 1, y * log2x...*

### 5. Константы FPU:

- *FLD1 – 1,0*
- *FLDZ - +0,0*
- *FLDPI - число Пи*
- *FLDL2E - log2e*
- *FLDL2T - log210*
- *FLDLN2 – ln(2)*
- *FLDLG2 – lg(2)*

### 6. Команды управления:

- *FINCSTP, FDECSTP - увеличить/уменьшить указатель вершины стека*
- *FFREE - освободить регистр*
- *FINIT, FNINIT - инициализировать сопроцессор / инициализировать без ожидания (очистка данных, инициализация CR и SR по умолчанию)*
- *FCLEX, FNCLEX - обнулить флаги исключений / обнулить без ожидания*
- *FSTCW, FNSTCW - сохранить CR в переменную / сохранить без ожидания*
- *FLDCW - загрузить CR*
- *FSTENV, FNSTENV – сохранить вспомогательные регистры (14/28 байт) / сохранить без ожидания*
- *FLDENV - загрузить вспомогательные регистры*
- *FSAVE, FNSAVE, FXSAVE - сохранить состояние (94/108 байт) и инициализировать, аналогично FINIT*
- *FRSTOR, FXRSTOR - восстановить состояние FPU*
- *FSTSW, FNSTSW - сохранение CR*
- *WAIT, FWAIT - обработка исключений*
- *FNOP - отсутствие операции*

### Команда CPUID (с 80486)

Идентификация процессора CPU, предназначена для считывания программным обеспечением информации о продавце, семействе, модели и поколении процессора, а также специфической для процессора дополнительной информации (поддерживаемые наборы команд, размеры буферов, кэшей, разнообразные расширения архитектуры и т.п.)

Перед выполнением команды CPUID в регистр EAX должно помещаться входное значение, которое и указывает — какую информацию необходимо выдать.

- *Если EAX = 0, то в EAX - максимальное допустимое значение (1 или 2), а EBX:ECX:EDX – 12-байтный идентификатор производителя (ASCII-строка).*
- *Если EAX = 1, то в EAX - версия, в EDX - информация о расширениях –*
  - *EAX - модификация, модель, семейство*
  - *EDX: наличие FPU, поддержка V86, поддержка точек останова, CR4, PAE, APIC, быстрые системные вызовы, PGE, машинно-специфичный регистр, CMOVcc**, MMX, FXSR (MMX2), SSE***
- *Если EAX = 2, то в EAX, EBX, ECX, EDX возвращается информация о кэшах и TLB*

### *Исключения FPU*

- *Неточный результат - произошло округление по правилам, заданным в CR. Бит в SR хранит направление округления*
- *Антипереполнение - переход в денормализованное число*
- *Переполнение - переход в "бесконечность" соответствующего знака*
- *Деление на ноль - переход в "бесконечность" соответствующего знака*
- *Денормализованный операнд*
- *Недействительная операция*

## 27. Расширения процессора. MMX. Регистры, поддерживаемые типы данных.

### (1997, Pentium MMX)

Расширение, которое было встроено для увеличения эффективности обработки больших потоков данных (изображение, звук, видео..) - простые операции над массивами однотипных чисел

### Регистры

- 8 64-битных регистров MM0..MM7 - **мантиссы регистров FPU (то есть целые)**. При записи в MMn экспонента и знаковый бит заполняются единицами (отрицательная бесконечность получается)
- Пользоваться одновременно и FPU, и MMX не получится, требуется FSAVE+FRSTOR

### Типы данных MMX:

- учетверённое слово (64 бита);
- упакованные двойные слова (2);
- упакованные слова (4);
- упакованные байты (8).

- Команды MMX перемещают упакованные данные в память или обычные регистры целиком, но арифметические и логические операции выполняют поэлементно.
- насыщение - замена переполнения/антипереполнения превращением в максимальное/минимальное значение (Светлый цвет + светлый цвет максимум равно белый (но не будет переполнения и темного цвета))

## 28. Расширения процессора. MMX. Классификация команд.

### (1997, Pentium MMX)

Расширение, которое было встроено для увеличения эффективности обработки больших потоков данных (изображение, звук, видео..) - простые операции над массивами

- Команды MMX перемещают упакованные данные в память или обычные регистры целиком, но арифметические и логические операции выполняют поэлементно.
- насыщение - замена переполнения/антипереполнения превращением в максимальное/минимальное значение (Светлый цвет + светлый цвет максимум равно белый (но не будет переполнения и темного цвета))

### Классификация команд

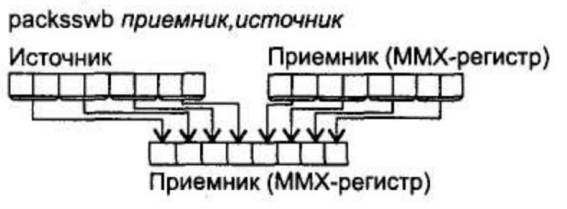
В режиме с насыщением, результаты операции, которые переполняются сверху или снизу отсекаются к границе datarange соответствующего типа данных

В режиме без насыщения, результаты, которые переполняются как в обычной процессорной арифметике

| Тип данных | Нижний предел | | Верхний предел | |
|---|---|---|---|---|
| | Шестн адцат. | Десяти чн. | Шестн адцат. | Десати чн. |
| Знаковый байт | 80H | -128 | 7FH | 127 |
| Знаковое слово | 8000H | -32768 | 7FFFH | 32767 |
| Беззнаковый байт | 00H | 0 | FFH | 255 |
| Беззнаковое слово | 0000H | 0 | FFFFH | 65535 |

### 1. Команды пересылки данных:

- MOVD, MOVQ - пересылка двойных/учетверённых слов
- PACKSSWB, PACKSSDW - упаковка со знаковым насыщением слов в байты/двойных слов в слова. Приёмник -> младшая половина приёмника, источник -> старшая половина приёмника




*Если значение слова больше или меньше границ диапазона знакового байта, то результат упаковки насыщается соответственно до 7Fh или до 80h. (до 7FFFh или до 8000h.)*

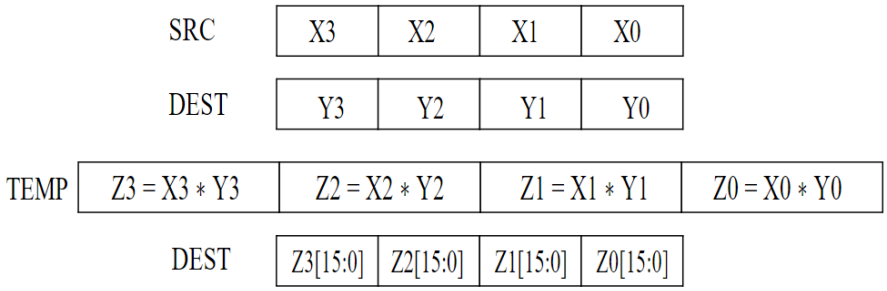
- PACKUSWB - упаковка слов в байты с беззнаковым насыщением.

*Если значение слова больше или меньше границ диапазона беззнакового байта, то результат упаковки насыщается соответственно до FFh или до 00h.*

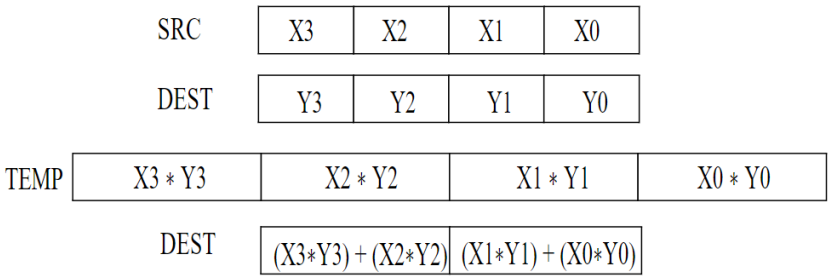
- PUNPCKH**BW**, PUNPCKH**WD**, PUNPCKH**DQ** - распаковка и объединение старших элементов источника и приёмника через 1
*Команда PUNPCKH распаковывает старшие элементы операнда-источника и операнда-назначения в операнд-назначение. Элементы двух операндов записываются в результат через один, т.е. в старший элемент результата помещается старший элемент операнда-источника, в следующий более младший элемент — старший элемент операнда-назначения, далее — следующий элемент из операнда-источника, элемент из операнда-назначения и т.д. до полного заполнения всех элементов результата, этот результат затем помещается в операнд-назначение.*

### 2. Арифметические операции:

- PADDB, PADDW, PADDD - поэлементное сложение, перенос игнорируется
- PADDSB, PADDSW - сложение с насыщением
- PADDUSB, PADDUSW - беззнаковое сложение с насыщением
- PSUBB, PSUBW, PDUBD - вычитание, заём игнорируется
- PSUBSB, PSUBSW - вычитание с насыщением
- PSUBUSB, PSUBUSW - беззнаковое вычитание с насыщением
- PMILHW, PMULLW - старшее/младшее умножение (сохраняет старшую или младшую части результата в приёмник)
(low)




- PMADDWD - умножение и сложение. Перемножает 4 слова, затем попарно складывает произведения двух старших и двух младших




### 3. Команды сравнения:

- PCMPEQB, PCMPEQW, PCMPEQD - проверка на равенство. Если пара равна - соответствующий элемент приёмника заполняется единицами, иначе – нулями
- PCMPGTB, PCMPGTW, PCMPGTD - сравнение. Если элемент приёмника больше, то заполняется единицами, иначе - нулями

### 4. Логические операции:

- PAND - логическое И
- PANDN - логическое НЕ-И (штрих Шеффера) (источник*НЕ(приёмник))
- POR - логическое ИЛИ
- PXOR - исключающее ИЛИ

### 5. Сдвиговые операции:

- PSLLW, PSLLD, PSLLQ - логический влево
- PSRLW, PSRLD, PSRLQ - логический вправо
- PSRAW, PSRAD - арифметический вправо

Арифметический сдвиг отличается от логического тем, что он не изменяет значение старшего бита, и предназначен для чисел со знаком.

29. Расширения процессора. SSE. Регистры, поддерживаемые типы данных.
30. Расширения процессора. SSE. Классификация команд

### Расширение SSE (Pentium III, 1999)

Решение проблемы параллельной работы с FPU

### Регистры:

- 8 128-разрядных регистров
- свой регистр флагов

### Типы данных

- Основной тип - вещественные одинарной точности (32 бита, в 1 регистре 4 числа)

- Целочисленные команды работают с регистрами MMX

**Команды**

- Пересылки
- Арифметические
- Сравнения
- Логические
- Преобразования типов
- Целочисленные
- Упаковки
- Управления состоянием
- Управления кэшированием

*Развитие: SSE2, SSE3..*

*Расширение AES*

*(Intel Advanced Encryption Standard New Instructions; AES-NI, 2008)*

*Цель - ускорение шифрования по алгоритму AES*

*Команды:*

- *раунда шифрования;*
- *раунда расшифровывания;*
- *способствования генерации ключа*

Не вошедшее

**Немного о памяти**

- Байт - минимальная адресуемая единица памяти (8 бит, 0…255=256 значений, 256 = 2^8 =$100_{16}$
- ).
- Машинное слово - машинно-зависимая величина, измеряемая в битах, равная разрядности регистров/шины данных.
- **Параграф - 16 байт**
- **Сегмент – 2^16 байт (64 Кбайт)** максимум, обычно - меньше
- ASCII (аски́) - American standard code for information interchange, США, 1963.
  - 7-битная кодировка (в расширенном варианте - 8-битная)
  - первые 32 символа - непечатные (служебные)
  - старшие 128 символов 8-битной кодировки - национальные языки, псевдографика и т. п.
- Системы счисления:
  - Двоичная (binary)  Суффикс - b. Пример: 1101b)
  - Шестнадцатеричная (hexadecimal) Суффикс - h (10h = 16). Некоторые компиляторы требуют префикса 0x (0x10))
- Представление отрицательных чисел
  - Знак - в старшем разряде (0 - “+”, 1 - “-”).
    Возможные способы:
  - прямой код

- обратный код (инверсия)
- **дополнительный код (инверсия и прибавление единицы)**

Примеры доп. кода на 8-разрядной сетке
-1:
1. 00000001
2. 11111110
3. 11111111

Смысл: -1 + 1 = 0 (хоть и с переполнением):
11111111 + 1 = (1)00000000

-101101:
1. 00101101
2. 11010010
3. 11010011

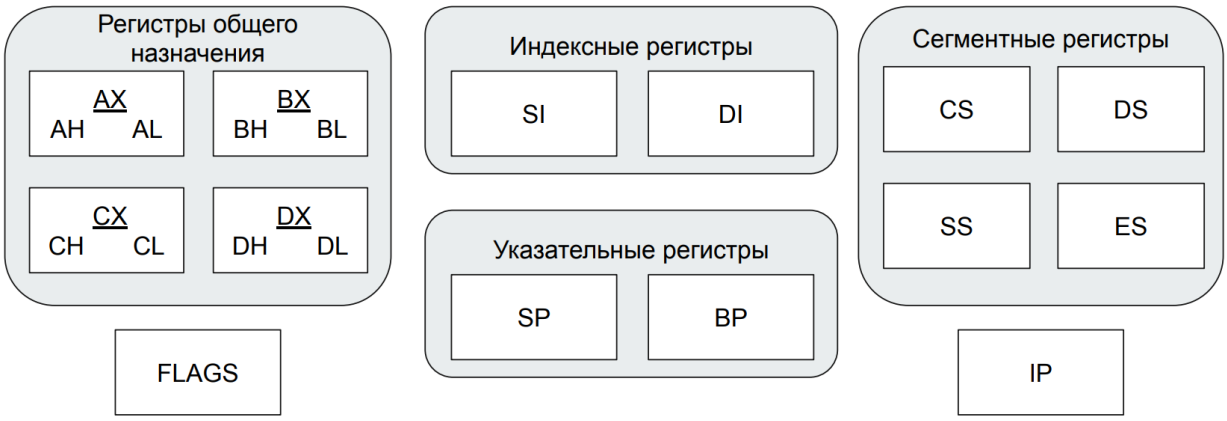
## Виды современных архитектур ЭВМ

- 8086 (16-разр.) 80386 (32-разр.) x86-64 (64-разр.)
- ARM
- IA64
- MIPS (включая Байкал)
- Эльбрус

## Семейство процессоров x86 и x86-64

- Микропроцессор 8086: 16-разрядный, 1978 г., 5-10 МГц, 3000 нм
- Предшественники: 4004 - 4-битный, 1971 г.; 8008 - 8-битный, 1972 г.; 8080 - 1974 г.
- Требует микросхем поддержки
- 80186 - 1982 г., добавлено несколько команд, интегрированы микросхемы поддержки
- 80286 - 1982 г., 16-разрядный, добавлен защищённый режим
- 80386, 80486, Pentium, Celeron, AMD, … - 32-разрядные, повышение быстродействия и расширение системы команд
- x86-64 (x64) - семейство с 64-разрядной архитектурой
- Отечественный аналог - К1810ВМ86, 1985 г.
- **8086-20 разрядов шины**

## Структура блока регистров




1. **IP (instruction pointer)**- регистр командного указателя
   - Смещение команды, которая должна быть выполнена следующей
   - Программно изменить нельзя, но можно узнать значение (отладка)
2. **Индексные регистры SI, DI**

- SI - source index (индекс источника)
- DI - destination index (индекс приёмника)
- Могут использоваться в большинстве команд, как регистры общего назначения
- Применяются в специфических командах поточной обработки данных

*(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*

Команда NOP (no operation)

- Ничего не делает
- Занимает место и время
- Размер - 1 байт, код - 90h
- Назначение - задержка выполнения либо заполнение памяти, например, для выравнивания

Загрузка сегментных регистров

- LDS <приемник>, <источник> - загрузить адрес, используя DS
- LES <приемник>, <источник> - загрузить адрес, используя ES
- LFS <приемник>, <источник> - загрузить адрес, используя FS
- LGS <приемник>, <источник> - загрузить адрес, используя GS
- LSS <приемник>, <источник> - загрузить адрес, используя SS

Приёмник - регистр, источник – переменна

Метки

**Метки**

В коде

- Пример:

```
    mov cx, 5
label1:
    add ax, bx
    loop label1
```

- Метки обычно используются в командах передачи управления

В данных

- label
  - *метка* label *тип*
  - Возможные типы: BYTE, WORD, DWORD, FWORD, QWORD, TBYTE, NEAR, FAR.
- EQU, =
  - *label* EQU *выражение*
  - макрос
  - вычисляет выражение в правой части и приравнивает его метке

BYTE(1), WORD(2), DWORD(4), FWORD(6), QWORD(8), TBYTE(10)

Если метка располагается перед командой процессора, сразу после нее всегда ставится оператор **:** (двоеточие), который указывает ассемблеру, что надо создать переменную с этим именем, содержащую адрес текущей команды:

```
some_loop:
        lodsw                   ; Считать слово из строки.
        cmp     ax,7            ; Если это 7 - выйти из цикла.
        loopne  some_loop
```

Когда метка стоит перед директивой ассемблера, она обычно оказывается одним из операндов этой директивы и двоеточие не ставится. Рассмотрим директивы, работающие напрямую с метками и их значениями, - LABEL, EQU и =.

```
метка   label   тип
```

Директива LABEL определяет метку и задает ее тип: BYTE (байт), WORD (слово), DWORD (двойное слово), FWORD (6 байт), QWORD (учетверенное слово), TBYTE (10 байт), NEAR (ближняя метка), FAR (дальняя метка). Метка получает значение, равное адресу следующей команды или следующих данных, и тип, указанный явно. В зависимости от типа команда

```
        mov     метка,0
```

запишет в память байт (слово, двойное слово и т. д.), заполненный нулями, а команда

```
        call    метка
```

выполнит ближний или дальний вызов подпрограммы.

С помощью директивы LABEL удобно организовывать доступ к одним и тем же данным, как к байтам, так и к словам, определив перед данными две метки с разными типами.

```
метка   equ     выражение
```

Директива EQU присваивает метке значение, которое определяется как результат целочисленного выражения в правой части. Результатом этого выражения может быть целое число, адрес или любая строка символов:

```
truth           equ     1
message1        equ     'Try again$'
var2            equ     4[si]
        cmp     ax, truth       ; cmp ax,1
        db      message1        ; db 'Try again$'
        mov     ax,var2         ; mov ax, 4[si]
```

Директива EQU чаще всего используется с целью введения параметров, общих для всей программы, аналогично команде #define препроцессора языка С.

Директива = эквивалентна EQU, но определяемая ею метка может принимать только целочисленные значения. Кроме того, метка, указанная этой директивой, может быть переопределена.

## Директивы выделения памяти

- Директива - инструкция ассемблеру, влияющая на процесс компиляции и не являющаяся командой процессора. Обычно не оставляет следов в формируемом машинном коде.
- Псевдокоманда - директива ассемблера, которая приводит к включению данных или кода в программу, но не соответствующая никакой команде процессора.
- Псевдокоманды определения данных указывают, что в соответствующем месте располагается переменная, резервируют под неё место заданного типа, заполняют значением и ставят в соответствие метку.
- Виды: DB (1), DW (2), DD (4), DF (6), DQ (8), DT (10).
- Примеры:

    - a DB 1
    - float_number DD 3.5e7
    - text_string DB ‘Hello, world!’
- DUP - заполнение повторяющимися данными
- ? - неинициализированное значение
    - uninitialized DW 512 DUP(?)

## Прочие директивы

- Задание набора допустимых команд: .8086, .186, .286, …, .586, .686, …
- Управление программным счётчиком :
    - ORG значение – с этого момента в программе будет отступ на значение
      Org 100h – сразу пропустить 256 байт
    - EVEN – выравнивать все, что идет после по адресу, кратному 2
    - ALIGN значение – по кратному значению
- Глобальные объявления
    - Public – переменная будет доступна из других модулей
    - comm,
    - extrn – подключает эту метку
    - global
- Условное ассемблирование IF выражение … ELSE … ENDIF