**МЗЯП**

## 1. Архитектура фон Неймана, принципы фон Неймана.

Архитектура фон Неймана - система принципов построения и функционирования вычислительных машин, предложенная в 1945 году.

В основу архитектуры легли пять основополагающих принципов:

1. **Использование двоичной системы счисления в вычислительных машинах** - это позволило упростить создание устройств, арифметические и логические операции выполняются достаточно быстро.
2. **Программное управление ЭВМ** - работа ЭВМ контролируется программой, состоящей из набора команд, которые выполняются последовательно друг за другом. Программа хранится в памяти.
3. **Принцип однородности памяти: в памяти хранятся не только данные, но и программы.** И программы, и данные хранятся в двоичной системе счисления, то есть способ их записи одинаковый.
4. **Ячейки памяти ЭВМ имеют адреса, которые последовательно пронумерованы.** В любой момент к ячейке памяти можно обратиться по её адресу, что позволяет хранить переменные.
5. **Возможность условного перехода в процессе выполнения программы.** Хоть программы и выполняются последовательно, возможен условный переход к любому участку кода.

Эти принципы легли в основу машины фон Неймана

Машина фон Неймана состоит из запоминающего устройства (ЗУ), арифметико-логического устройства (АЛУ), устройства управления (УУ) и устройств ввода-вывода. Схема:



(не обязательно)Команда состоит из указания, какую операцию следует выполнить, и адресов ячеек памяти, где хранятся данные, над которыми следует выполнить указанную операцию, а также адреса ячейки, куда следует записать результат (если его требуется сохранить в ЗУ). То есть команда состоит из комбинации: операция + адрес начальных данных + адрес, куда записывать результат. Разрядность операндов = длине машинного слова (величина, равная разрядности регистров/шины данных).

УУ управляет всеми частями компьютера. От управляющего устройства на другие устройства поступают сигналы «что делать», а от других устройств УУ получает информацию об их состоянии. Также УУ имеет счетчик команд, который увеличивается при выполнении команды. Является конечным дискретным автоматом.

УУ и АЛУ входят в состав процессора.

АЛУ выполняет бинарные арифметические и логические операции, и отправляет результат в соответствии с командой либо в память, либо в устройство вывода. Отличие состоит в том, что выбирается оптимальный вид, как передается результат.

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

**Машинная команда** - инструкция из аппаратно определённого набора, которую способен выполнять процессор. Машинные команды можно разделить на несколько групп:

1. Команды передачи данных - копируют данные из одной ячейки памяти в другую.
2. Арифметические и логические операции.
3. Команды переходов:условная и безусловная передача управления.
4. Команды работы с подпрограммами.
5. 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.

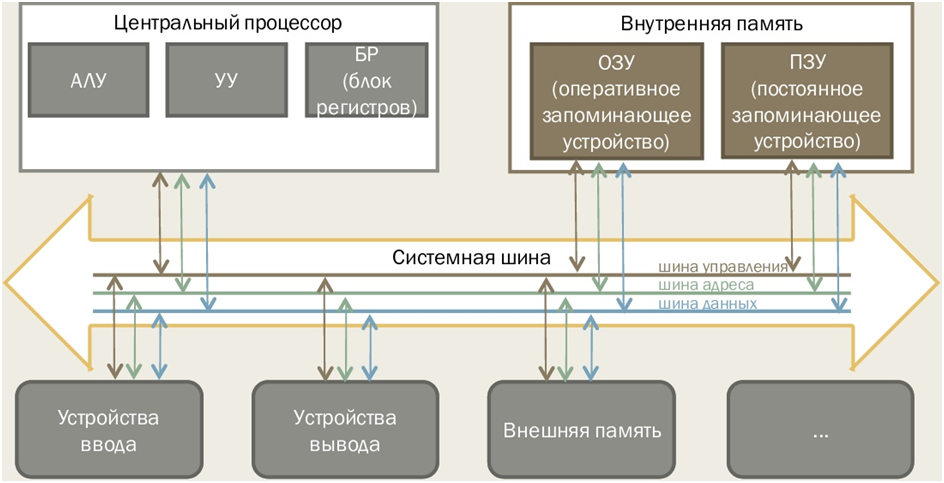
**Машинный код** - система команд конкретной вычислительной машины, которая интерпретируется непосредственно процессором.

**Язык ассемблера** - машинно-зависимый язык программирования низкого уровня, команды которого прямо соответствуют машинным командам.

## 3. Виды памяти ЭВМ. Структурная схема ЭВМ.

(это для подготовки к его вопросам про шину, для ответа можно просто перерисовать схему и перечислить виды памяти)

Сначала нужно разобраться со структурной схемой ЭВМ, на которой будет понятно, какие есть виды памяти:



Центральной частью является системная шина - абстракция, которая обеспечивает взаимодействие между собой всех устройств. Шина делится на три логические части: **шина управления, шина адреса и шина данных.** К шине подключен центральный процессор и внутренняя память, которая делится на ОЗУ и ПЗУ.

Через шину происходят все процессы обмена. Пусть процессор считывает какое-то значение из оперативной памяти. Тогда он на шину адреса передаёт адрес, откуда считать значение, а на шину управления отправляет сигнал: запрос значения из ячейки памяти. ОЗУ находит нужное значение, отправляет результат на шину данных и отправляет сигнал на шину управления.

По факту функцию шины выполняет материнская плата. На практике шин бывает несколько: между процессором и ОЗУ, USB.

Из схемы видим **основные виды памяти** в ЭВМ:

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. ОЗУ (оперативная память) - в неё загружают данные и программы, с её помощью работает компьютер в целом. Процессор имеет к нему прямой доступ. Очищается при отключении питания.
4. ПЗУ - также это память только для чтения. Нужна для запуска компьютера, так как оперативная память очищается. В ПЗУ хранится стартовая программа загрузка компьютера.
5. Внешняя память - различные накопители. Используются для долгосрочного хранения данных.

(дополнительная информация о запуске компьютера)Процессор в соответствии с архитектурой при подаче питания начинает работать с некоего 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.

## 3.1 Запуск и исполнение программы.

Чтобы из исходного кода получить машинный код, необходимо скомпилировать файл и получить исполняемый файл.

**Исполняемый файл** - файл, содержащий программу в виде, в котором она может быть исполнена компьютером.

Чтобы получить исполняемый файл, нужно произвести два шага:

1. Компиляция.
2. Линковка.

**Компилятор** - программа для преобразования исходного текста другой программы на определенном языке в объектный модуль.

**Компоновщик** - программа для связывания нескольких объектных файлов в один исполняемый.

В DOS и Windows расширения исполняемых файлов: .exe и .com

Последовательность запуска программы вот такая:

1. Определения формата файла - ОС определяет, можно ли такое вообще запустить.
2. Чтение и разбор заголовка (для exe).
3. Чтение разделов исполняемого файла в ОЗУ по необходимым адресам.
4. Дополнительная подготовка к запуску (подгрузка динамических библиотек).
5. Передача управления на точку входа - команду, с которой программа может начать свою работу. В .exe точка входа указывается в заголовке, для .com - это начало.

Последовательность для .com файла:

1. Выделение свободного сегмента данных нужного размера и передача адреса сегмента во все сегментные регистры.
2. В первые 100h памяти записывается служебная структура - PSP.
3. Загружается содержимое файла без изменений.
4. Указатель стека указывает на конец сегмента, в стек записывается адрес возврата 0000h.
5. Управление передаётся по адресу CS:100h.

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

## (вопросы 4-7 перекликаются между собой)

## 4. Сегментная модель памяти в архитектуре 8086.

**Сегментная адресация памяти** — схема логической адресации памяти компьютера в архитектуре [x86](https://ru.wikipedia.org/wiki/X86). В реальном режиме работы всё адресное пространство делится на одинаковые сегменты максимальным размером 65536 (2^16) байт. Начало каждого последующего сегмента (так называемая **База сегмента**) смещено относительно базы предыдущего на минимальный размер сегмента, то есть на 16 байт (т. н. параграф). Таким образом, сегменты могут частично перекрывать друг друга.

Линейный адрес конкретной ячейки памяти делится на две части: *сегмент* и *смещение*.

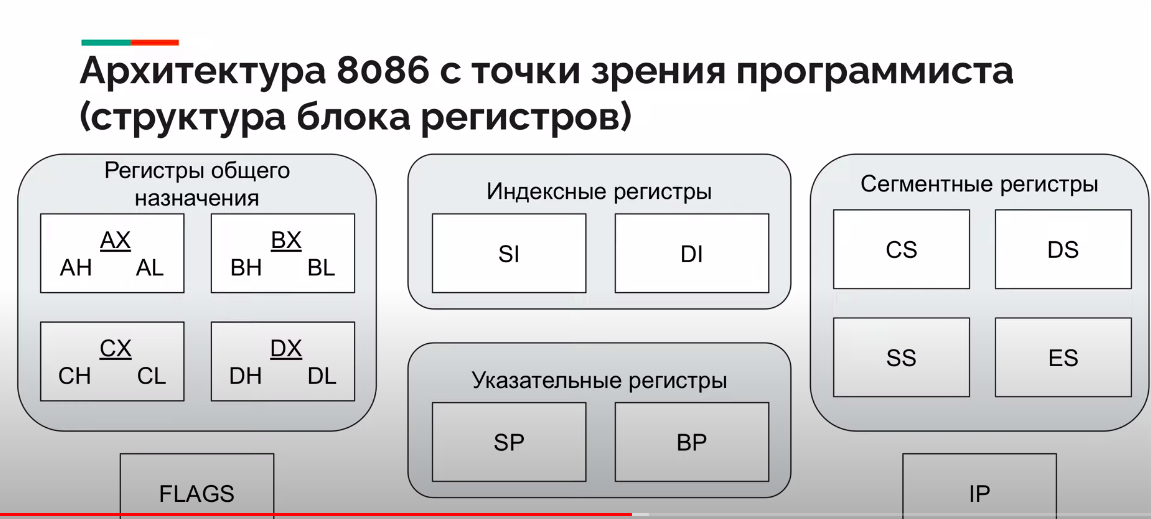
1. *Сегментом* называется условно выделенная область адресного пространства определённого размера.
2. С*мещением* — адрес ячейки памяти относительно начала сегмента.

В результате получается *сегментный (логический) адрес*, который соответствует линейному адресу *база сегмента*+*смещение*.

## 5. Процессор 8086. Регистры общего назначения.

Процессор 8086 - 16-разрядный процессор, выпущенный Intel в 1978 году. Реализованная в процессоре архитектура стала основой широко известной архитектуры [x86](https://ru.wikipedia.org/wiki/X86). Современные процессоры этой архитектуры сохраняют возможность выполнять все команды этого набора.

Архитектура 8086 подразумевает следующие регистры:



Регистры общего назначения: AX, BX, CX и DX. Размерность каждого - 2 байта. У каждого регистра есть младшая (Low) и старшая часть (High).

Являются взаимозаменяемые в основном, но для некоторых операций применимы конкретные регистры:

1. AX - предназначен для арифметических операций умножения и деления им.
2. BX - базовый регистр для вычислений адресов.
3. CX - используется в циклах и при повторении различных команд.
4. DX используется при обмене с внешними устройствами.

## 6. Процессор 8086. Сегментные регистры. Адресация в реальном режиме. Понятие сегментной части адреса и смещения.

Сегментные регистры отвечают за сегменты памяти. Каждый сегментный регистр определяет адрес начала сегмента в памяти, при этом сегменты могут совпадать или пересекаться. По умолчанию регистр CS используется при выборке инструкций, регистр SS при выполнении операций со стеком, регистры DS и ES при обращении к данным.

Реальный режим работы - режим совместимости современных процессоров с 8086. Разрядность шины адреса - 1 МБ (20 разрядов). Физический адрес получается сложением адреса начала сегмента (на основе сегментного регистра) и смещения.

Сегментный регистр хранит в себе старшие 16 разрядов (из 20) адреса начала сегмента. 4 младших разряда всегда нулевые, так что сегментный регистр хранит номер параграфа начала сегмента. Как вычислить физический адрес:

1

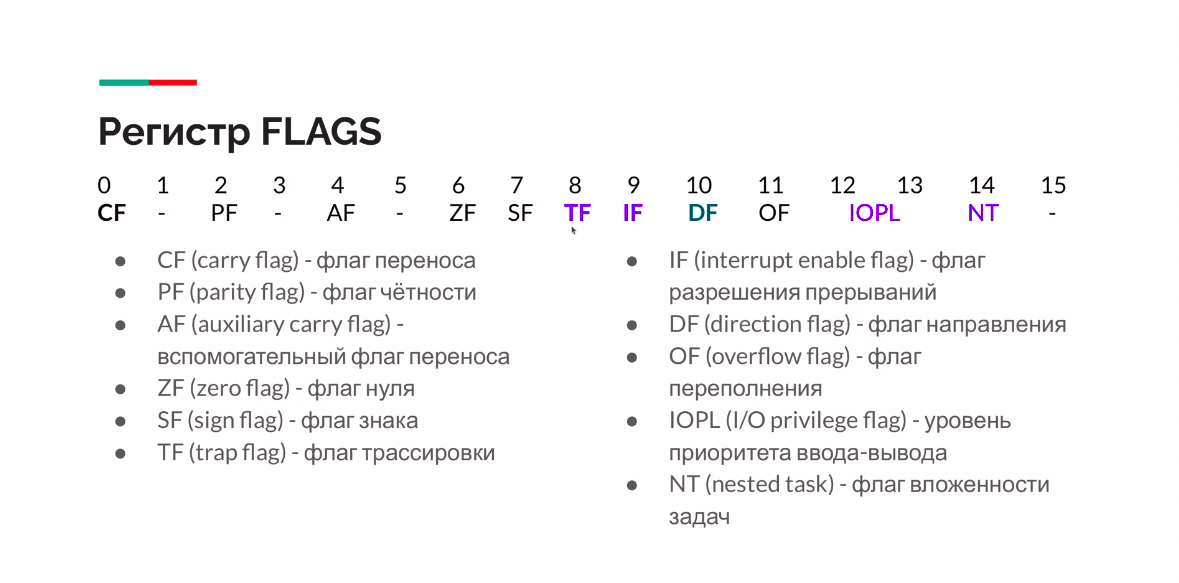
**Сегментная адресация памяти** — схема логической адресации памяти компьютера в архитектуре [x86](https://ru.wikipedia.org/wiki/X86). Линейный адрес конкретной ячейки памяти делится на две части: *сегмент* и *смещение*.

1. *Сегментом* называется условно выделенная область адресного пространства определённого размера.
2. С*мещением* — адрес ячейки памяти относительно начала сегмента.

В результате получается *сегментный (логический) адрес*, который соответствует линейному адресу *база сегмента*+*смещение* и который выставляется процессором на шину адреса.

## 7. Процессор 8086. Регистр флагов.

Регистр флагов - специальный регистр, к которому напрямую обратиться нельзя. Используется как отдельные 16 битов, которые имеют собственные значения (текстом после картинки):



Флаги делятся на три группы:

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

Некоторые флаги (CF, TF, IF, DF, IOPL, NT) можно изменить, остальные - нельзя.

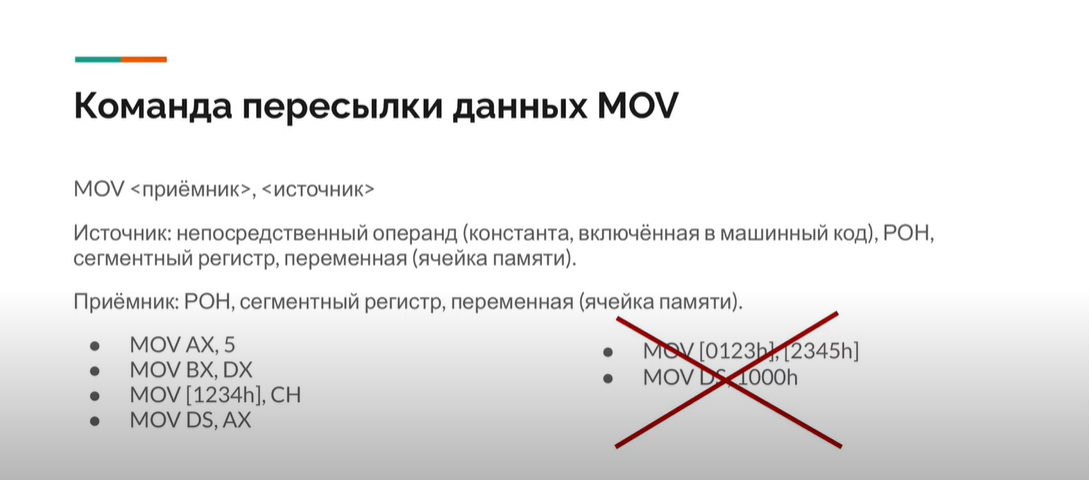
Назначение каждых флагов:

1. CF - флаг переноса. (carry flag). Возникает при работе с беззнаковыми числами, выставляется в 1, когда был осуществлён перенос с максимального бита.
2. PF - флаг четности. Выставляется в зависимости от того, сколько единиц получилось в младшем байте. Нужен для контроля корректности передачи данных.
3. AF - вспомогательный байт переноса. Нужен для работы с упакованными числами.
4. ZF - флаг нуля. Принимает 1, когда результат арифметической операции получился 0.
5. SF - флаг знака. Всегда равен старшему биту результата.
6. OF - флаг переполнения. Нужен, если идёт работа со знаковыми числами, и старший бит, который показывает знак, меняется.
7. IF - флаг разрешения прерываний. Если 1 - тогда прерывания разрешены.
8. TF - флаг трассировки. Устанавливается отладчиком для последовательного выполнения команд
9. DF - флаг направления. Используется для поточной обработки данных.
10. NT
11. IOPL

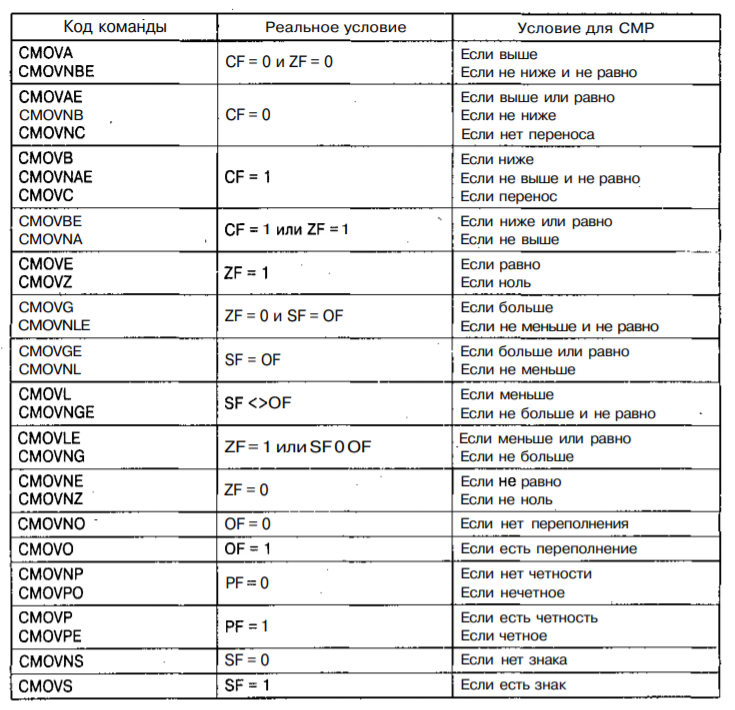
## 8. Команды пересылки данных

(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)

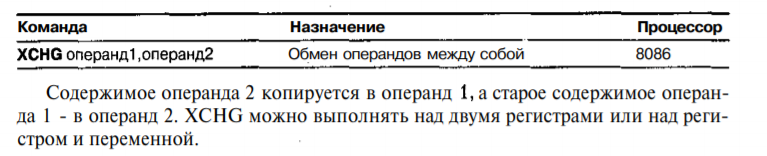
Первая команда - **mov**. Аналогия в языках высокого уровня - операция присвоения. Источник не изменяется, значение копируется в приёмник. Ограничение: нельзя скопировать переменную в переменную (процессору доступно поставить только один адрес в шину). В сегментный регистр нельзя записать напрямую команду.



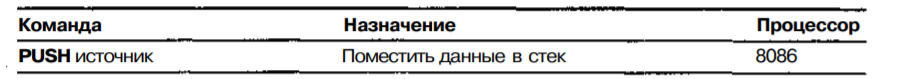
Следующий набор команд - **cmov.** Это набор команд, которые копируют содержимое источника в приемник, если удовлетворяется то или иное условие. Источником может быть регистр общего назначения или переменная, а приемником - только регистр. Требование, которое должно выполниться, - просто равенство нулю или единице тех или иных флагов из регистра FLAGS. Чаще всего используется в сочетании с командами сравнения. В таблице приведены все возможные варианты:



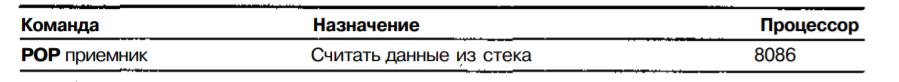
Команда **xchg** меняет значения операндов между собой. Операндами могут быть два регистра или регистр и переменная.



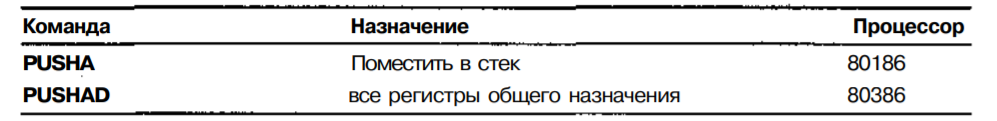
Команда **push** помещает значение источника в стек. Источником может быть рег+\*истр, сегментный регистр, непосредственный операнд или переменная. Фактически эта команда уменьшает ESP на размер источника в байтах (2 или 4) и копирует содержимое источника в память по адресу SS:[ESP].



Команда **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.



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.



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

Команда сравнения - **cmp**. Вычитает источник из приёмника, но результат не сохраняется, влияя лишь на флаги: CF, PF, AF, ZF, SF, OF. Применятся чаще всего совместно с командами условных переходов.



(Просто чтобы знать)Также есть команда **TEST,** которая влияет только на флаги, но выполняет логическое умножение. С ее помощью можно узнать значение отдельных битов.

## 10. Команды условной и безусловной передачи управления.

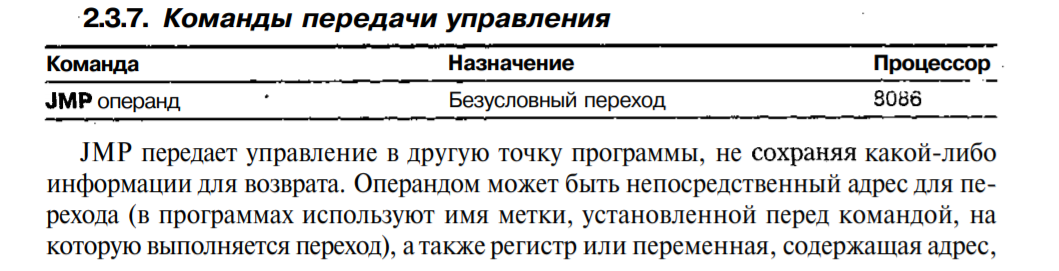
**Безусловный переход** - переход, который выполняется всегда.

**Условный переход** - переход, который выполняется тогда, когда выполняется определенное условие, заданное флагами процессора.

(Не обязательно, но Кузнецов может спросить). Виды переходов:

1. Короткий (short) - если адрес перехода находится в пределах 128 байтов от команды.
2. Ближний (near) - если адрес перехода находится в том же сегменте памяти.
3. Дальний (far) - адрес перехода находится в другом сегменте.

Команда безусловного перехода - **jmp**. JMP передает управление в другую точку программы, не сохраняя какой-либо информации для возврата. Операндом может быть непосредственный адрес для перехода (в программах используют имя метки, установленной перед командой, на которую выполняется переход)(прямой переход), а также регистр или переменная, содержащая адрес(косвенный переход). Если переход ближний или короткий, то jmp просто меняет значение IP, а если дальний - то записывается новое значение и в IP, и новый селектор в CS.



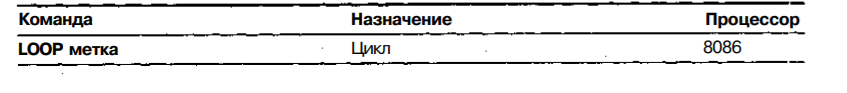
вКоманды условного перехода - **JCC(jmp<c><c>)**. Набор очень большой, но каждая инструкция сравнивает значения флаг

Команды условного перехода - JCC(jmp<c><c>). Набор очень большой, но каждая инструкция сравниает значения флагов, и если определенные условия верны, то произойдет переход по метке, которая передается в команду. Дальний переход не поддерживается!



Команды **JCXZ** и **JECXZ** выполняют переход, если значение регистра CX (ECX) = 0.

Набор команд **LOOP** используется для циклов: уменьшает значение CX на 1 и выполняет переход на метку, если ECX != 0. Переход на метку может быть только ближним.



## 11. Арифметические команды.

Целочисленная арифметика (основные команды)

● **ADD** <приёмник>, <источник> - выполняет арифметическое сложение приёмника и источника. Сумма помещается в приёмник, источник не изменяется.

* Приемник - переменная (область памяти), РОН (регистр общего назначения)
* Источник - то же самое, что приемник или непосредственный операнд (число)

Нельзя одновременно 2 переменных

● **SUB** <приёмник>, <источник> - арифметическое вычитание источника из приёмника.

* То же, что и для ADD

● **MUL** <источник> - беззнаковое умножение. Умножаются источник и AL/AX, в зависимости от размера источника.

* Источник - регистр или переменная
* Приемник - регистр AL, АХ, ЕАХ

Результат помещается в AX либо DX:AX.

Если источник - байт, то АХ = AL \* источник. Если источник - слово, то DX:AX = AX \* источник

● **DIV** <источник> - целочисленное беззнаковое деление. Делится AL/AX на источник. Результат помещается в AL/AX, остаток - в AH/DX/EDX.

* Приемник - AL, АХ или ЕАХ
* Источник - регистр или переменная

Если источник - байт, то AL = AX / источник. Если источник - слово, то АХ = DX:AX/источник.

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.

Еще есть команды деления/умножения со знаком IDIV, IMUL

● **INC** <приёмник> - инкремент на 1 , DEC <приёмник> - декремент на 1

* Обе команды работают быстрее ADD и SUB соответственно, потому что занимают 1 байт, а не 3.
* Не затрагивают флаг CF
* Эти команды изменяют регистр флагов (FLAGS) в зависимости от результата

## 12. Двоично-десятичная арифметика.

**Упакованный BCD-формат** - это упакованное двоично-десятичное число - байт от 00h до 99h (цифры A..F не задействуются).

*// для справки*

*Что такое неупакованное двоично-десятичное число?*

*Ответ: Десятичная цифра, хранящаяся в байте.*

*Что такое упакованное двоично-десятичное число?*

*Ответ: Две десятичные цифры, хранящиеся в полубайтах одного байта*

***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. «****D****ecimal Adjust* ***A****fter* ***A****ddition») — десятичная настройка после добавления;*

***DAS*** *(англ. «****D****ecimal Adjust* ***A****fter* ***S****ubtraction») — десятичная настройка после вычитания.*

*Обратите внимание, что в упакованном BCD-представлении отсутствует поддержка операций умножения и деления.*

////

## **DAA**

Команда **DAA** (Decimal Adjust AL after Addition) позволяет получать результат сложения упакованных двоично-десятичных данных в таком же упакованном BCD-формате. То есть она корректирует после сложения, пример:

mov AL,71H ; AL = 0x71h

add AL,44H ; AL = 0x71h + 0x44h = 0xB5h

daa ; AL = 0x15h

; CF = 1 - перенос является частью результата 71 + 44 = 115

*///*

*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**

Команда **DAS** (Decimal Adjust AL after Subtraction) позволяет получать результат вычитания упакованных двоично-десятичных данных в таком же упакованном BCD-формате. То есть она корректирует после вычитания, пример:

mov AL, 71H ; AL = 0x71h

sub AL, 44H ; AL = 0x71h - 0x44h = 0x2Dh

das ; AL = 0x27h

; CF = 0 - заем (перенос) является частью результата

**ASCII-формат - это неупакованное двоично-десятичное число (байт от 00h до 09h).**

///

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 АF = 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

///

## **AAA**

Команда **AAA** (ASCII Adjust After Addition) позволяет преобразовать результат сложения двоично-десятичных данных в ASCII-формат. Для этого команда AAA должна выполняться после команды двоичного сложения ADD, которая помещает однобайтный результат в регистр AL. Если будет перенос, он запишется в AH.

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'

## **AAS**

Команда **AAS** (ASCII Adjust After Subtraction) позволяет преобразовать результат вычитания двоично-десятичных данных в ASCII-формат. Для этого команда AAS должна выполняться после команды двоичного вычитания SUB, которая помещает однобайтный результат в регистр AL. Если был заем, будет вычитание 1 из AH.

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

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)

## AAM

## AAM imm8 (imm8 - система счисления aka ib ? вроде )

Команда **AAM** (ASCII Adjust AX After Multiply) позволяет преобразовать результат умножения неупакованных двоично-десятичных данных в ASCII-формат. Для этого команда AAM должна выполняться после команды беззнакового умножения MUL (но не после команды умножения со знаком IMUL), которая помещает двухбайтный результат в регистр AX.

Команда AAM распаковывает результат умножения, содержащийся в регистре AL, деля его на второй байт кода операции ib (равный 0x0Ah для безоперандной мнемоники AAM). Частное от деления (наиболее значащая цифра результата) помещается в регистр AH, а остаток (наименее значащая цифра результата) — в регистр AL .

Для того, чтобы преобразовать содержимое регистра AX к ASCII-формату, необходимо после команды AAM выполнить команду OR AX,0x3030h.

Пример:

mov AL, 3 ; множимое в формате неупакованного BCD помещается в регистр AL

mov BL, 9 ; множитель в формате неупакованного BCD помещается в регистр BL

mul BL ; AX = 0x03 \* 0x09 = 0x001Bh

aam ; AX = 0x0207h

or AX,3030H ; AX = 0x3237h, т.е. AH = '2', AL = '7'

## **AAD**

## AAD imm8

Команда AAD (ASCII Adjust AX Before Division) используется для подготовки двух разрядов неупакованных BCD-цифр (наименее значащая цифра в регистре AL, наиболее значащая цифра в регистре AH) для операции деления DIV, которая возвращает неупакованный BCD-результат.

Команда AAD устанавливает регистр AL в значение AL = AL + (imm8 \* AH) , где imm8 – это второй байт кода операции ib (равный 0x0Ah для безоперандной мнемоники AAD), с последующей очисткой регистра AH. После команды AAD регистр AX будет равен двоичному эквиваленту оригинального неупакованного двухзначного числа.

Пример:

mov AX, 0207H ; делимое в формате неупакованного BCD помещается в регистр AX

mov BL, 05H ; делитель в формате неупакованного BCD помещается в регистр BL

aad ; AX = 0x001Bh

div BL ; AX = 0x0205h

or AL, 30H ; AL = 0x35h = '5'

## 13. Команды побитовых операций. Логические команды.

● **AND** <приёмник - регистр, переменная>, <источник - число, регистр, переменная> - побитовое “И”.

Источник и приемник не могут быть переменными одновременно

Пример: AND al, 00001111b

● **OR** <приёмник - регистр, переменная>, <источник- число, регистр, переменная> - побитовое “ИЛИ”.

Источник и приемник не могут быть переменными одновременно

Пример: OR al, 00001111b

● **XOR** <приёмник- регистр, переменная>, <источник- число, регистр, переменная> - побитовое исключающее “ИЛИ”. Любой бит результата равен 1, если соответствующие биты операндов различны, и нулю - в противном случае.

Источник и приемник не могут быть переменными одновременно

Пример: XOR AX, AX

● **NOT** <приёмник> - инверсия, флаги не затрагиваются

● **TEST** <приёмник - регистр, переменная>, <источник - число, регистр, переменная> Аналог AND, но результат не сохраняется, выставляются флаги SF, ZF, PF, OF и CF обнуляются, значение AF не определено. TEST, так же как и СМР, используется в основном в сочетании с командами условного перехода (Jcc), условной пересылки данных (CMOVcc) и условной установки байтов (SETcc).

Источник и приемник не могут быть переменными одновременно

Все эти команды меняют регистр FLAGS.

*Логический, арифметический, циклический сдвиг. SAR, SAL, SHR, SHL, ROR, ROL, RCR, RCL*

* **SAR** приемник, счетчик - Арифметический сдвиг вправо
* **SAL** приемник, счетчик - Арифметический сдвиг влево
* **SHR** приемник, счетчик - Логический сдвиг вправо
* **SHL** приемник, счетчик - Логический сдвиг влево

Эти четыре команды выполняют двоичный сдвиг приемника (регистр или переменная) вправо (в сторону младшего бита) или влево (в сторону старшего бита) на значение счетчика (число или регистр CL, из которого учитываются только младшие 5 бит, принимающие значения от 0 до 31).

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

Команды SAL и SHL выполняют одну и ту же операцию (на самом деле это одна и та же команда) - на каждый шаг сдвига старший бит заносится в CF, все и младшбиты сдвигаются влево на одну позицию, ий бит обнуляется.

Команда SHR осуществляет прямо противоположную операцию: младший бит заносится в CF, все биты сдвигаются на 1 вправо, старший бит обнуляется. Эта команда эквивалентна беззнаковому целочисленному делению на 2. Команда SAR действует по аналогии с SHR, только старший бит не обнуляется, а сохраняет предыдущее значение. SAR, таким образом, эквивалентна 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.

Для сдвигов на несколько битов значение OF не определено. Флаги SF, ZF, PF назначаются всеми сдвигами в соответствии с результатом, параметр AF не определен (кроме случая, когда счетчик сдвига равен нулю: ничего не происходит и флаги не изменяются).

* **ROR** приемник, счетчик - Циклический сдвиг вправо
* **ROL** приемник, счетчик - Циклический сдвиг влево
* **RCR** приемник, счетчик - Циклический сдвиг вправо через флаг переноса CF
* **RCL** приемник, счетчик - Циклический сдвиг влево через флаг переноса

Эти команды осуществляют циклический сдвиг приемника (регистр или переменная) на число битов, указанное в счетчике (число или регистр CL, из которого учитываются только младшие 5 бит, принимающие значения от 0 до 31). При выполнении циклического сдвига на 1 команды ROR (ROL) перемещают каждый бит приемника вправо (влево) на одну позицию, за исключением самого младшего (старшего), который записывается в позицию самого старшего (младшего) бита.

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

После выполнения команд циклического сдвига флаг CF всегда равен последнему вышедшему за пределы приемника биту, флаг OF определен только для сдвигов на 1 - он устанавливается, если изменилось значение самого старшего бита, и сбрасывается в противном случае. Флаги SF, ZF, AF и PF не изменяются

Операции над битами и байтами BT, BTR, BTS, BTC, BSF, BSR, SETcc

*База - регистр или переменная.*

*Смещение - от 0 до 15/31 в зависимости от размера регистра(базы), иначе берётся остаток от деления на 16/32*

● **BT** <база>, <смещение> - считать в CF значение бита из битовой строки

Пример

bt ax,0 ;Проверка младшего бита AX

jc m1 ;Переход, если бит равен 1

mov cx,3 ;CX=3

bt ax,cx ;Проверка 3-го бита AX

jnc m1 ;Переход, если бит равен 0

● **BTS** <база>, <смещение> - установить бит в 1

● **BTR** <база>, <смещение> - сбросить бит в 0

● **BTC** <база>, <смещение> - инвертировать бит

● **BSF** <приёмник>, <источник> - прямой поиск бита (от младшего разряда)

● **BSR** <приёмник>, <источник> - обратный поиск бита (от старшего разряда)

*Сканирование битов выполняется командами BSF и BSR. Эти команды очень похожи. У них 2 операнда. Первый операнд должен быть 16-битным регистром, в него записывается результат. Второй операнд может быть 16-битным регистром или словом в памяти — это обрабатываемое значение.*

*Команда BSF просматривает биты второго операнда от младшего к старшему и помещает индекс первого единичного бита в регистр. Биты нумеруются, начиная с нуля. Если единичный бит найден, то флаг нуля сбрасывается (ZF=0). Если все биты нулевые, то флаг нуля устанавливается (ZF=1), а значение первого операнда будет неопределённым (на разных процессорах может быть по-разному).*

Пример:

mov ax,01011000b ;AX=58h

bsf bx,ax ;BX=3, ZF=0

xor ax,ax ;AX=0

bsf bx,ax ;BX=?, ZF=1

● **SETcc** <приёмник> - выставляет приёмник (1 байт) в 1 или 0 в зависимости от условия, аналогично Jcc

## 14. Команды работы со строками.

Строковые операции: копирование, сравнение, сканирование, чтение, запись Строка-источник - DS:SI, строка-приёмник - ES:DI.

Работает с: SB - байт, SW - слово, SD - двойное слово

● **MOVS/MOVSB/MOVSW** <приёмник>, <источник> - копирует байт или слово из приемника в источник. После выполнения команды, регистры SI и DI увеличиваются на 1 (или 2), если флаг DF = 0, или уменьшаются на 1 (или 2), если DF = 1

● **CMPS/CMPSB/CMPSW** <приёмник>, <источник> - Сравнивает байт или слово из приемника с источником. После выполнения команды, регистры SI и DI увеличиваются на 1 (или 2), если флаг DF = 0, или уменьшаются на 1 (или 2), если DF = 1.

● **SCAS/SCASB/SCASW** <приёмник> - Сканирование (сравнение с AL/AX). После выполнения команды, регистры SI и DI увеличиваются на 1 (или 2), если флаг DF = 0, или уменьшаются на 1 (или 2), если DF = 1.

● **LODS/LODSB/LODSW** <источник> - Чтение (в AL/AX). После выполнения команды, регистры SI и DI увеличиваются на 1 (или 2), если флаг DF = 0, или уменьшаются на 1 (или 2), если DF = 1.

● **STOS/STOSB/STOSW** <приёмник> - Запись (из AL/AX). После выполнения команды, регистры SI и DI увеличиваются на 1 (или 2), если флаг DF = 0, или уменьшаются на 1 (или 2), если DF = 1.

Префиксы: REP/REPE/REPZ/REPNE/REPNZ

**REP** - повторить следующую строковую операцию

**REPE/REPZ** - повторить след. строковую операцию, если равно/если нуль *// повторять команду, пока CX≠0 или ZF≠0;*

**REPNE/REPNZ** - повторить след. строковую операцию, если не равно/не нуль *//повторять команду, пока CX≠0 или ZF=0.*

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

Пример использования: REP LODS AX

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

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

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

## 15. Команда трансляции по таблице.

XLAT/XLATB - трансляция в соответствии с таблицей

XLAT [адрес]

адрес - сегментный регистр, если не указан, то по умолчанию DS

**XLATB**

Помещает в AL байт из таблицы по адресу **DS:BX** (*в Зубкове ЕS:BX - это вроде неправильно*) со смещением относительно начала таблицы, равным AL.

Адрес, указанный в исходном коде, не обрабатывается компилятором и служит в качестве комментария.

Если в адресе явно указан сегментный регистр, он будет использоваться вместо DS.

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 он не нужен, можно применить форму записи XLATB.

**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 С.

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

**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 - вычисление эффективного адреса

LEA <приёмник>, <источник>

Вычисляет эффективный адрес источника и помещает его в приёмник. С помощью LEA можно вычислить адрес переменной, которая описана сложным методом адресации, например по базе с индексированием. 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. Позволяет вычислить адрес, описанный сложным методом адресации. По сути, является аналогом операции mov ПРИЕМНИК, offset ИСТОЧНИК.

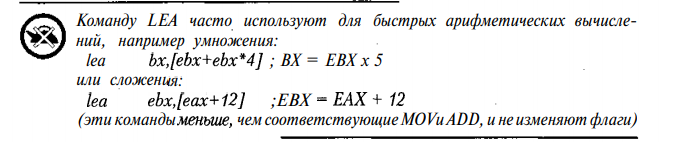
Иногда используется для быстрых арифметических вычислений:

lea bx, [bx+bx\*4]

lea bx, [ax+12]

Эти вычисления занимают меньше памяти, чем соответствующие MOV и ADD, и не изменяют флаги.

Тот же пример красивых вычислений:



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

(структура программы: У Зубкова страница 106)

Структура программы на ассемблере

● Модули (файлы исходного кода)

○ Сегменты (описание блоков памяти)

■ команды процессора;

■ инструкции описания структур данных, выделения памяти для переменных и констант;

■ макроопределения.

Полный формат строки:

метка команда/директива операнды ; комментарий

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

**Директивы ассемблера** - ключевые слова в тексте программы на языке ассемблера, влияющие на процесс ассемблирования или свойства выходного файла.

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

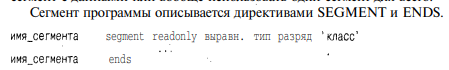
Пример:

имя SEGMENT [READONLY] выравнивание тип разряд 'класс'

…

имя ENDS

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

**

*Параметры:*

*● Выравнивание - расположение начала сегмента с адреса, кратного какому-либо значению. Варианты: BYTE, WORD (2 байта), DWORD (4 байта), PARA (16 байт, по умолчанию), PAGE (256 байт).*

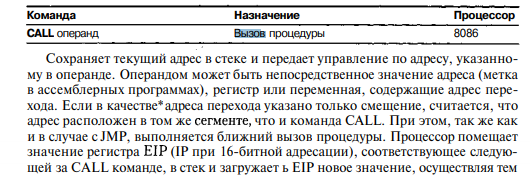
*● Тип: PUBLIC (сегменты с одним именем объединятся в один); STACK (для стека); COMMON (сегменты будут “наложены” друг на друга по одним и тем же адресам памяти); AT <начало> - расположение по фиксированному физическому адресу, параметр - сегментная часть этого адреса; PRIVATE - вариант по умолчанию.*

*● Класс - метка, позволяющая объединить сегменты (расположить в памяти друг за другом)*

**

## 18. Подпрограммы. Объявление, вызов.

Процедурой в ассемблере является все то, что в других языках называют подпрограммами, функциями, процедурами и т. д. Ассемблер не накладывает на процедуры никаких ограничений - 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 труд- . нечитаемым программам, и в язык ассемблера были включены директивы логического оформления процедур.



Синтаксис описания процедуры:

**ИмяПроцедуры** PROC язык расстояние

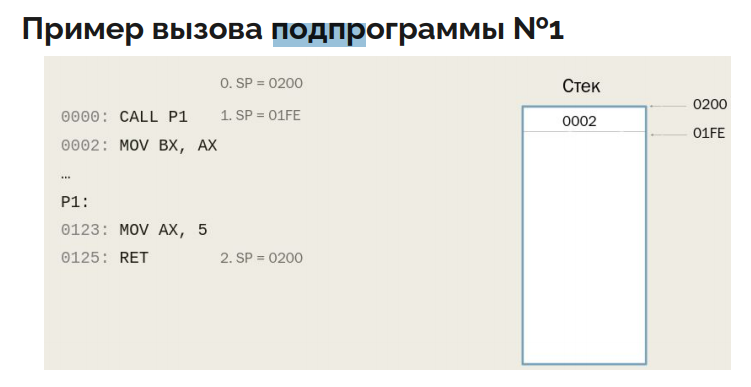
; тело процедуры

**ИмяПроцедуры** ENDP

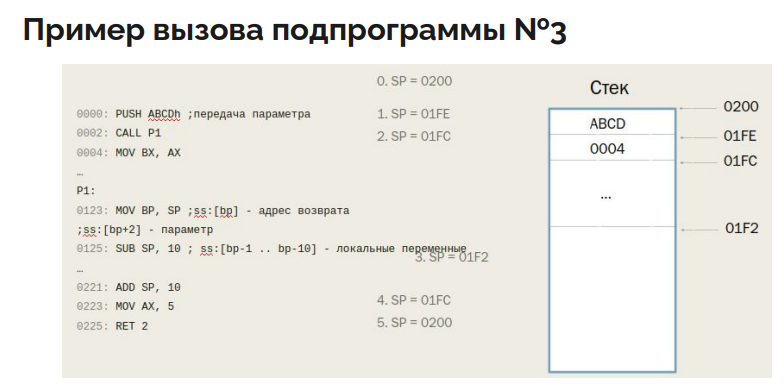
При вызове подпрограммы параметры в большинстве случаев помещают в стек, а в ЕВР записывают текущее значение ESP. Если подпрограмма использует стек для хранения локальных переменных, ESP изменится, но ЕВР можно будет использовать для того, чтобы считывать Значения параметров напрямую из стека

BP – base pointer

●Используется в подпрограмме для сохранения "начального" значения SP

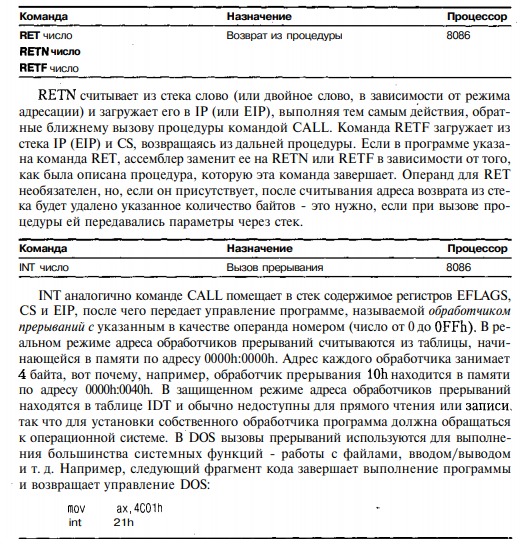


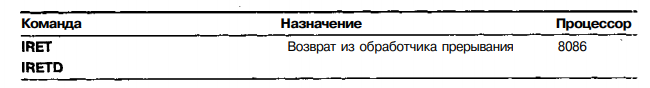




## 19. Подпрограммы. Возврат управления.

Процедурой в ассемблере является все то, что в других языках называют подпрограммами, функциями, процедурами и т. д. Ассемблер не накладывает на процедуры никаких ограничений - на любой адрес программы можно передать управление командой CALL, и оно вернется к вызвавшей процедуре, как только встретится команда RET.



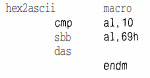


Возврат управления из обработчика прерывания или исключения. IRET загружает из стека значения IP, CS и FLAGS, a IRETD - EIP, CS и EFLAGS соответственно. Единственное отличие IRET от RETF состоит в том, что значение регистра флагов восстанавливается, из-за чего многим обработчикам прерываний приходится изменять величину EFLAGS, находящегося в стеке, чтобы, например, вернуть флаг CF, установленный в случае ошибки.

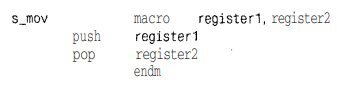
## 20. Макроопределения.

Макроопределением (или макросом) называется участок программы, которому присвоено имя и который ассемблируется всякий раз, когда ассемблер встречает это имя в тексте программы. Макрос начинается директивой MACRO и заканчивается ENDM.

Пример:

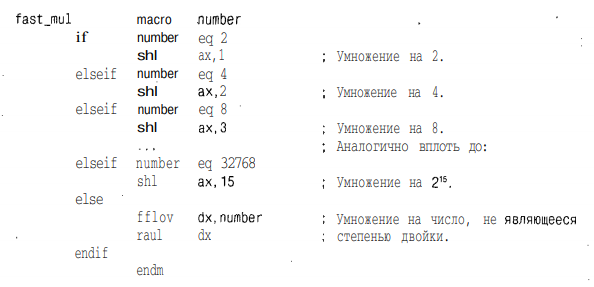


Сейчас в программе можно использовать слово hex2ascii, как если бы это было имя команды, и ассемблер заменит каждое такое слово на три команды, содержащиеся в макроопределении. Разумеется, можно оформить этот же участок кода в виде процедуры и вызывать его командой CALL - если процедура вызывается больше одного раза, этот вариант программы займет меньше места, но **вариант с макроопределением станет выполняться быстрее**, так как в нем не будет лишних команд CALL и RET. Однако скорость выполнения - не главное преимущество макросов. В отличие от процедур **макроопределения могут вызываться с параметрами**, следовательно, в зависимости от ситуации, включаемый код будет немного различаться, например:



Теперь можно использовать S\_MOV вместо команды MOV для того, чтобы скопировать значение из одного сегментного регистра в другой. Следующее важное средство, использующееся в макроопределениях, - **директивы условного ассемблирования**.

Например: напишем макрос, выполняющий умножение регистра АХ на число, причем, если множитель - степень двойки, то умножение будет выполняться более быстрой командой сдвига влево.



Можно, конечно, усложнить этот макрос, применяя особые свойства команды LEA и ее комбинации, сдвиги и сложения, однако в нынешнем виде он чрезмерно громоздкий. Проблема решается с помощью третьего средства, постоянно использующегося в макросах, - **блоков повторений**.

## 21. Стек. Аппаратная поддержка вызова подпрограмм.

**Стек** - организованный специальным образом участок памяти

**Использование:**

1. Временное хранение переменных
2. Передача параметров вызываемым подпрограммам
3. Сохранение адреса возврата при вызове процедур и прерываний

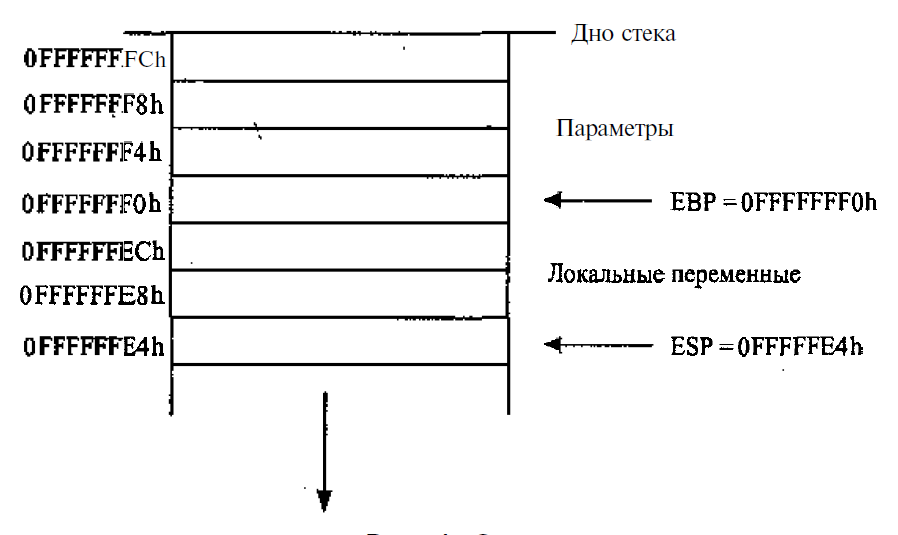
**Сегмент стека** - область памяти программы, используемая её подпрограммами, а также (вынужденно) обработчиками прерываний

**Расположение стека**: сегмент памяти, описываемый регистром SS

**Где отражается текущее смещение вершины стека?**: регистр ESP

Значение смещения при записи уменьшается, “рост вниз”

**При запуске программы SP указывает на конец сегмента**



При вызове подпрограммы параметры в большинстве случаев помещают в стек,

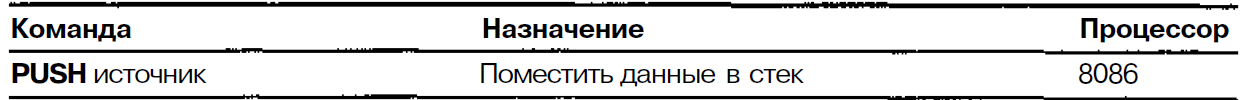
а в ЕВР записывают текущее значение ESP. Если подпрограмма использует стек

для хранения локальных переменных, ESP изменится, но ЕВР можно будет ис-

пользовать для того, чтобы считывать Значения параметров напрямую из стека

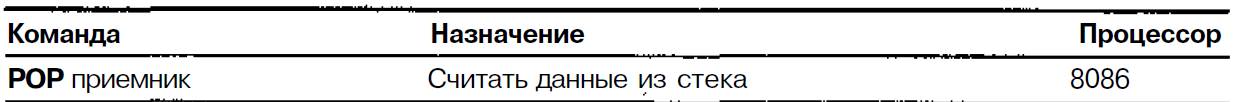
(их смещения запишутся как ЕВР + номер параметра).

**Команды для работы со стеком**

****

Уменьшает SP на размер источника в байтах(2 или 4) и записывает значение по адресу SS:SP.

Команда PUSH почти всегда используется в паре с POP.



Помещает в приемник слово или двойное слово, находящееся в вершине сте-

ка, увеличивая SP на 2 или 4 соответственно. POP выполняет действие, полно-

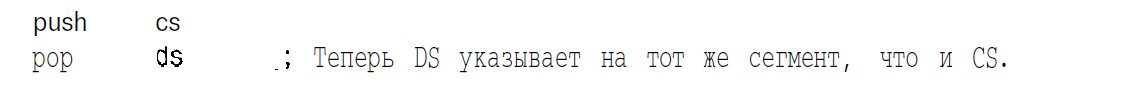
стью обратное PUSH.

Приемник: регистр общего назначения, сегментный регистр (кроме CS (чтобы загрузить CS из стека, надо воспользоваться командой RET)), переменная

Если в роли приемника выступает операнд, использующий ESP для косвенной адресации, команда POP вычисляет адрес операнда уже после того, как она увеличивает ESP.

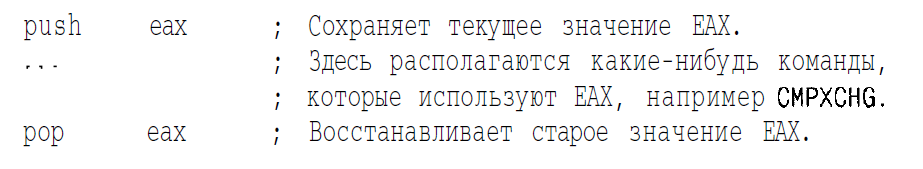
Совместное использование push и pop:

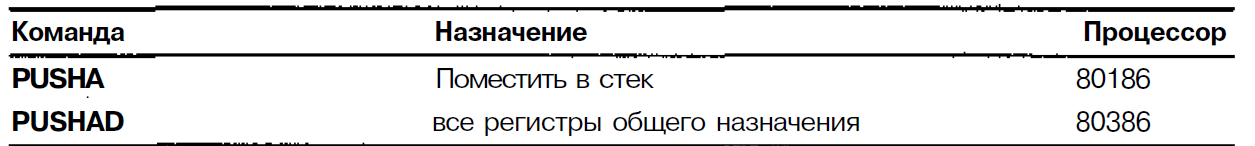
Чтобы **скопировать содержимое одного сегментного регистра в другой** (что нельзя выполнить одной командой MOV), можно использовать такую последовательность команд:



Другой вариант применения команд PUSH/POP - **временное хранение пере-**

**менных**, например:



****

PUSHA располагает в стеке регистры в следующем порядке: АХ, СХ, DX, ВХ, SP, ВР, SI и DI.

PUSHAD помещает в стек EAX, ECX, EDX, EBX, ESP, EBP, ESI и EDI. (В случае с SP и ESP используется значение, которое находилось в регист-

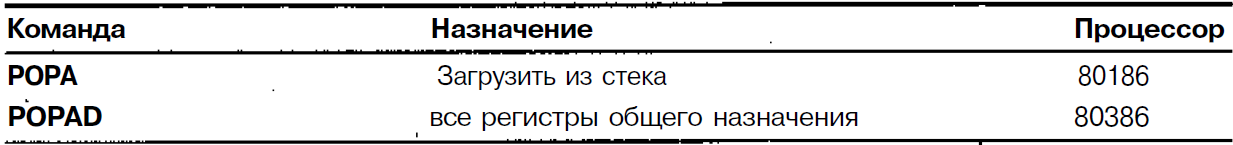
ре до начала работы команды.)

В паре с командами POPA/POPAD, считывающими эти же регистры из стека в обратном порядке, это позволяет писать подпро-

граммы (обычно обработчики прерываний), которые не должны изменять

значения регистров по окончании своей работы. В начале такой подпрограммы

вызывают команду PUSHA, а в конце - РОРА.

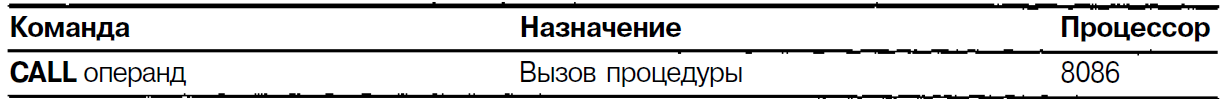


Команды выполняют действия, полностью обратные действиям PUSHA

и PUSHAD, но помещенное в стек значение SP или ESP игнорируется.

РОРА загружает из стека DI, SI, ВР, увеличивает SP на два, загружает ВХ, DX, СХ, АХ, a POPAD загружает EDI, ESI, ЕВР, увеличивает ESP на 4 и загружает EBX, EDX, ECX, EAX.

**Аппаратная поддержка вызова подпрограмм:**

****

* Сохраняет адрес следующей команды в стеке (уменьшает SP и записывает по его адресу IP либо CS:IP, в зависимости от размера аргумента). *Процессор помещает значение регистра EIP (IP при 16-битной адресации), соответствующее следующей за CALL команде, в стек и загружает в EIP новое значение, осуществляя тем самым передачу управления.*
* Операндом может быть непосредственное значение адреса (метка

в ассемблерных программах), регистр или переменная, содержащие адрес пере-

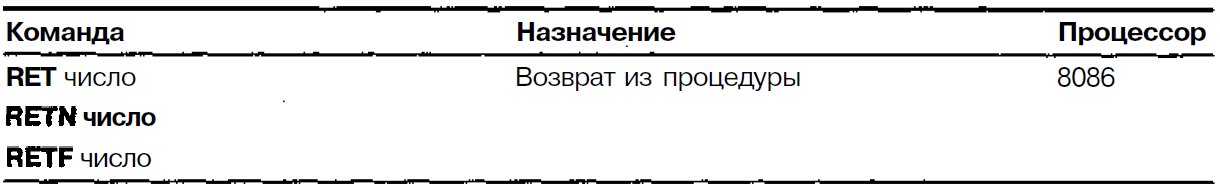
хода.

* Если в качестве адреса перехода указано только смещение, считается, что

адрес расположен в том же сегменте, что и команда CALL. При этом, так же как

и в случае с JMP, выполняется ближний вызов процедуры.

* 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.
* Чтобы выполнить дальний CALL в реальном режиме, режиме V86 или в защищенном режиме при переходе в сегмент с теми же привилегиями, процессор помещает в стек значения регистров CS и EIP (IP при 16-битной адресации) и осуществляет дальний переход аналогично команде JMP.

****

**Загружает из стека адрес возврата, увеличивает SP**

**Если указан операнд, его значение будет дополнительно прибавлено к SP для очистки стека от параметров**

**RETN -** считывает из стека слово (или двойное слово, в зависимости от режима

адресации) и загружает его в IP (или EIP), выполняя тем самым действия, обрат-

ные ближнему вызову процедуры командой CALL.

**RETF** - загружает из стека IP (EIP) 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.

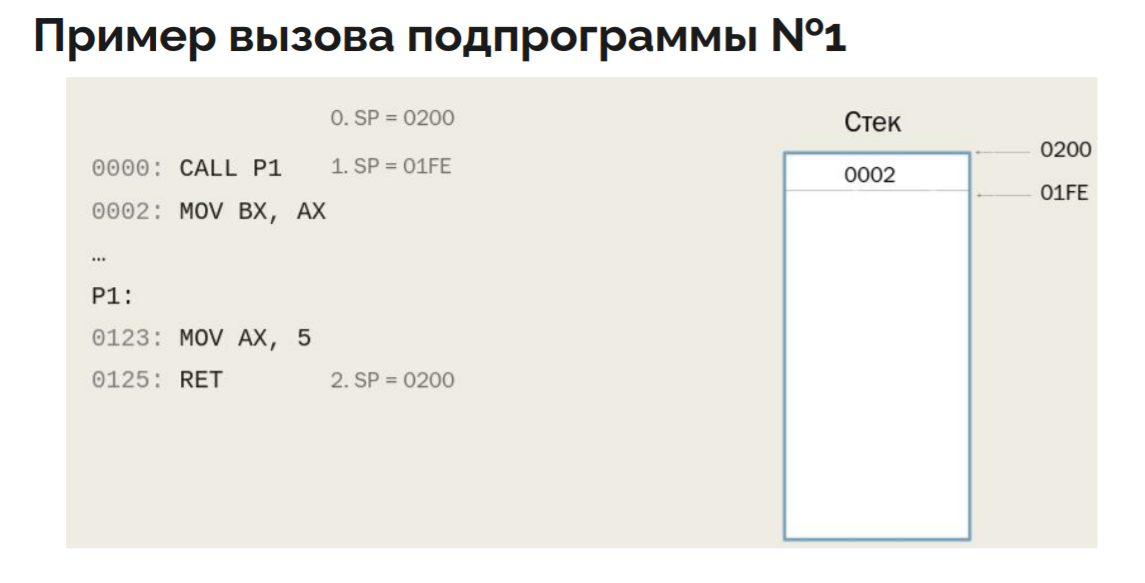
Операнд для RET необязателен, но, если он присутствует, после считывания адреса возврата из стека будет удалено указанное количество байтов - это нужно, если при вызове процедуры ей передавались параметры через стек.

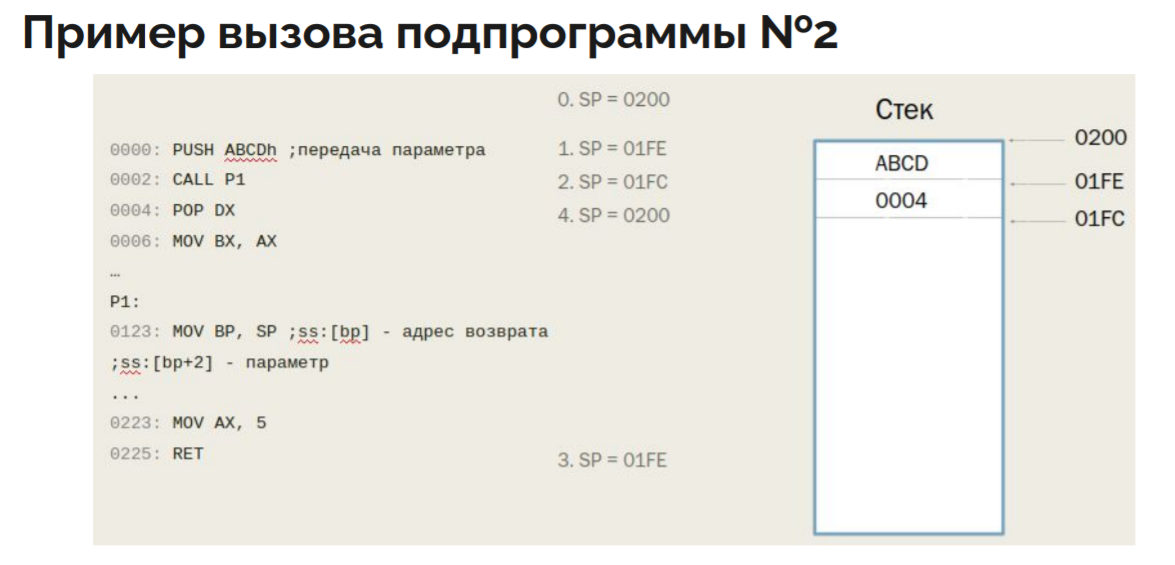
**BP – base pointer**

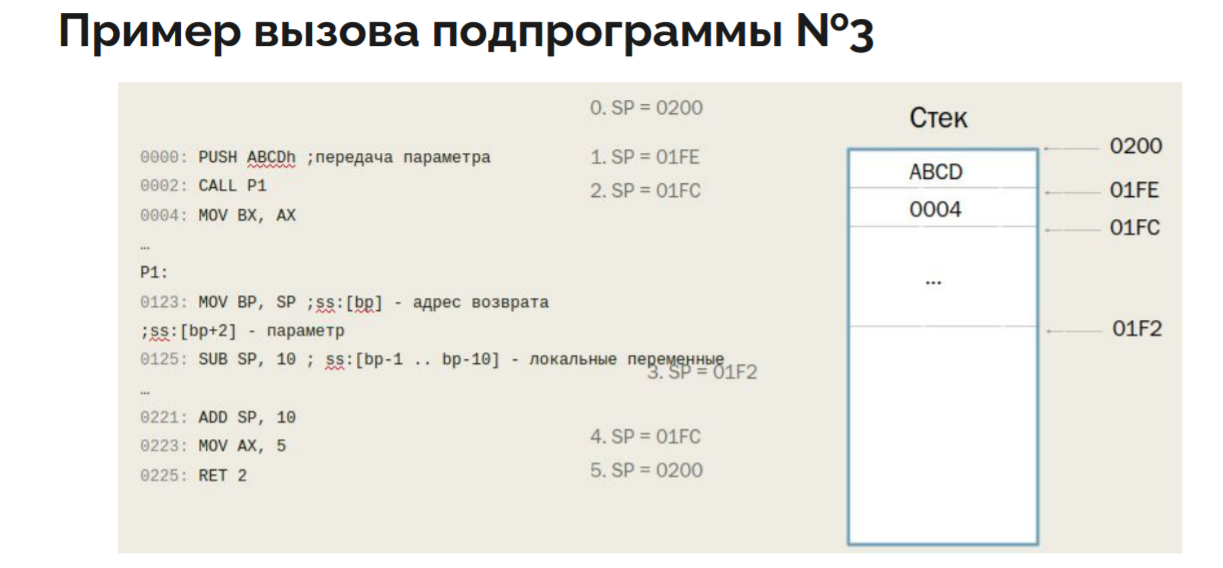
● Используется в подпрограмме для сохранения "начального" значения SP

● Адресация параметров

● Адресация локальных переменных

****

****

****

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

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

*Прерывание можно рассматривать как некоторое особое событие в системе, требующее моментальной реакции. Практически все системы ввода/вывода в компьютере работают с использованием прерываний. Например: нажатие клавиши, щелчок мышкой - аппаратура вырабатывает прерывания. В ответ на них система, соответственно, считывает код нажатой клавиши или запоминает координаты курсора мыши.*

Виды прерываний:

* аппаратные (асинхронные) - события от внешних устройств;
* внутренние (синхронные) - события в самом процессоре, например, деление на ноль;
* программные - вызванные командой int.

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

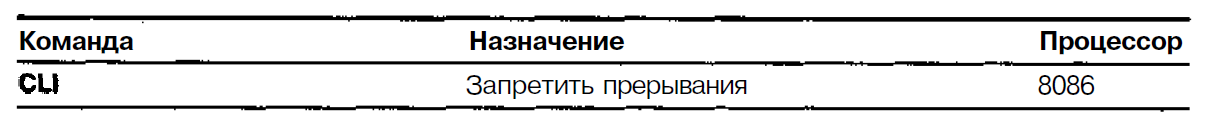
Замаскировать можно прерывания аппаратные(асинхронные)

Чтобы гарантировать непрерывное выполнение определенной последовательности команд при наличии критических участков можно использовать команду CLI, а в конце расположить команду STI.

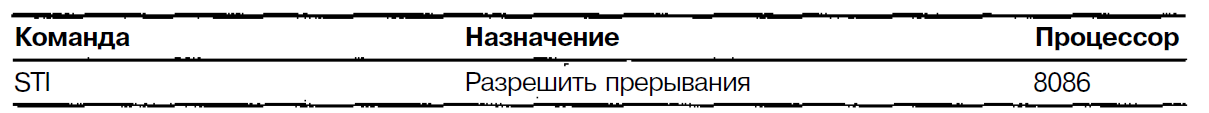
Команда CLI запрещает только маскируемые прерывания, на обработку немаскируемого прерывания эта команда никакого влияния не оказывает.

IF - флаг прерываний. Сброс этого флага в 0 приводит к тому, что процессор

перестает обрабатывать прерывания от внешних устройств.



Сбрасывает флаг IF в 0.



Устанавливает флаг IF в 1, отменяя тем самым действие команды CLI.

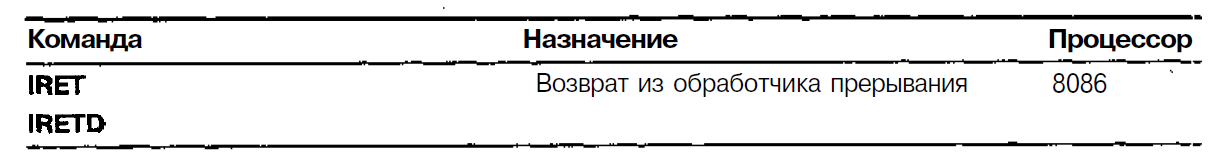
**Вектор прерывания** — номер, который идентифицирует соответствующий обработчик прерываний. Векторы прерываний объединяются в **таблицу векторов прерываний**, содержащую адреса обработчиков прерываний.

● Располагается в самом начале памяти, начиная с адреса 0. Находится в диапазоне адресов от 0000:0000 до 0000:03FFh

● Доступно 256 прерываний.

● Каждый вектор занимает 4 байта - полный адрес.

● Размер всей таблицы - 1 Кб.



Возврат управления из обработчика прерывания или исключения.

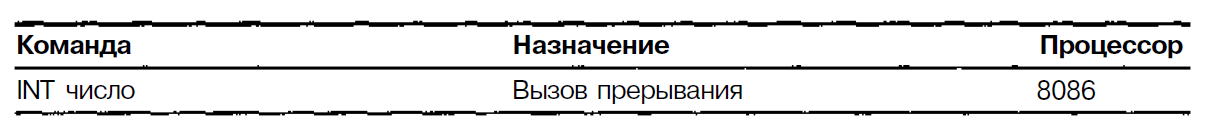
IRET загружает из стека значения IP, CS и FLAGS, a IRETD - EIP, CS и EFLAGS соот-

ветственно.

Единственное отличие IRET от RETF состоит в том, что значение регистра флагов восстанавливается, из-за чего многим обработчикам прерываний

приходится изменять величину EFLAGS, находящегося в стеке, чтобы, например,

вернуть флаг CF, установленный в случае ошибки.



INT аналогично команде CALL помещает в стек содержимое регистров EFLAGS,

CS и EIP, после чего передает управление программе, называемой обработчиком

прерываний с указанным в качестве операнда номером (число от 0 до OFFh).

Алгоритм перехвата прерывания

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

Существуют специальные функции, предназначенный для замены прерываний.

Если вы будете использовать эти функции, MS-DOS гарантирует, что операция по замене вектора будет выполнена правильно.

1. Чтение вектора

* Прерывание 21h
* AH = 35h
* AL = номер прерывания в таблице
* После выполнения функции в регистрах ES:BX будет искомый адрес обработчика прерывания.

1. Установка нового обработчика

* Прерывание 21h
* AH = 25h
* AL = номер прерывания в таблице
* Адрес обработчика прерываний следует передать через регистры DS:DX

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

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

Микропроцессор i80386 — первая модель 32-разрядного микропроцессора фирмы Intel.

32-разрядные:

● Регистры, кроме сегментных

● Шина данных

● Шина адреса (2^32 = 4Гб ОЗУ)

**Режимы работы:**

1. "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

1. "Защищённый" режим

● обращение к памяти происходит по виртуальным адресам с использованием механизмов защиты памяти;

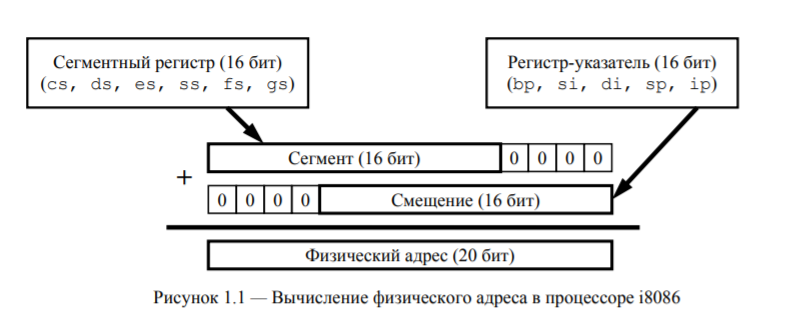
● набор доступных операций определяется уровнем привилегий (кольца защиты): системный и пользовательский уровни

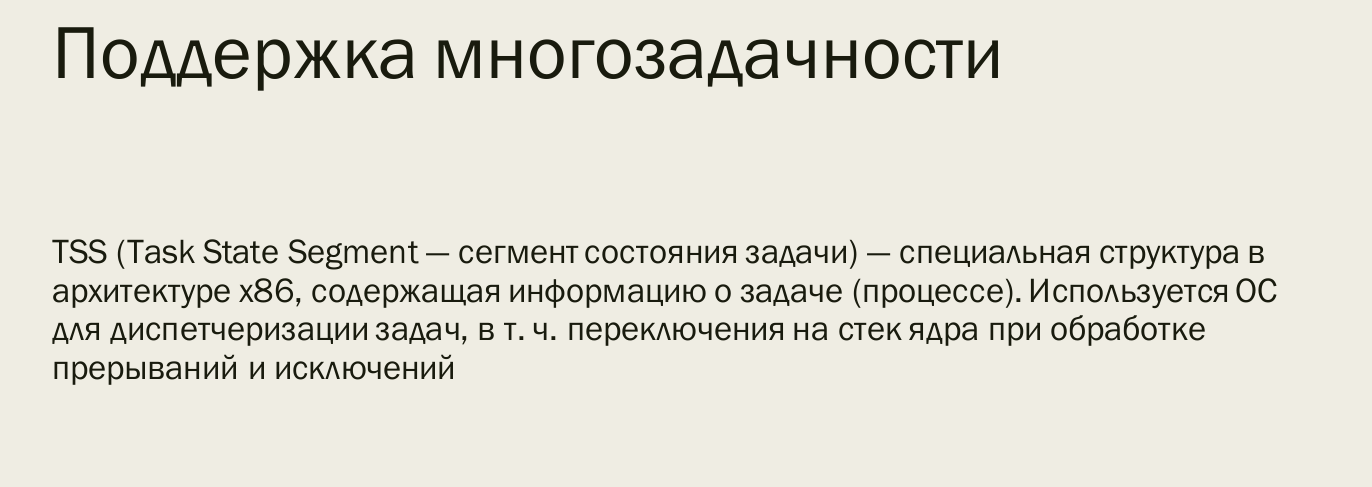
1. Режим виртуального i8086.

Переход в этот режим возможен, если процессор уже находится в защищённом режиме. В виртуальном

режиме, в отличие от защищённого, возможна работа программ реального режима.

Название «защищённый» режим проистекает от возможности запрещать доступ к определённым областям памяти (защищать определённые области памяти). Механизм формирования физического адреса, выдаваемого на адресную шину, показан на рисунке. Таким образом, адресация ячейки памяти производится по номеру сегмента и эффективному адресу ячейки в сегменте (называемому также смещением).

**Многозадачность**

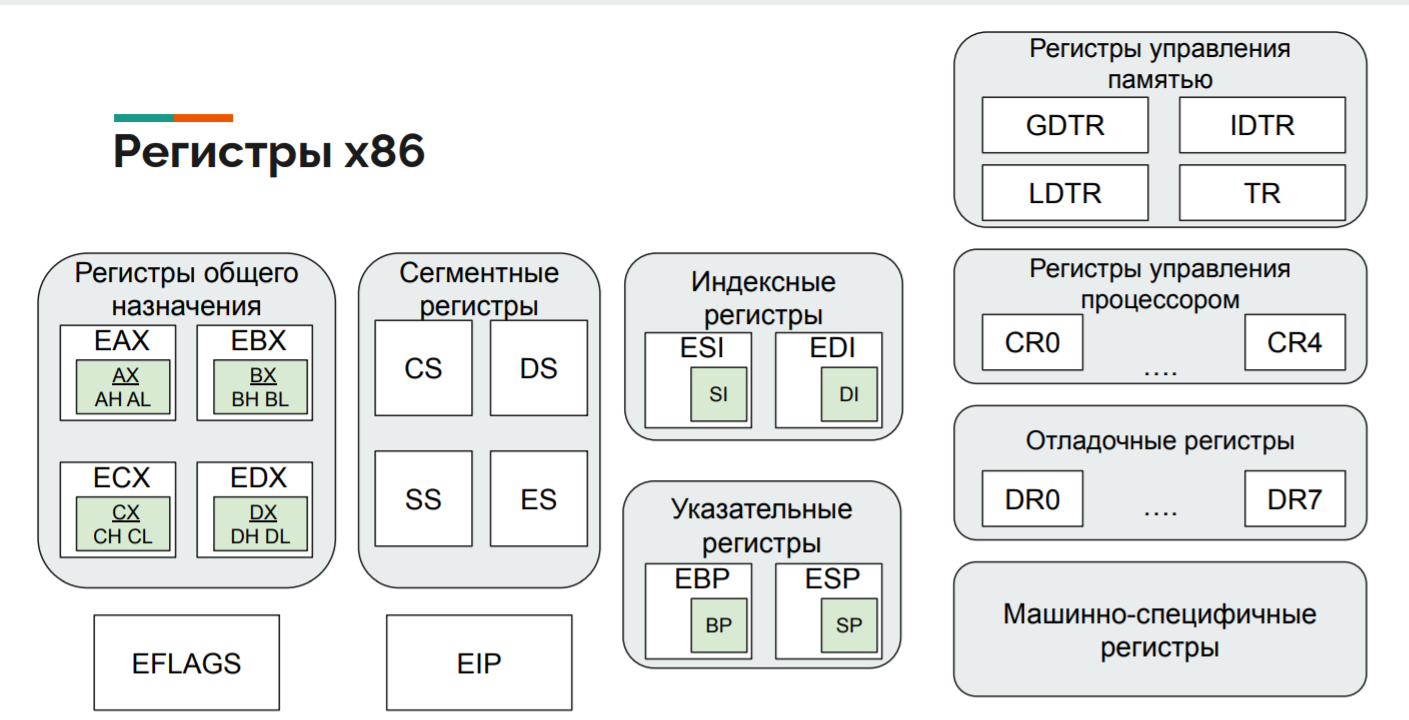
****

**ЗоЧем это нужно?**

В защищённом режиме используются полные возможности 32-разрядного процессора — обеспечивается непосредственный доступ к 4 Гбайт физического адресного пространства и многозадачный режим с параллельным выполнением нескольких программ (процессов). Микропроцессор предоставляет необходимый для этого режима надежный механизм защиты задач друг от друга [с помощью четырёхуровневой системы привилегий](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BB%D1%8C%D1%86%D0%B0_%D0%B7%D0%B0%D1%89%D0%B8%D1%82%D1%8B). Также в этом режиме доступна страничная организация памяти, повышающая уровень защиты задач друг от друга и эффективность их выполнения.

**Регистры**

Регистры МП i80386 являются расширением регистров прежних МП: 8086, 80186, 80286. Все 16-разрядные регистры МП предыдущих поколений содержатся внутри 32-разрядной архитектуры i80386.

****

Добавлены регистры поддержки работы в защищенном режиме (обеспечивание разделения доступа программ между собой, между программами и ОС и тд;

**Регистр EFLAGS** FLAGS + 5 специфических флагов

**Регистры управления памятью**

● GDTR: 6-байтный регистр, содержит 32-битный линейный адрес начала таблицы глобальных дескрипторов (GDT) и 16-битный размер (лимит, уменьшенный на 1)

● IDTR: 6-байтный регистр, содержит 32-битный линейный адрес начала таблицы глобальных дескрипторов обработчиков прерываний (IDT) и 16-битный размер (лимит, уменьшенный на 1)

● LDTR: 10-байтный регистр, содержит 16-битный селектор для GDT и весь 8-байтный дескриптор из GDT, описывающий текущую таблицу локальных дескрипторов

● TR: 10-байтный регистр, содержит 16-битный селектор для GDT и весь 8-байтный дескриптор из GDT, описывающий TSS текущей задачи

**Регистры управления процессором**

● CR0 - флаги управления системой

○ PG - включение режима страничной адресации

○ управление отдельными параметрами кеша

○ WP - запрет записи в страницы "только для чтения"

○ NE - ошибки FPU вызывают исключение, а не IRQ13

○ TS - устанавливается процессором после переключения задачи

○ PE - включение защищённого режима

● CR1 - зарезервирован

● CR2 - регистр адреса ошибки страницы - содержит линейный адрес страницы, при обращении к которой произошло исключение #PF

● CR3 - регистр основной таблицы страниц

○ 20 старших бит физического адреса начала каталога таблиц либо 27 старших бит физического адреса начала таблицы указателей на каталоги страниц, в зависимости от бита PAE в CR4

○ Управление кешированием и сквозной записью страниц

● CR4 - регистр управления новыми возможностями процессоров (с Pentium)

**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

**Машинно-специфичные регистры**

● Управление кешем

● Дополнительное управление страничной адресацией

● Регистры расширений процессора: MMX и т.д.

**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



## 24. Математический сопроцессор. Типы данных.

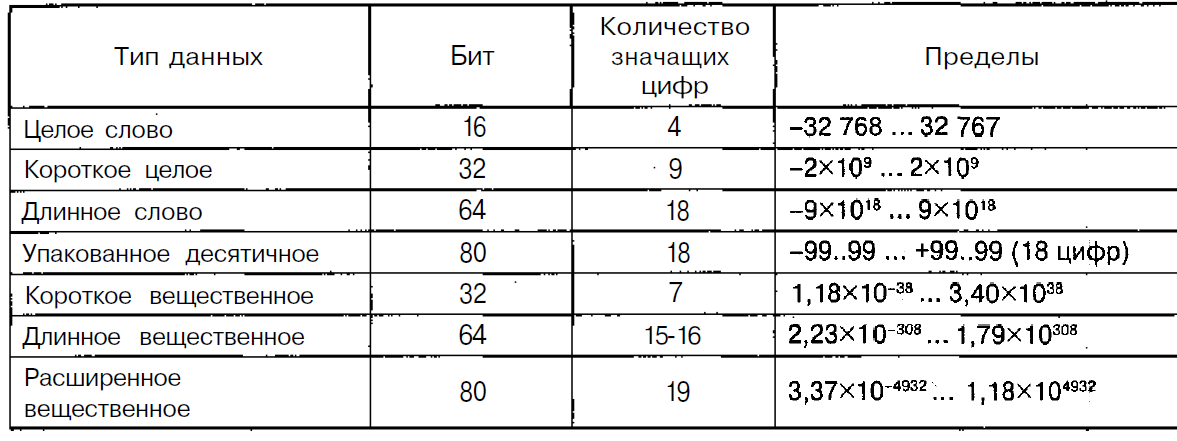
В процессорах Intel все операции с плавающей запятой выполняет специаль-

ное устройство - **FPU (Floating Point Unit)** - с собственными регистрами и набо-

ром команд, поставлявшееся сначала в виде сопроцессора (8087, 80287, 80387,

80487), а начиная с 80486DX - встраивающееся в основной процессор.

Числовой процессор может выполнять операции с над 7-ю типами данных



**Форма представления числа с плавающей запятой в FPU**

* Нормализованная форма представления числа (1,...\*2exp)
* Экспонента увеличена на константу для хранения в положительном виде
* Пример представления 0,625 в коротком вещественном типе:

– 1/4+1/8 = 0,101b

– 1,01b\*2-1

– Бит 31 - знак мантиссы, 30-23 - экспонента, увеличенная на 127, 22-0 - мантисса без первой цифры

– 00111111001000000000000000000000

* Все вычисления FPU - в расширенном 80-битном формате

**Форматы IEEE, применяемые в процессорах Intel**

* короткое вещественное: бит 31 - знак мантиссы, биты 30-23 - 8-битная экспонента +127, биты 22-0 - 23-битная мантисса без первой цифры;
* длинное вещественное: бит 63 - знак мантиссы, биты 62-52 - 11-битная экспонента +1024, биты 51-0 - 52-битная мантисса без первой цифры;
* расширенное вещественное: бит 79 - знак мантиссы, биты 78-64 - 15-битная экспонента +16 383, биты 63-0 - 64-битная мантисса с первой цифрой (то есть бит 63 равен 1).

**FPU выполняет все вычисления в 80-битном расширенном формате, а 32-**

**и 64-битные числа используются для обмена данными с основным процессором**

**и памятью.**

**Особые числа FPU**

* Положительная бесконечность: знаковый - 0, мантисса - нули, экспонента - единицы
* Отрицательная бесконечность: знаковый - 1, мантисса - нули, экспонента - единицы
* NaN (Not a Number): – qNAN (quiet) - при приведении типов/отдельных сравнениях – sNAN (signal) - переполнение в большую/меньшую сторону, прочие ошибочные ситуации
* Денормализованные числа (экспонента = 0): находятся ближе к нулю, чем наименьшее представимое нормальное число

## 

## 25. Математический сопроцессор. Регистры.

FPU предоставляет восемь регистров для хранения данных и пять вспомога-

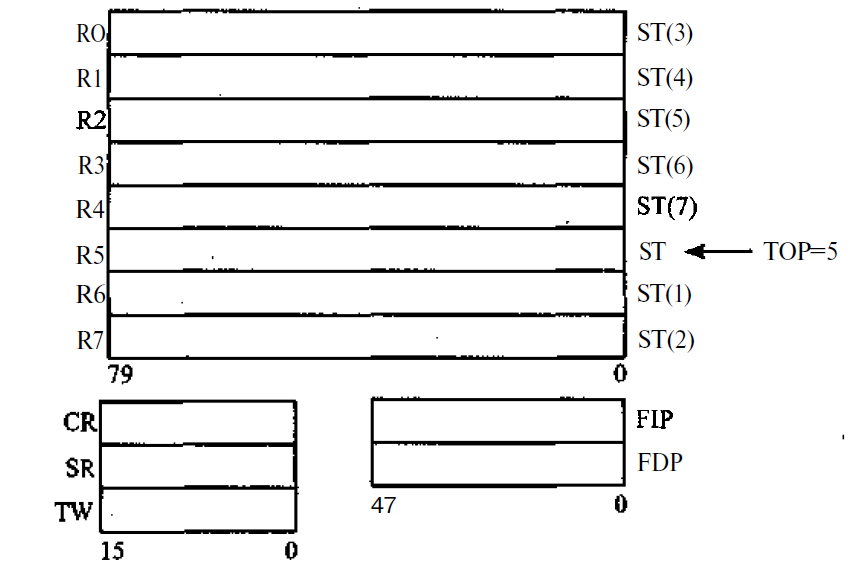
тельных регистров.

**Регистры данных (RO - R7)** не адресуются по именам, а рассматриваются в качестве стека ST. ST соответствует регистру - текущей вершине стека. Более глубокие элементы - ST(1), ST(2) и так далее до ST(7).

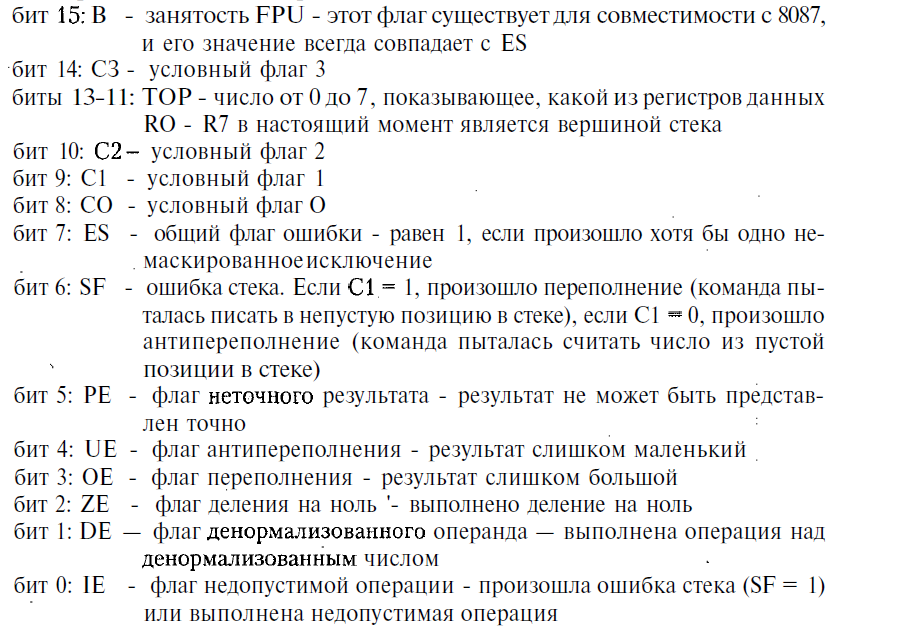
Если, например, в какой-то момент времени регистр R5 называется ST , то после запи-

си в этот стек числа оно будет записано в регистр R4, который станет называться

ST, R5 станет называться ST(1) и т. д.

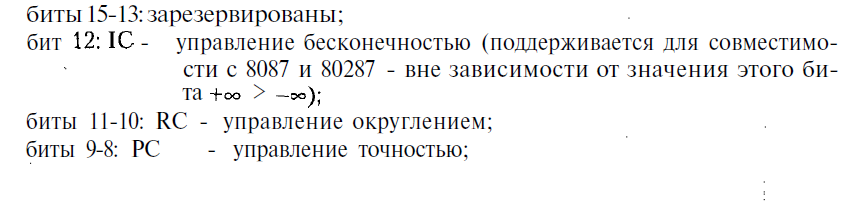


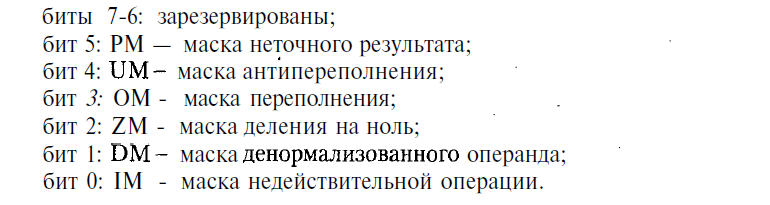
**Регистр состояний SR** - содержит слово состояния FPU. Сигнализирует о различных ошибках, переполнениях



**Регистр управления CR -** Контроль округления, точности.

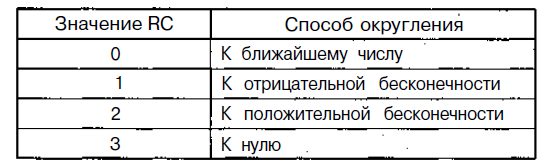
76 76 2130 762130





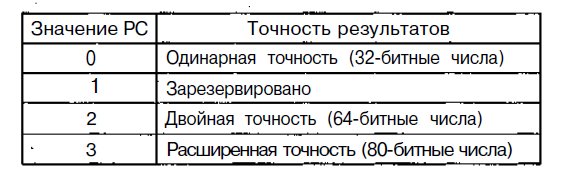
Биты RC определяют способ округления результатов команд FPU до задан-

ной точности



Биты PC определяют точность результатов команд FADD, FSUB, FSUBR,

FMUL, FDIV, FDIVR и FSQRT

****

**Регистр тэгов TW -** 8 пар битов, описывающих состояния регистров данных.

Биты 15-14 описывают регистр R7, 13-12 - R6 и т. д. Если пара битов (тэгов) равна 11, соответствующий регистр пуст. 00 означает, что регистр содержит число, 01 — ноль, 10 — не-число, бесконечность, денормализованное число, неподдерживаемое число.

**Регистры FIP и FDP** - содержат адрес последней выполненной команды (кро-

ме FINIT, FCLEX, FLDCW, FSTCW, FSTSW, FSTSWAX, FSTENV, FLDENV,

FSAVE, FRSTOR и FWAIT) и адрес ее операнда соответственно и используются

в обработчиках исключений для анализа вызвавшей его команды.

**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

## 26. Математический сопроцессор. Классификация команд.

| **Команды пересылки данных** | |
| --- | --- |
| FLD | Загрузить вещественное число из источника (переменная или ST(n)) в стек и уменьшает ТОР на 1. Номер вершины в SR увеличивается |
| FST/FSTP | Скопировать/считать число с вершины стека в приёмник. |
| 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 |
| **Базовая арифметика FPU** | |
| 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 | 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 |
| FSQRT | вычисляет квадратный корень ST(0) |
| **Команды сравнения FPU** | |
| FCOM, FCOMP, FCOMPP | сравнить и вытолкнуть из стека |
| FUCOM, FUCOMP, FUCOMPP | сравнить без учёта порядков и вытолкнуть |
| FICOM, FICOMP, FICOMP | сравнить целые |
| FTST | сравнивает с нулём |
| FXAM | выставляет флаги в соответствии с типом числа |
| **Трансцендентные операции FPU** | |
| FSIN | Вычисляет синус числа, находящегося в ST(0), и сохраняет результат в этом  же регистре. |
| FCOS | Вычисляет косинус числа, находящегося в ST(0), и сохраняет результат в этом  же регистре. |
| FSINCOS | Вычисляет синус и косинус числа, находящегося в ST(0), помещает синус BJST(O),  а затем косинус в стек (так что синус оказывается в ST(1), косинус - в ST(0), и ТОР  уменьшается на 1). |
| FPTAN | Вычисляет тангенс числа, находящегося в регистре ST(0), заменяет его на  вычисленное значение и затем помещает 1 в стек, так что результат оказывается  в ST(1), ST(0) содержит 1, а ТОР уменьшается на единицу. |
| FPATAN | Вычисляет арктангенс числа, получаемого при делении ST(1) на ST(0), сохра-  няет результат в ST(1) и выталкивает ST(0) из стека (помечает ST(0) как пустой  и увеличивает ТОР на 1). |
| **Константы FPU** | |
| FLD1 | Поместить в стек 1,0 |
| FLDZ | Поместить в стек +0,0 |
| FLDPI | Поместить в стек число pi |
| FLDL2E | Поместить в стек Iog2(e) |
| FLDL2T | Поместить в стек log2(10) |
| FLDLN2 | Поместить в стек ln(2) |
| FLDLG2 | Поместить в стек lg(2) |
| **Команды управления FPU** | |
| FINCSTP, FDECSTP | увеличить/уменьшить указатель вершины стека |
| FFREE | освободить регистр |
| FINIT, FNINIT CR и SR по умолчанию | инициализировать сопроцессор / инициализировать без ожидания (очистка данных, инициализация |
| 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 |

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

Начиная с модификации процессора Pentium P54C, все процессоры Intel содер-

жат расширение ММХ, предназначенное для увеличения эффективности про-

грамм, работающих с большими потоками данных (обработка изображений, звука,

видео, синтез), то есть для всех тех случаев, когда нужно выполнить несложные

операции над массивами однотипных чисел.

ММХ предоставляет несколько новых типов данных, регистров и команд, позволяющих осуществлять арифметические и логические операции над несколькими числами одновременно.

**Регистры ММХ**

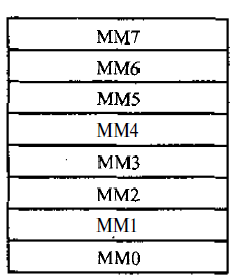
Расширение ММХ включает в себя восемь 64-битных регистров общего поль-

зования ММО - ММ7. Физически никаких новых регистров с введением ММХ не появилось, ММО - ММ7 - это в точности мантиссы восьми регистров FPU, от RO до R7.При записи числа в регистр ММХ оно оказывается в битах 63-0 соответствующего регистра FPU, а экспонента (биты 78-64) и ее знаковый бит (бит 79) заполняются единицами.

Запись числа в регистр FPU также приводит к изменению соответствующего регистра ММХ. Любая команда ММХ, кроме EMMS, приводит к тому, что поле ТОР регистра SR и весь регистр TW в FPU обнуляются. Команда EMMS заполняет ре-

гистр TW единицами.

Таким образом, нельзя одновременно пользоваться командами FPU и командами ММХ, а если это необходимо - следует применять команды FSAVE/FRSTOR каждый раз перед переходом от FPU к ММХ и обратно (эти команды сохраняют состояние регистров ММХ точно так же, как и FPU).



**Типы данных ММХ**

ММХ использует четыре новых типа данных:

1. учетверенное слово — простое 64-битное число;
2. упакованные двойные слова - два 32-битных двойных слова, упакованные

в 64-битный тип данных.

Двойное слово 1 занимает биты 63—32, и двойное слово 0 - биты 31-0;

1. упакованные слова - четыре 16-битных слова, упакованные в 64-битный тип

данных.

Слово 3 занимает биты 63-48, слово 0 - биты 15-0;

1. упакованные байты - восемь байт, упакованных в 64-битный тип данных.

Байт 7 занимает биты 63-56, байт 0 - биты 7-0

Команды ММХ перемещают упакованные данные в память или в обычные

регистры как целое, но выполняют арифметические и логические операции над

каждым элементом по отдельности.

Арифметические операции в ММХ могут использовать специальный способ

обработки переполнений и антипереполнений - **насыщение.**

Если результат операции больше, чем максимальное значение для его типа данных (+127 для байта со знаком), то результат подразумевают равным этому максимальному значению.

Если он меньше минимального значения - соответственно его считают равным

минимально допустимому значению.

*Например, при операциях с цветом насыщение позволяет ему превращаться в чисто белый при переполнении и в чисто черный при антипереполнении, в то время как обычная арифметика привела бы к нежелательной инверсии цвета.*

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

1. **Команды пересылки данных** **MMX**

* 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
* PACKUSWB - упаковка слов в байты с беззнаковым насыщением
* PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ - распаковка и объединение старших элементов источника и приёмника через 1

1. **Арифметические операции MMX**

* PADDB, PADDW, PADDD - поэлементное сложение, перенос игнорируется
* PADDSB, PADDSW - сложение с насыщением
* 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)
* PMADDWD - умножение и сложение. Перемножает 4 слова, затем попарно складывает произведения двух старших и двух младших

1. **Команды сравнения MMX**

* 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

1. **Логические операции MMX**

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

1. **Сдвиговые операции MMX**

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

## 29. Расширения процессора. SSE. Регистры, поддерживаемые типы данных.

SSE(Streaming SIMD Extensions - потоковые SIMD-расширения), где SIMD (Single

Instruction - Multiple Data) - общий для SSE и ММХ подход к обработке боль-

шого количества данных одной командой.

**Использование:**

1. Современные приложения, которые работают с двумерной и трехмерной графикой
2. видео-, аудио- и другие виды потоковых данных

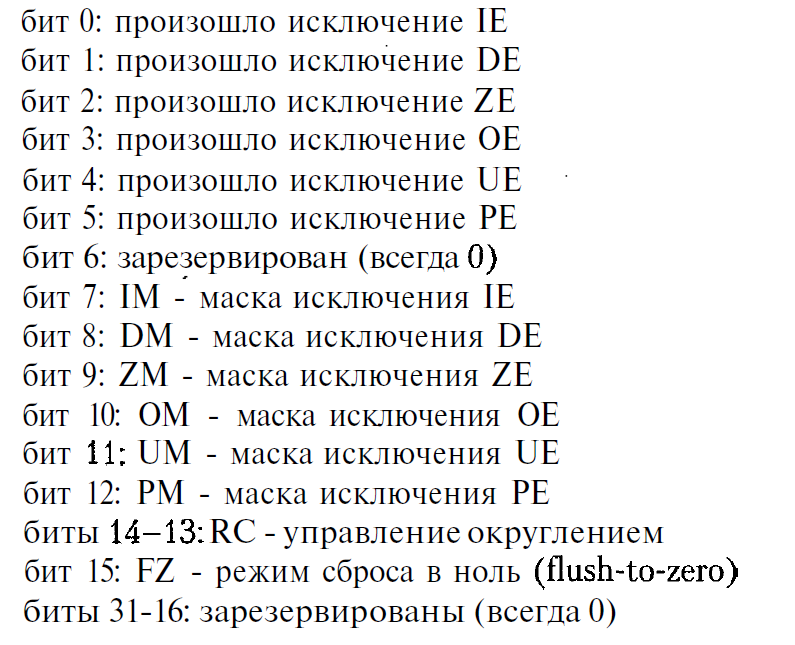
В отличие от MMX, данное расширение не использует существующие ресурсы процессора, оно вводит **8 новых независимых 128-битных регистров данных (XMMO - XMM7)**

Благодаря этому решаются проблемы MMX - не требуется команд типа EMMS для переключения режимов и можно пользоваться другими расширениями, работая

с SSE.

Также вводится дополнительный 32-битный регистр управления/состояния **MXCSR**. Используется для

1. Маскировка исключений
2. Выбор режимов
3. Определение состояния флагов



Все маскирующие биты по умолчанию 1 => никакие исключения не обрабатываются.

**Режимы округления RC**

00 - к ближайшему числу (по умолчанию)

01 - к отрицательной бесконечности

10 - к положительной бесконечности

11 - к нулю.

**Режим сброса в ноль FZ (включен по умолчанию)** - команды SSE не превращают слишком маленькое число с плавающей запятой в денормализованное, а возвращает 0. Знак нуля соответствует знаку получившегося бы денормализованного числа,

и, кроме того, устанавливаются флаги РЕ и UE.

**Типы данных SSE**

Основной тип данных - **упакованные числа с плавающей запятой одинарной точности (32 бита)**.

В одном 128-битном регистре размещаются сразу 4 таких числа

в битах 127-96 (число 3)

95-64 (число 2)

63-32 (число 1)

31-0 (число 0)

Это стандартные 32-битные числа с плавающей запятой, используемые числовым сопроцессором.

Целочисленные команды SSE могут работать с упакованными байтами, словами или двойными словами. Однако эти команды оперируют данными, находящимися в регистрах ММХ.

## 30. Расширения процессора. SSE. Классификация команд.

**Команды SSE**

Все команды SSE доступны из любых режимов процессора — реального, защищенного и режима V86.

**Классификация команд**

1. **Команды пересылки данных**

* Начинаются с MOV…
* Возможные пересылки:

1. Выравненные упакованные числа
2. Невыравненные упакованные числа
3. Старшие упакованные числа
4. Младшие упакованные числа
5. Старшие упакованные числа в младшие
6. Младшие упакованные числа в старшием
7. Маску в переменную
8. Одно вещественное число

По сути различаются количеством копируемых бит, местом копирования (выше), возможными значениями приемника/источника (регистр SSE, переменная в памяти)

1. **Арифметические**

* ADDPS - Сложение упакованных вещественных чисел (Выполняет параллельное сложение четырех пар чисел с плавающей запятой,

находящихся в источнике (переменная или регистр SSE) и приемнике (регистр SSE). Результат записывается в приемник) ADDSS - Сложение одного вещественного числ (Выполняет сложение нулевых (занимающих биты 31-0) чисел с плавающей запятой в источнике (переменная или регистр SSE) и приемнике (регистр SSE). Результат записывается в биты 31-0 приемника, биты 127-32 остаются без изменений.)

* Аналогично SUBSS, SUBPS, MULPS, MULSS, DIVPS, DIVSS, SQRTPS, SQRTSS
* RCPPS - Обратная величина для упакованных чисел. RCPSS - Обратная величина для одного числа
* RSQRTPS - Обратный корень из упакованных чисел. RSQRTSS - Обратный корень из одного числа
* MAXPS, MAXSS, MINPS, MINSS

1. **Сравнения**

* CMPPS - Сравнение упакованных вещественных чисел. CMPSS - Сравнение одной пары упакованных чисел
* COMISS- Сравнение одной пары чисел с установкой флагов.UCOMISS - Сравнение одной пары неупорядоченных чисел с установкой флагов

1. **Преобразования типов**

* CVTPI2PS - Упакованные целые в вещественные
* CVTPS2PI - Упакованные вещественные в целые
* CVTSI2SS- Целок в вещественное
* CVTSS2SI - Вещественное в целое
* CVTTPS2PI - Вещественных в целые с обрезанием
* CVTTSS2SI - Преобразование вещественного в целое с обрезанием

1. **Логические**

* ANDPS - И
* ANDNPS - НЕ-И
* ORPS - ИЛИ
* XORPS - Исключающее ИЛИ

1. **Целочисленные SIMD-команды**

Помимо расширения для работы с упакованными вещественными числами в SSE входит расширение набора команд для работы с упакованными целыми числами, которые размещаются в регистрах ММХ.

* PEXTRW - Распаковать одно слово
* PINSRW - Запаковать одно слово

и т.д.

1. **Упаковки**

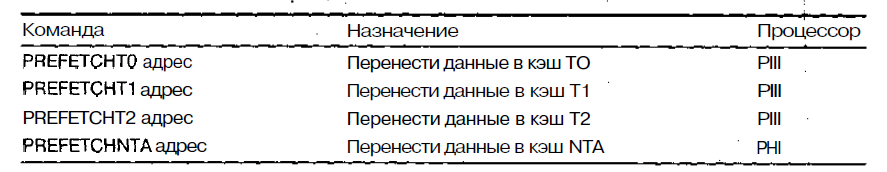
* SHUFPS - Переставить упакованные вещественные
* UNPCKHPS - Распаковать старшие вещественные числа
* UNPCKLPS - Распаковать младшие вещественные числа

1. **Управления состояние**

* LDMXCSR - Загрузить регистр MXCSR (Помещает значение источника (32-битная переменная) в регистр управления и состояния SSE MXCSR)
* STMXCSR - Сохранить регистр MXCSR (Помещает значение регистра MXCSR в приемник (32-битная переменная).
* FXSAVE - Сохранить состояние FPU, MMX, SSE (Сохраняет содержимое всех регистров FPU, ММХ и SSE в приемнике (512- байтовая область памяти).
* FXRSTOR - Восстановить состояние FPU, MMX, SSE

1. **Управления кэшированием**

* MASKMOVQ - Запись байтов минуя кэш
* MOVNTQ - Запись 64 бит минуя кэш (Содержимое источника (регистр ММХ) записывается в приемник (64-битная переменная в памяти), сводя к минимуму загрязнение кэша)
* MOVNTPS - Запись 128 бит минуя кэш
* SFENCE - Защита записи





## Определения и возможные доп. вопросы:

1. **Что такое машинное слово?**

***Машинное слово*** — машинно- зависимая величина, измеряемая в битах, равная разрядности регистров/шины данных.

на практике:

Машинное слово (word) — это количество данных, которые процессор может обработать за одну операцию.

1. **Какие шины существуют?**

Шина адреса, шина данных, шина управления.

1. **Что такое неупакованное двоично-десятичное число?**

Десятичная цифра, хранящаяся в байте.

1. **Что такое упакованное двоично-десятичное число?**

Две десятичные цифры, хранящиеся в полубайтах одного байта

1. **Какой максимальный размер сегмента?**

Максимальный размер сегмента 2^16 байт (64 кбайта) == 2 в степени разрядности шины данных.

1. **Насколько современный компьютер приближен к архитектуре Фон Неймана?**

Ну можно что-то сказать про апдейт в области памяти: добавились кэши, блок регистров (про регистры даже кузнецов сам говорил).

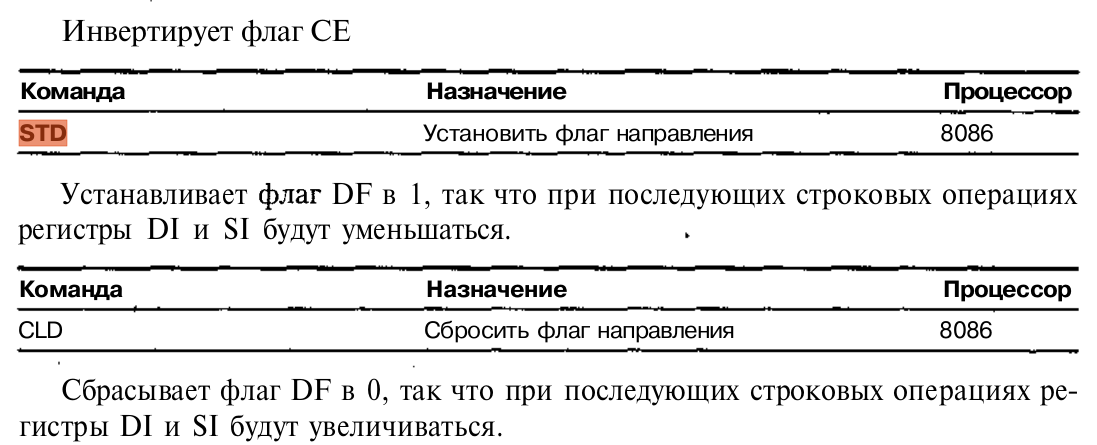
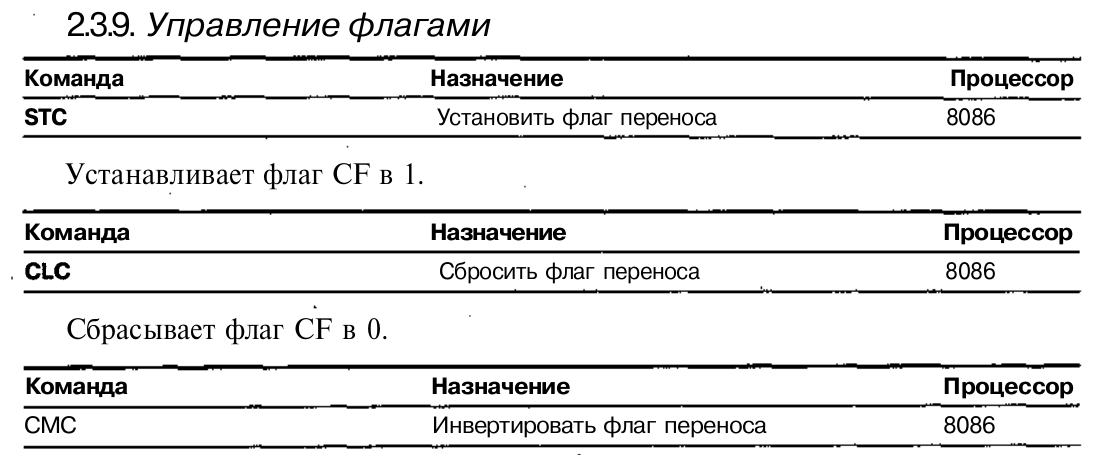
1. **Где можно использовать xlat и какой у команды доп параметр?**

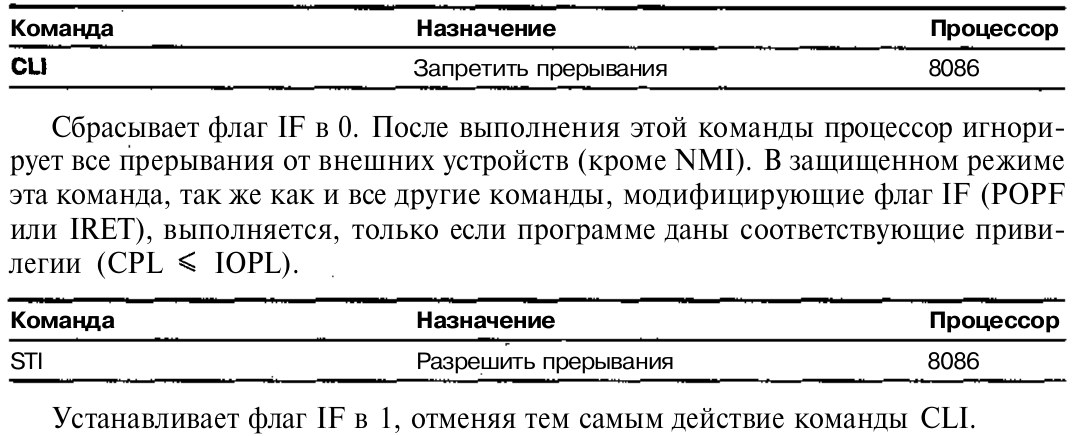
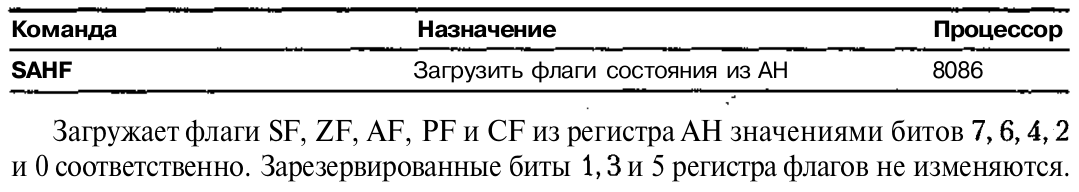
XLAT можно использовать для трансляции таблицы.

Операнд команды XLAT служит только для указания **сегмента** расположения таблицы трансляции. Если операнд не указан, будет использовано значение сегментного регистра **DS**. Тот же результат будет получен при использовании синтаксиса **XLATB**.

**Например**, можно написать преобразование шестнадцатеричного числа в ASCII-код соответствующего ему символа.

1. **Какими командами можно установить флаги в регистре флагов?**





. Также, битовые инструкции (BT, BTS, BTR и BTC) копируют указанный бит во флаг CF.

1. **Зачем нужны флаги состояний, флаги управления, служебные?**

**Флаги состояния**

* отражают результат выполнения арифметических инструкций, таких как ADD, SUB, MUL, DIV.
* позволяют одной и той же арифметической инструкции выдавать результат трёх различных типов
  + **беззнаковое:** **CF** показывает условие переполнения (перенос или заём), **ZF** отражает нулевой результат.
  + **знаковое** (доп. код) - **OF** показываем перенос или заём, **SF** отражает знак, **ZF** нулевой результат.
  + **двоично-десятично кодированное (BCD)** целое число - перенос/заём показывает флаг **AF**.
* Инструкции условного перехода **J*cc***, **SET*cc***, **LOOP*cc***, **CMOV*cc*** используют один или несколько флагов состояния для проверки условия. Например, инструкция перехода JLE (jump if less or equal — переход, если «меньше или равен», ≤) проверяет условие «ZF=1 или SF ≠ OF».

**Управляющий флаг (DF - флаг направления)**

* Управляет строковыми инструкциями *(MOVS, CMPS, SCAS, LODS и STOS)*:
  + установка флага заставляет **уменьшать** адреса (обрабатывать строки от старших адресов к младшим)
  + обнуление заставляет адреса **увеличивать**.

*Инструкции STD и CLD соответственно устанавливают и обнуляют флаг DF.*

**Системные флаги**

*Системные флаги и поле IOPL управляют операционной средой и не предназначены для использования в прикладных программах.*

* **IF** — обнуление этого флага запрещает отвечать на [маскируемые](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B5%D1%80%D1%8B%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5#%D0%9C%D0%B0%D1%81%D0%BA%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) запросы на прерывание.
* **TF** — установка этого флага разрешает пошаговый режим [отладки](https://ru.wikipedia.org/wiki/%D0%9E%D1%82%D0%BB%D0%B0%D0%B4%D0%BA%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D1%8B), когда после каждой выполненной инструкции происходит прерывание программы и вызов специального обработчика прерывания (см. также: [Int3](https://ru.wikipedia.org/wiki/Int3)).

Про эти ДимСаныч ничего не говорил, но на всякий случай вот:

* *IOPL — показывает уровень приоритета ввода-вывода исполняемой программы или задачи: чтобы программа или задача могла выполнять инструкции ввода-вывода или менять флаг IF, её текущий уровень приоритета (CPL) должен быть ≤ IOPL.*
* *NT — этот флаг устанавливается, когда текущая задача «вложена» в другую, прерванную задачу, и сегмент состояния TSS текущей задачи обеспечивает обратную связь с TSS предыдущей задачи. Флаг NT проверяется инструкцией IRET для определения типа возврата — межзадачного или внутризадачного.*
* *RF — флаг маскирования ошибок отладки.*
* *VM — установка этого флага в* [*защищённом режиме*](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D1%89%D0%B8%D1%89%D1%91%D0%BD%D0%BD%D1%8B%D0%B9_%D1%80%D0%B5%D0%B6%D0%B8%D0%BC) *вызывает переключение в* [*режим виртуального 8086*](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B6%D0%B8%D0%BC_%D0%B2%D0%B8%D1%80%D1%82%D1%83%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B3%D0%BE_8086)*.*
* *AC — установка этого флага вместо с битом AM в регистре CR0 включает контроль* [*выравнивания*](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%80%D0%B0%D0%B2%D0%BD%D0%B8%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85) *операндов при обращениях к памяти — обращение к невыравненному операнду вызывает исключительную ситуацию.*
* *VIF — виртуальная копия флага IF; используется совместно с флагом VIP.*
* *VIP — устанавливается для указания наличия отложенного прерывания; используется совместно с флагом VIF.*
* *ID — возможность программно изменить этот флаг в регистре флагов указывает на поддержку инструкции* [*CPUID*](https://ru.wikipedia.org/wiki/Cpuid)*.*[*[2]*](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B3%D0%B8%D1%81%D1%82%D1%80_%D1%84%D0%BB%D0%B0%D0%B3%D0%BE%D0%B2#cite_note-CPUID-2)

1. **Каким образом в FPU хранятся вещественные числа?**

Числа с плавающей точкой хранятся как знак + порядок(экспонента) + мантисса.

1. **Флаги DF, IF. Зачем делать флаг IF = 0(выключать обработку прерываний)?**

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

*IF (interrupt enable flag)* - флаг разрешения прерываний - если 0 процессор перестает обрабатывать прерывания от внешних устройств(аппаратные/асинхронные)

Зачем выключать обработку прерываний?

* когда у нас есть критический участок кода, который нам нужно выполнить, но мы знаем, что там будет прерывание - мы выключаем его обработку.
* когда нужно изменить обработчик прерываний, необходимо выключить обработку, чтобы случайно не выполнилось прерывание, пока идёт “перенастройка”

1. **Почему нельзя использовать в командах пересылки данных 2 операнда типа память?**

системная шина (которая в процессоре одна) состоит из 3 шин: шина управления, шина данных, шина адреса, на шине управления выставляется команда получения/записи данных, на шину адреса - адрес и по шине данных они пересылаются, а одновременно послать два сигнала по ним невозможно)

1. **Почему команды обработки строк так называются, если они работают с байтами/словами?**

Строка - последовательность байт, оканчивающаяся \0.

Используя префиксы rep/repne и тд, можно проанализировать/обработать строку длины cx.

Несмотря на то, что цепочечные команды имеют отношение к одному байту или одному слову, **префикс REP** обеспечивает повторение команды несколько раз. Префикс кодируется непосредственно перед цепочечной командой, например, **REP MOVSB.** Для использования префикса REP необходимо установить начальное значение в регистре **CX**. При выполнении цепочечной команды с префиксом REP происходит уменьшение на 1 значения в регистре CX до нуля.

**Таким образом, можно обрабатывать строки любой длины**.

**Подробнее**

*Для автоматической обработки строк всеми вышеперечисленными командами используются префиксы повторения:* ***REP, REPE/REPZ*** *и* ***REPNE/REPNZ****. Перед использованием команды с префиксом повторения необходимо поместить в CX/ECX число повторений команды.* ***Префикс повторения автоматически уменьшает регистр CX/ECX и повторяет выполняемую команду до тех пор, пока CX/ECX не будет равен нулю****. REPE/REPZ и REPNE/REPNZ прекращают повторение команды не только при CX/ECX=0, но и в зависимости от состояния флага ZF: REPE/REPZ может повторяться лишь до тех пор, пока ZF установлен, а REPNE/REPNZ — пока ZF=0. Повторяемая команда обработки строк будет обрабатывать каждый раз следующий элемент строки.*

1. **Что, помимо действия над байтами/словами делают команды обработки строк?**

Изменяют регистры di,si в зависимости от флага df.

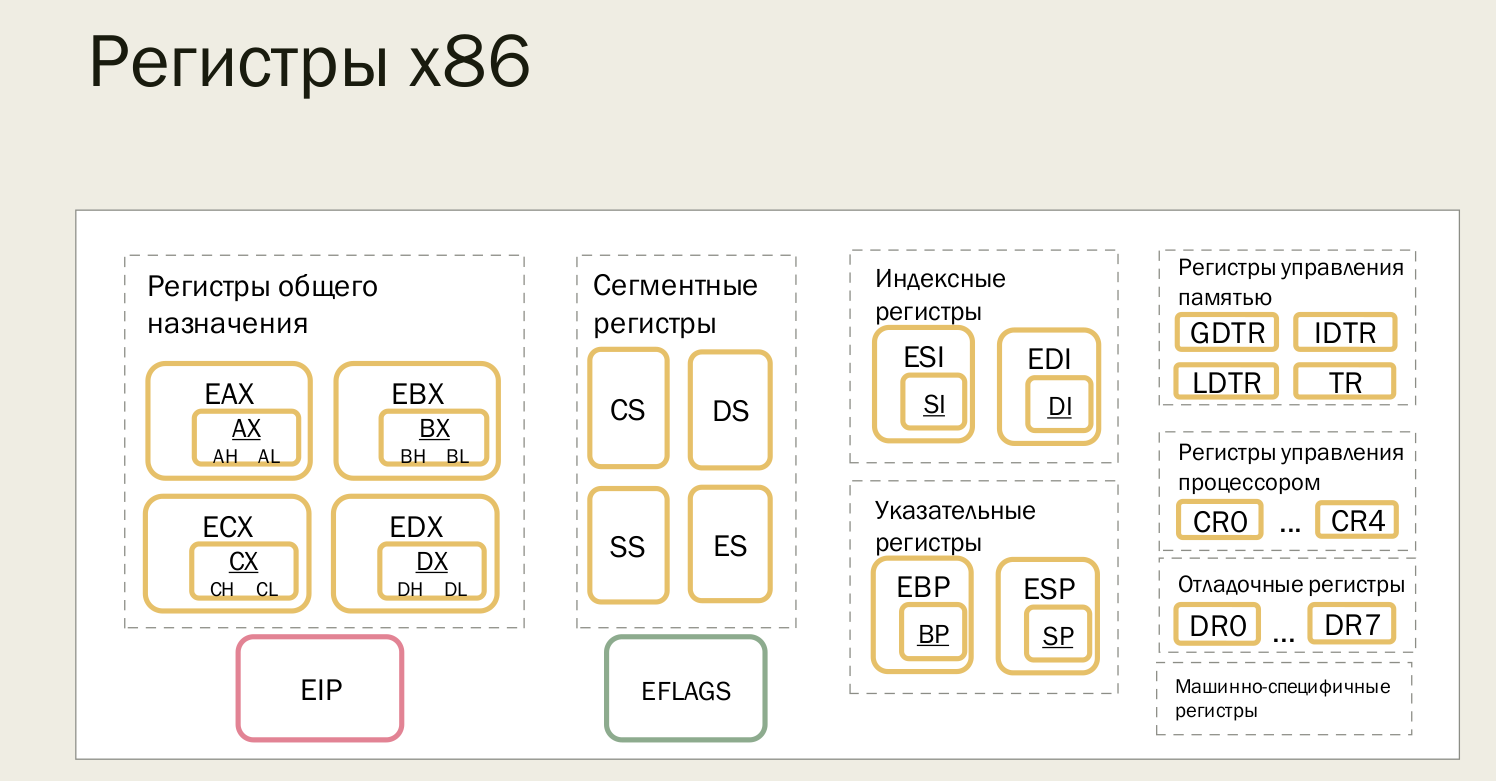
Например, команда:

● **MOVS/MOVSB/MOVSW** <приёмник>, <источник> - копирует байт или слово из приемника в источник. После выполнения команды, регистры SI и DI увеличиваются на 1 (или 2), если флаг DF = 0, или уменьшаются на 1 (или 2), если DF = 1

1. **Какие регистры общего назначения в 32/64 разрядных процессорах. Какие отличия от архитектуры 8086?**

В 32/64 разрядных процессорах появились 32х/64 битные регистры, младшие части которых остались 16 битными как в 8086.

* x64 расширяет 8 **регистров общего назначения** до 64-бит и добавляет 8 новых 64-bit регистра.
* Регистры 64-bit имеют имена, начинающиеся с "r", поэтому , например, расширение 64-bit eax называется rax.
* Новые регистры называются r8-R15.
* младшие 32 бита, 16 бит и 8 бит каждого регистра адресуются в операндах. Даже **ESI,** чьи нижние 8 бит ранее не были адресуемы.



1. **Подробно рассказать про ОЗУ/ПЗУ.**

Память делится на внешнюю и внутреннюю

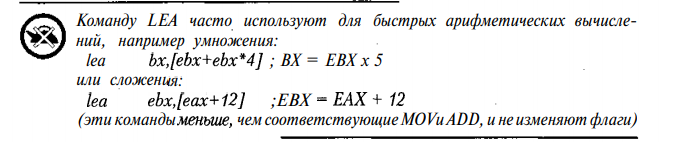
К внутренней памяти относится:

* ОЗУ (оперативное запоминающее устройство)
* ПЗУ (постоянное запоминающее устройство). В ПЗУ хранится информация, которая записывается туда при изготовлении ЭВМ. Важнейшая микросхема ПЗУ - BIOS.

*ОЗУ* является энергозависимой памятью. Это означает, что информация, временно хранящаяся в модуле, стирается при перезагрузке или выключении компьютера. Поскольку информация хранится на транзисторах с электрическим питанием, при отключении электрического тока данные исчезают. Каждый раз, когда вы запрашиваете файл или информацию, они извлекаются из хранилища компьютера или из Интернета. Данные хранятся в ОЗУ, поэтому каждый раз при переключении из одной программы или страницы на другую, информация моментально становится доступной. Когда компьютер выключается, память очищается, пока процесс не начнется снова. Пользователи могут легко изменять, модернизировать или расширять энергозависимую память.

*ПЗУ* является энергонезависимой памятью. Это означает, что информация хранится на микросхеме постоянно. Для хранения данных этой памяти не нужно электропитание, данные записываются в отдельные ячейки памяти в форме двоичного кода. *(Данные записываются в ПЗУ в процессе производства. Для этого изготавливается трафарет с определенным набором битов, который накладывается на фоточувствительный материал, а затем открытые (или закрытые) части поверхности вытравливаются. Единственный способ изменить программу в ПЗУ — поменять всю микросхему.)* **Энергонезависимая память используется для неизменяемых компонентов компьютера, например, для загрузочной части программного обеспечения** или служебных команд встроенного ПО, которые запускают принтер. Выключение компьютера никак не влияет на ПЗУ. Пользователи не могут изменять энергонезависимую память.

1. **Какие математические операции можно заменить с помощью lea.**



**ПЛЮШКИ ОТ LEA:**

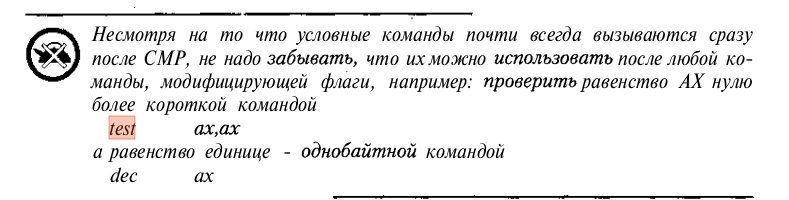
1. возможность выполнять сложение с **двумя** или **тремя операндами**
2. возможность хранить результат в любом регистре, а не только в одном из исходных операндов.

**Следствия:**

1. **LEA EAX, [ EAX + EBX + 1234567 ] - три операнда**
2. **LEA EAX, [ EBX + ECX ] - сохраняем результат операции в регистр, отличный от операнда**
3. **LEA EAX, [ EBX + N \* EBX ] - вот такое умножение на константу, только N должно быть степенью двойки**
4. **Как называются все флаги, изменяемые cmp?**

CF, PF, AF, ZF, SF, OF - флаги, которые изменяет cmp. Это флаги **состояния**.

1. **Какие ещё команды, изменяющие флаги и используемые для сравнения существуют? Как они это делают?**

****

1. **С помощью каких команд можно работать с отдельными битами?**

Операции над битами и байтами BT, BTR, BTS, BTC, BSF, BSR, SETcc

*База - регистр или переменная.*

*Смещение - от 0 до 15/31 в зависимости от размера регистра(базы), иначе берётся остаток от деления на 16/32*

● **BT** <база>, <смещение> - считать в CF значение бита из битовой строки

Пример

bt ax,0 ;Проверка младшего бита AX

jc m1 ;Переход, если бит равен 1

mov cx,3 ;CX=3

bt ax,cx ;Проверка 3-го бита AX

jnc m1 ;Переход, если бит равен 0

● **BTS** <база>, <смещение> - установить бит в 1

● **BTR** <база>, <смещение> - сбросить бит в 0

● **BTC** <база>, <смещение> - инвертировать бит

● **BSF** <приёмник>, <источник> - прямой поиск бита (от младшего разряда)

● **BSR** <приёмник>, <источник> - обратный поиск бита (от старшего разряда)

*Сканирование битов выполняется командами BSF и BSR. Эти команды очень похожи. У них 2 операнда. Первый операнд должен быть 16-битным регистром, в него записывается результат. Второй операнд может быть 16-битным регистром или словом в памяти — это обрабатываемое значение.*

*Команда BSF просматривает биты второго операнда от младшего к старшему и помещает индекс первого единичного бита в регистр. Биты нумеруются, начиная с нуля. Если единичный бит найден, то флаг нуля сбрасывается (ZF=0). Если все биты нулевые, то флаг нуля устанавливается (ZF=1), а значение первого операнда будет неопределённым (на разных процессорах может быть по-разному).*

Пример:

mov ax,01011000b ;AX=58h

bsf bx,ax ;BX=3, ZF=0

xor ax,ax ;AX=0

bsf bx,ax ;BX=?, ZF=1

● **SETcc** <приёмник> - выставляет приёмник (1 байт) в 1 или 0 в зависимости от условия, аналогично Jcc

1. **Сколько максимум сегментов кода может быть в программе? Какого размера максимум может быть программа?**

Ограничений на количество сегментов кода нет.

.COM файл - размер файла < 64 Kb

.EXE - размер программы ограничен количеством памяти компьютера.(внутренняя + внешняя??!?!!?!!)ы

1. **Расскажите про прерывания в реальном режиме работы и в защищенном.**

*Реальный режим*

Процессор использует таблицу векторов прерываний. Эта таблица располагается в самом начале оперативной памяти, т.е. её физический адрес - 00000.

Таблица векторов прерываний реального режима состоит из 256 элементов по 4 байта, её размер 1 КБ. Элементы таблицы - дальние указатели на процедуры обработки прерываний. Указатели состоят из 16-битового сегментного адреса процедуры обработки прерывания и 16-битового смещения. Причём смещение хранится по младшему адресу, а сегментный адрес - по старшему.

Когда происходит программное или аппаратное прерывание, текущее содержимое регистров CS, IP а также регистра флагов FLAGS записывается в стек программы (который, в свою очередь, адресуется регистровой парой SS:SP). Далее из таблицы векторов прерываний выбираются новые значения для CS и IP, при этом управление передаётся на процедуру обработки прерывания.

Перед входом в процедуру обработки прерывания принудительно сбрасываются флажки трассировки TF и разрешения прерываний IF. Поэтому если ваша процедура прерывания сама должна быть прерываемой, вам необходимо разрешить прерывания командой STI. В противном случае, до завершения процедуры обработки прерывания все прерывания будут запрещены.

Завершив обработку прерывания, процедура должна выдать команду IRET, по которой из стека будут извлечены значения для CS, IP, FLAGS и загружены в соответствующие регистры. Далее выполнение прерванной программы будет продолжено.

Что же касается аппаратных маскируемых прерываний, то в компьютере IBM AT и совместимых с ним существует всего шестнадцать таких прерываний, обозначаемых IRQ0-IRQ15. В реальном режиме для обработки прерываний IRQ0-IRQ7 используются вектора прерываний от 08h до 0Fh, а для IRQ8-IRQ15 - от 70h до 77h.

*Защищённый режим*

В защищённом режиме все прерывания разделяются на два типа - *обычные прерывания и исключения.*

Обычное прерывание инициируется командой INT (программное прерывание) или внешним событием (аппаратное прерывание). Перед передачей управления процедуре обработки обычного прерывания флаг разрешения прерываний IF сбрасывается и прерывания запрещаются.

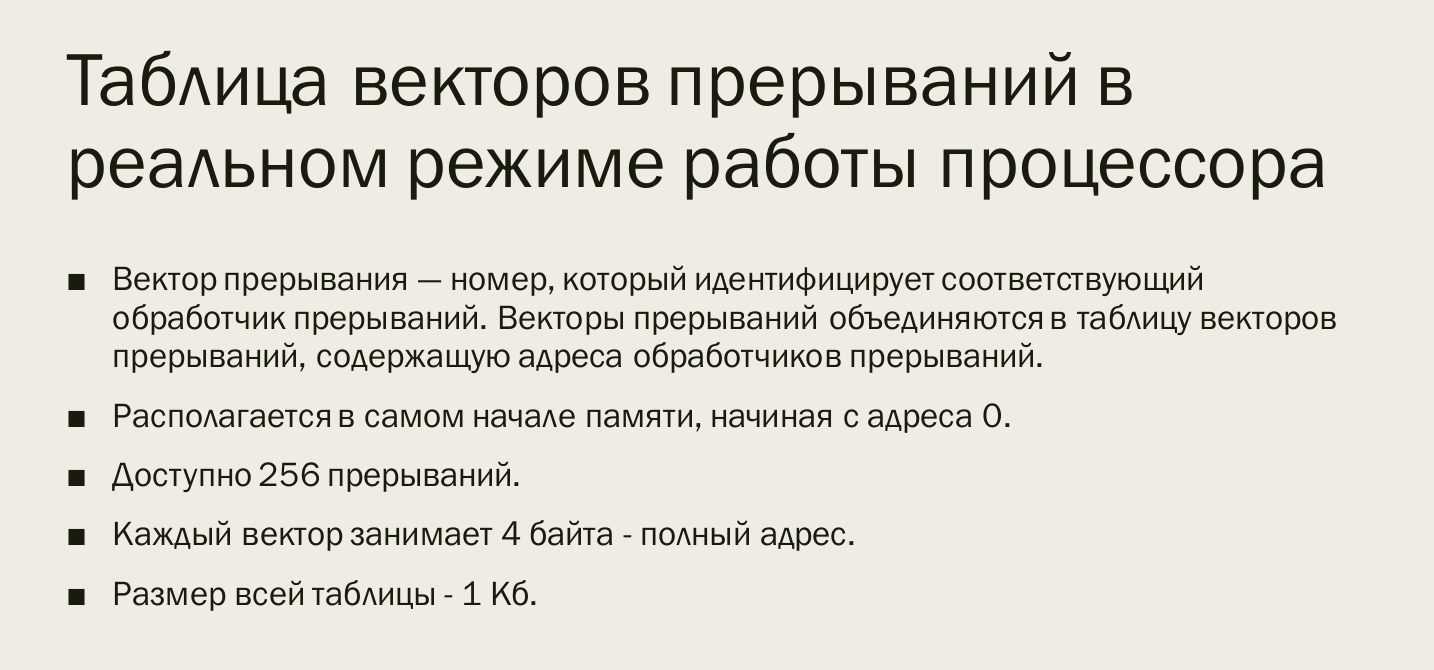
Исключение происходит в результате ошибки, возникающей при выполнении какой-либо команды, *например*, *если команда пытается выполнить запись данных за пределами сегмента данных или использует для адресации селектор, который не определён в таблице дескрипторов*. По своим функциям исключения соответствуют зарезервированным для процессора внутренним прерываниям реального режима. Когда процедура обработки исключения получает управление, флаг IF не изменяется. Поэтому в мультизадачной среде особые случаи, возникающие в отдельных задачах, не оказывают влияния на выполнение остальных задач.

*В защищённом режиме прерывания могут приводить к переключению задач.*

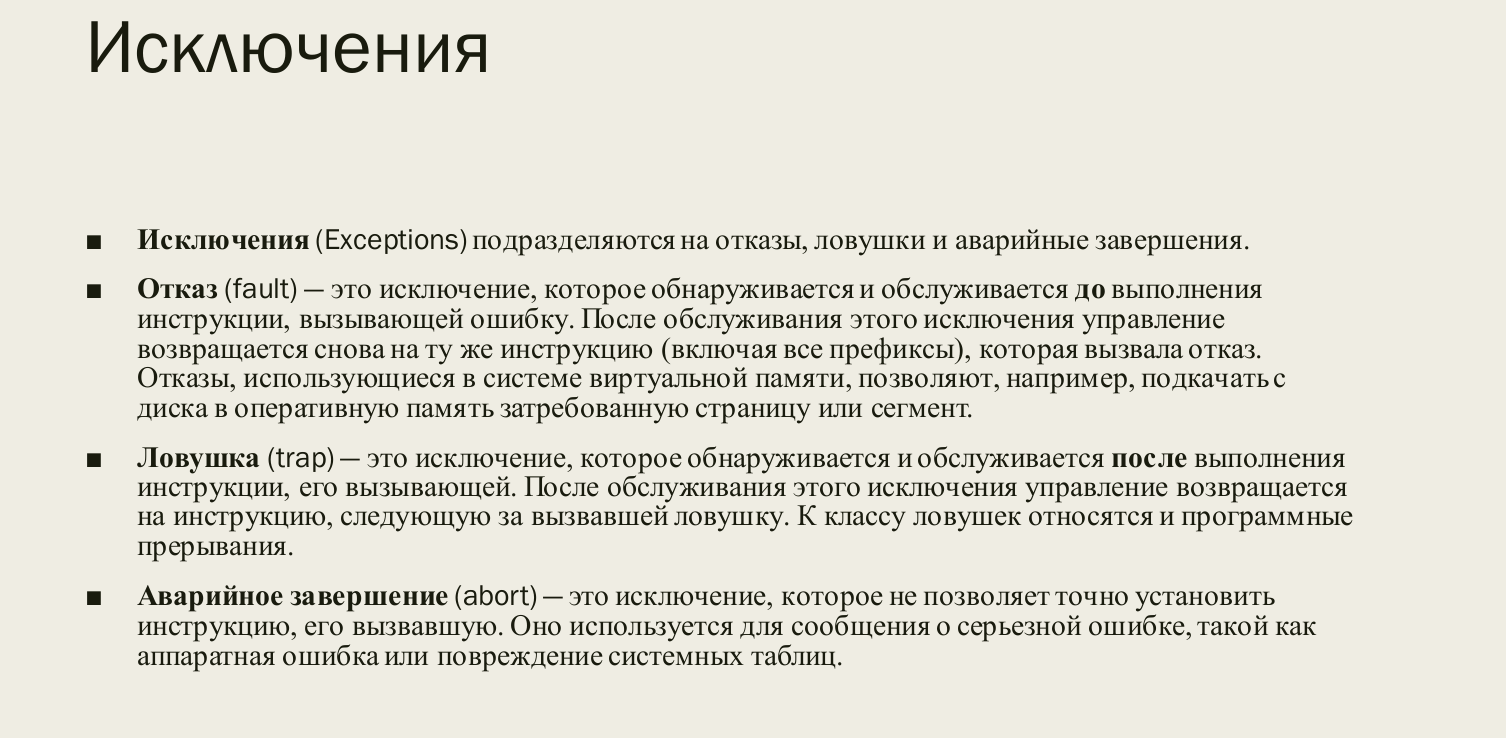
*ПОДРОБНЕЕ ТУТЬ:* [*https://frolov-lib.ru/books/bsp.old/v06/ch3.htm*](https://frolov-lib.ru/books/bsp.old/v06/ch3.htm)

1. **Где располагается таблица векторов прерываний?**

Находится в начале памяти с адреса 0. Занимает 1024 байта. Размер вектора 4 байта.



1. **Исключения. (** https://clck.ru/VckBy )

****

**Цитаты ДМК**

**Исключения** - можно сказать, аналог синхронных программных прерываний, которые возникают при различных ситуациях в процессе работы программ на процессоре.

То есть это могут быть отказы, ловушки и аварийные завершения.

**Пример отказа:** обращение идет к адресу,который в память не загружен. Соответственно обрабатывается отказ, подгружается с диска страница и команда продолжает выполняться

**Пример ловушки:** происходит деление на ноль, это вызывает ловушку. Эта ситуация обрабатывается и если она корректно обработана, то выполнение программы продолжается со следующей команды

То есть исключения - это механизм подобный программным прерываниям реального режима