# [Архитектура фон Неймана. Принципы фон Неймана](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-01%5D-%D0%90%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0-%D1%84%D0%BE%D0%BD-%D0%9D%D0%B5%D0%B9%D0%BC%D0%B0%D0%BD%D0%B0.-%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF%D1%8B-%D1%84%D0%BE%D0%BD-%D0%9D%D0%B5%D0%B9%D0%BC%D0%B0%D0%BD%D0%B0)

## **Архитектура фон Неймана**

****

* Процессор состоит из блоков УУ и АЛУ
* УУ - дискретный конечный автомат.  
  Структурно состоит из: дешифратора команд (операций), регистра команд, узла вычислений текущего исполнительного адреса, счётчика команд (регистр IP).
* АЛУ - под управлением УУ производит преобразование над данными (операндами). Разрядность операнда - длина машинного слова. (Машинное слово - машинно-зависимая величина, измеряемая в битах, равная разрядности регистров/шины данных)

## **Принципы фон Неймана**

1. Использование двоичной с/с в вычислительных машинах.

2. Программное управление ЭВМ.

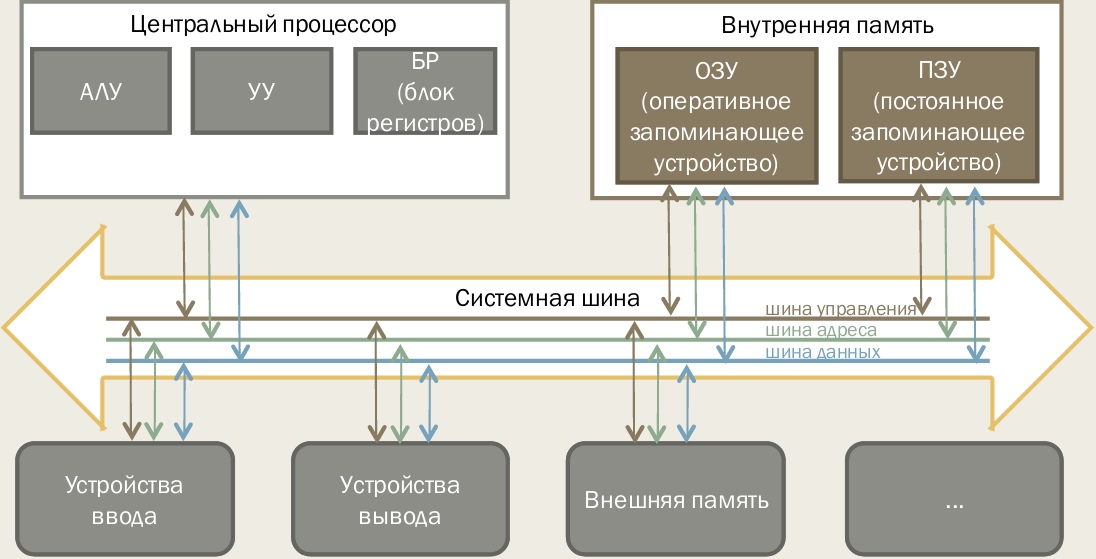
3. Принцип однородности памяти. Память используется не только для хранения данных, но и для программ.

4. Ячейки памяти ЭВМ имеют адреса, которые последовательно пронумерованны.

5. Возможность условного перехода в процессе выполнения программы.

# [Структурная схема ЭВМ. Виды памяти. Системная шина.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-02%5D-%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%BD%D0%B0%D1%8F-%D1%81%D1%85%D0%B5%D0%BC%D0%B0-%D0%AD%D0%92%D0%9C.-%D0%92%D0%B8%D0%B4%D1%8B-%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D0%B8.-%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%BD%D0%B0%D1%8F-%D1%88%D0%B8%D0%BD%D0%B0.)

## **Структурная схема ЭВМ.**

****

## **Виды памяти**

Байт - минимальная адресуемая единица памяти (8 бит).

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

Параграф - 16 байт

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.

## **Системная шина**

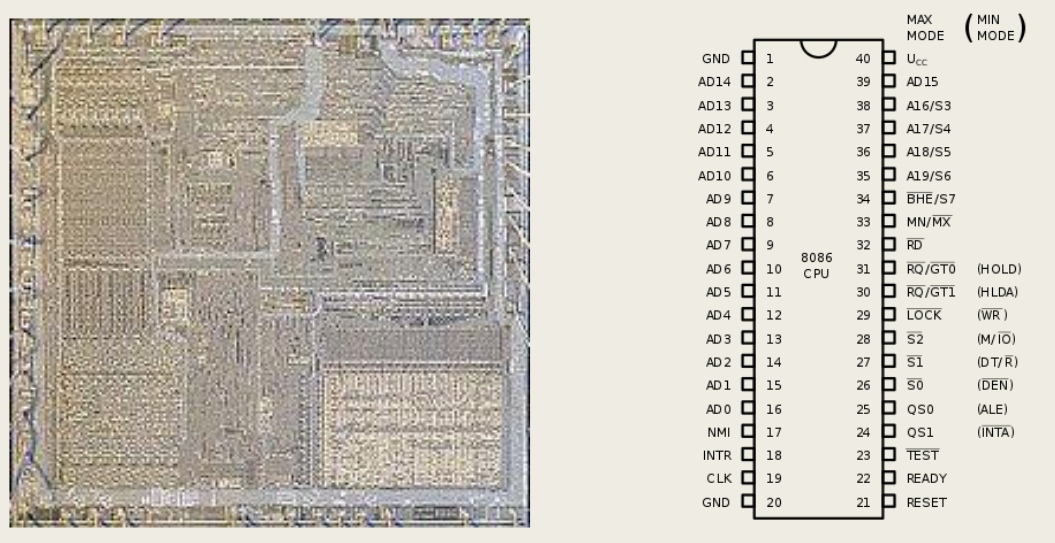
Системная шина - соединение, служащее для передачи данных между функциональными блоками компьютера.

Судя по схеме, системная шина делится на несколько отдельных (управления, адреса, данных).

Любая часть ЭВМ (со схемы) имеет доступ к шине.

# [Процессор 8086. Разрядность. Регистры.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-03%5D-%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80-8086.-%D0%A0%D0%B0%D0%B7%D1%80%D1%8F%D0%B4%D0%BD%D0%BE%D1%81%D1%82%D1%8C.-%D0%A0%D0%B5%D0%B3%D0%B8%D1%81%D1%82%D1%80%D1%8B.)

## **Процессор 8086**

****

## **Разрядность**

Разрядность: 16 бит.

## **Регистры**

Всего в 8086 14 регистров.

## **Регистры общего назначения: AX, BX, CX, DX.**

Каждый из регистров имеет старшую и младшую часть, по 1 байту на каждый.

## **Индексные регистры: SI, DI**

Используются для индексации.

## **Сегментные регистры: CS (code segment), SS (stack segment), DS (data segment), ES (extra segment).**

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.

## **Указатель команды: IP.**

Этот регистр, содержит адрес-смещение следующей команды, относительно сегмента CS. Связан с CS как CS:IP

Предположим, что в CS хранится адрес 2CB5H, а в регистре IP хранится смещение 123H. Таким образом, адрес следующей команды 2CB50H + 123H = 2CC73H.

## **Регистр флагов: FLAGS.**

Внутри используют не все 16 бит. Каждый флаг - один бит.

## **Регистры для работы со стеком: SP и BP**

SP - stack pointer, BP - base pointer. SP всегда указывает на вершину стека (то есть при операциях PUSH он уменьшается на переданный размер (стек растет вниз), а при POP - увеличивается). Регистр BP обычно тоже указывает на какое-то место в стеке - с его помощью адресуют локальные переменные, например, MOV AX, [BP+4]

# [Процессор 8086. Регистр флагов.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-04%5D-%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80-8086.-%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.)

## **Регистры флагов в 8086.**

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

Каждый флаг представляет собою 1 бит, выставляемый в 0 (флаг сброшен) или в 1 (флаг установлен). Не существует специальных команд, позволяющих обратится к этому регистру напрямую.

Хотя разрядность регистра FLAGS 16 бит, реально используют не все 16. Остальные были зарезервированны при разработке процессора, но так и не были использованны.

Вот за что отвечает каждый бит в регистре FLAGS:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| CF | - | PF | - | AF | - | ZF | SF | TF | IF | DF | OF | IOPL | IOPL | NT | - |

* **CF (carry flag)** - флаг переноса - устанавливается в 1, если результат предыдущей операции не уместился в приемник и произошел перенос или если требуется заем при вычитании. Иначе 0.
* **PF (parity flag)** - флаг чётности - устанавливается в 1, если младший байт результата предыдущей операции содержит четное количество единиц.
* **AF (auxiliary carry flag)** - вспомогательный флаг переноса - устанавливается в 1, если в результате предыдущей операции произошел перенос из 3 в 4 или заем из 4 в 3 биты.
* **ZF (zero flag)** - флаг нуля - устанавливается в 1, если если результат предыдущей команды равен 0.
* **SF (sign flag)** - флаг знака - всегда равен старшему биту результата.
* **TF (trap flag)** - флаг трассировки - предусмотрен для работы отладчиков в пошаговом режиме. Если поставить в 1, после каждой команды будет происходить передача управления отладчику.
* **IF (interrupt enable flag)** - флаг разрешения прерываний - если 0 процессор перестает обрабатывать прерывания от внешних устройств.
* **DF (direction flag)** - флаг направления - контролирует поведение команд обработки строк. Если 0, строки обрабатываются слева направо, если 1 справа налево.
* **OF (overflowflag)** - флаг переполнения - устанавливается в 1, если результат предыдущей операции над числами со знаком выходит за допустимые для них пределы.
* **IOPL (I/O privilege level)** - уровень приоритета ввода-вывода - а это на 286, на не нужно пока.
* **NT (nested task)** - флаг вложенности задач - а это на 286, на не нужно пока.

# [Процессор 8086. Шина адреса. Сегментная модель памяти.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-05%5D-%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80-8086.-%D0%A8%D0%B8%D0%BD%D0%B0-%D0%B0%D0%B4%D1%80%D0%B5%D1%81%D0%B0.-%D0%A1%D0%B5%D0%B3%D0%BC%D0%B5%D0%BD%D1%82%D0%BD%D0%B0%D1%8F-%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C-%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D0%B8.)

## **Шина адреса в 8086**

* Шина адреса - 20 бит, что позволяет адресовать 2^20 (1мб) памяти, а не 2^16 б.
* Шина данных - 16 бит.

## **Сегментная модель памяти**

Архитектура 8086 имеет четыре сегментных регистра (см. вопрос №3).

Логический адрес записывают как сегмент:смещение (и те, и те в 16 с/с). В реальном режиме для вычисления физического адреса, адрес из сегмента сдвигают влево на 4 разряда (можно сказать, что просто приписывают 0 в конце или умножают на 16) и добавляют смещение. Например, логический адрес 7522:F139 дает физический адрес 84359.

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

Такой режим работы процессора называют реальным режимом адресации процессора

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

# [Выполнение программы. Машинный код. Исполняемые файлы. Язык ассемблера.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-06%5D-%D0%92%D1%8B%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5-%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D1%8B.-%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%BD%D1%8B%D0%B9-%D0%BA%D0%BE%D0%B4.-%D0%98%D1%81%D0%BF%D0%BE%D0%BB%D0%BD%D1%8F%D0%B5%D0%BC%D1%8B%D0%B5-%D1%84%D0%B0%D0%B9%D0%BB%D1%8B.-%D0%AF%D0%B7%D1%8B%D0%BA-%D0%B0%D1%81%D1%81%D0%B5%D0%BC%D0%B1%D0%BB%D0%B5%D1%80%D0%B0.)

## **Выполнение программы**

1. Определение формата файла (.COM или .EXE, в случае 8086)

2. Чтение и разбор заголовка

3. Считывание разделов исполняемого модуля (файла) в ОЗУ по необходимым адресам.

4. Подготовка к запуску, если требуется. (установка регистров; настройка окружения, загрузка библиотек (см. 1 ЛР, 2 часть))

5. Передача управления на точку входа.

Дальше выполняются инструкции заданные в самой программе.

## **Машинный код**

Машинный код - набор команд, который напрямую интерпертируется процессором.

Каждая машинная инструкция выполняет определенное действие.

## **Исполняемые файлы**

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

Стадии получения: компиляция + линковка (компоновка).

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

Линковщик (компоновщик) - связывает несколько объектных файлов в исполняемый файл

## **Язык ассемблера**

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

# [Классификация команд процессора x86.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-07%5D-%D0%9A%D0%BB%D0%B0%D1%81%D1%81%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F-%D0%BA%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4-%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80%D0%B0-x86.)

## **Классификация команд процессора х86**

* Команды пересылки данных. Например, mov.
* Арифметические и логические команды. Add, sub, mul, div, and, or, xor и т. д.
* Команды переходов. То есть, нарушение естественного (последовательного) выполнения команд. Команды условного - когда есть какое-то условие (например, je). Безусловного перехода - без условия (например, jmp)
* Команды работы с подпрограммами. Вызов подпрограмы - call, возврат из неё - ret.
* Команды управления процессором. Например, NOP - no operation (ничего не делает). HALT - full stop

# [Команда пересылки данных.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-08%5D-%D0%9A%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%B0-%D0%BF%D0%B5%D1%80%D0%B5%D1%81%D1%8B%D0%BB%D0%BA%D0%B8-%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85.)

## **MOV <приемник>, <источник>.**

* Приемник: POH (регистр общего назначения), сегментный регистр, переменная (то есть ячейка памяти)
* Источник: непосредственный операнд (например, число), POH, сегментный регистр, переменная

Нельзя загрузить в сегментный регистр значение непосредственно из памяти. Поэтому для этого используют промежуточный регистр (в начале лабы всегда так делали).

Переменные не могут быть одновременно и источником, и приемником.

## **XCHG <операнд1>, <операнд2>**

Обмен операндов между собой. Выполнятеся либо над двумя регистрами, либо регистр + переменная.

# [Команды целочисленной арифметики.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-09%5D-%D0%9A%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D1%8B-%D1%86%D0%B5%D0%BB%D0%BE%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%BD%D0%BE%D0%B9-%D0%B0%D1%80%D0%B8%D1%84%D0%BC%D0%B5%D1%82%D0%B8%D0%BA%D0%B8.)

## **ADD <приёмник>, <источник>**

Сложение приемника и источника. Сумма - в приемник, источник не изменяется.

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

## **SUB <приёмник>, <источник>**

Вычитание, всё точно так же как и в ADD. Отрицательные числа можно отслеживать с помощью флага SF.

## **MUL <источник>**

Умножение без знака.

* Источник - область памяти, РОН.

Умножаются источник и AL/AX, в зависимости от размера источника. Результат помещается в AX либо DX:AX.

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

## **DIV <источник>**

Деление без знака.

* Источник - область памяти, РОН.

Деление AL/AX на источник. Результат помещается в AL/AX, остаток - в AH/DX.

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

Ещё есть команды деления/умножения со знаком: IDIV, IMUL

## **DEC <операнд>, INC <операнд>**

* INC - инкремент.
* DEC - декремент.

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

INC и DEC, в отличии от ADD и SUB, не затрагивают флаг CF.

Все эти команды меняют регистр флагов (FLAGS), в зависимости от результата

# [Команды побитовой арифметики.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-10%5D-%D0%9A%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D1%8B-%D0%BF%D0%BE%D0%B1%D0%B8%D1%82%D0%BE%D0%B2%D0%BE%D0%B9-%D0%B0%D1%80%D0%B8%D1%84%D0%BC%D0%B5%D1%82%D0%B8%D0%BA%D0%B8.)

* AND <приёмник>, <источник> - побитовое "И"
* OR <приёмник>, <источник> - побитовое "ИЛИ"
* XOR <приёмник>, <источник> - побитовое исключающее "ИЛИ"
* NOT <приёмник> - инверсия
* SHL <приёмник>, <счётчик> - сдвиг влево (SAL - эквивалентная команда)
* SHR <приёмник>, <счётчик> - сдвиг вправо
* SAR <приёмник>, <счётчик> - сдвиг вправо с сохранением знакового бита
* ROR <приёмник>, <счётчик> - циклический сдвиг вправо
* RСR <приёмник>, <счётчик> - циклический сдвиг вправо, через флаг переноса
* ROL <приёмник>, <счётчик> - циклический сдвиг влево
* RСL <приёмник>, <счётчик> - циклический сдвиг влево, через флаг переноса

Все эти команды меняют регистр FLAGS.

# [Команды передачи управления.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-11%5D-%D0%9A%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D1%8B-%D0%BF%D0%B5%D1%80%D0%B5%D0%B4%D0%B0%D1%87%D0%B8-%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F.)

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

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

## **Виды безусловных переходов**

JMP - оператор безусловного перехода.

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

Для короткого и ближнего переходов непосредственный операнд (число) прибавляется к IP. Регистры и переменные заменяют старое значение в IP (CS:IP).

## **Команда безусловной передачи управления JMP**

JMP <операнд>

* Передаёт управление в другую точку программы, не сохраняя какой-либо информации для возврата.
* Операнд - непосредственный адрес, регистр или переменная.

## **Команды условных переходов J.. (Зубков, Assembler, …, глава 2)**

* Переход типа short или near
* Обычно используются в паре с CMP
* "Выше" и "ниже" - при сравнении беззнаковых чисел
* "Больше" и "меньше" - при сравнении чисел со знаком

| **Команда** | **Описание** | **Состояние флагов для выполнения перехода** |
| --- | --- | --- |
| JO | Есть переполнение | OF = 1 |
| JNO | Нет переполнения | OF = 0 |
| JS | Есть знак | SF = 1 |
| JNS | Нет знака | SF = 0 |
| JE/JZ | Если равно/если ноль | ZF = 0 |
| JNE/JNZ | Если не равно/если не ноль | ZF = 0 |
| JP/JPE | Есть четность/четное | PF = 1 |
| JNP/JPO | Нет четности/нечетное | PF = 0 |
| JCXZ | CX = 0 |  |

* + ***Беззнаковые***

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

* + ***Знаковые***

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

# [Структура программы на языке ассемблера. Модули. Сегменты.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-12%5D-%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0-%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D1%8B-%D0%BD%D0%B0-%D1%8F%D0%B7%D1%8B%D0%BA%D0%B5-%D0%B0%D1%81%D1%81%D0%B5%D0%BC%D0%B1%D0%BB%D0%B5%D1%80%D0%B0.-%D0%9C%D0%BE%D0%B4%D1%83%D0%BB%D0%B8.-%D0%A1%D0%B5%D0%B3%D0%BC%D0%B5%D0%BD%D1%82%D1%8B.)

Любая программа состоит из сегментов

/// ! Виды сегментов:

* Сегмент кода
* Сегмент данных
* Сегмент стека

/// ! Описание сегмента в исходном коде:

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

…

*имя* ENDS

* Выравнивание по умолчанию - параграф (PARA)
* Тип по умолчанию - PRIVATE
* Класс - любая метка, взятая в одинарные кавычки. Все сегменты с одинаковым классом, будут расположенным друг за другом (в исполнимом файле, даже PRIVATE)

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

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

## **Директива SEGMENT**

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

Выравнивание:

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

Тип:

* **PUBLIC** - заставляет компоновщик соединить все сегменты с одинаковым именем. Новый объединенный сегмент будет целым и непрерывным. Все адреса (смещения) объектов, а это могут быть, в зависимости от типа сегмента, команды или данные, будут вычисляться относительно начала этого нового сегмента;
* **STACK** - определение сегмента стека. Заставляет компоновщик соединить все одноименные сегменты и вычислять адреса в этих сегментах относительно регистра SS. Комбинированный тип STACK (стек) аналогичен комбинированному типу PUBLIC, за исключением того, что регистр SS является стандартным сегментным регистром для сегментов стека. Регистр SP устанавливается на конец объединенного сегмента стека. Если не указано ни одного сегмента стека, компоновщик выдаст предупреждение, что стековый сегмент не найден. Если сегмент стека создан, а комбинированный тип STACK не используется, программист должен явно загрузить в регистр SS адрес сегмента (подобно тому, как это делается для регистра DS);
* **COMMON** - располагает все сегменты с одним и тем же именем по одному адресу. Все сегменты с данным именем будут перекрываться и совместно использовать память. Размер полученного в результате сегмента будет равен размеру самого большого сегмента;
* **AT** - располагает сегмент по абсолютному адресу параграфа (параграф — объем памяти, кратный 16, поэтому последняя шестнадцатеричная цифра адреса параграфа равна 0). Абсолютный адрес параграфа задается выражением хххx. Компоновщик располагает сегмент по заданному адресу памяти (это можно использовать, например, для доступа к видеопамяти или области ПЗУ), учитывая атрибут комбинирования. Физически это означает, что сегмент при загрузке в память будет расположен, начиная с этого абсолютного адреса параграфа, но для доступа к нему в соответствующий сегментный регистр должно быть загружено заданное в атрибуте значение. Все метки и адреса в определенном таким образом сегменте отсчитываются относительно заданного абсолютного адреса;
* **PRIVATE** (по умолчанию) - сегмент не будет объединяться с другими сегментами с тем же именем вне данного модуля.

Класс:

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

## **Директива ASSUME**

ASSUME *регистр* : *имя сегмента*

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

## **Модель памяти**

.model *модель*, *язык*, *модификатор*

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

## **Конец программы и точка входа**

...

END start

* start - имя метки, объявленной в сегменте кода и указывающее на команду, с которой начнётся исполнение программы.
* Если в программе несколько модулей, только один может содержать начальный адрес.

# [Директивы выделения памяти. Метки.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-13%5D-%D0%94%D0%B8%D1%80%D0%B5%D0%BA%D1%82%D0%B8%D0%B2%D1%8B-%D0%B2%D1%8B%D0%B4%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F-%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D0%B8.-%D0%9C%D0%B5%D1%82%D0%BA%D0%B8.)

Псевдокоманда - директива ассемблера, которая приводит к включению данных или кода в программу, но не соответствует никакой команде процессора.

Псевдокоманды определения данных указывают, что в соответствующем месте располагается переменная, резервируют под неё место заданного типа, заполняют значением и ставят в соответствие метку.

Виды: DB (1), DW (2), DD (4), DF (6), DQ (8), DT (10).

Примеры:

* a DB 1
* float\_number DD 3.5e7
* text\_string DB 'Hello, world!'

DUP - заполнение повторяющимися данными.

? - неинициализированное значение.

uninitialized DW 512 DUP(?)

# [Директива SEGMENT.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-14%5D-%D0%94%D0%B8%D1%80%D0%B5%D0%BA%D1%82%D0%B8%D0%B2%D0%B0-SEGMENT.)

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

Выравнивание:

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

Тип:

* PUBLIC - заставляет компоновщик соединить все сегменты с одинаковым именем. Новый объединенный сегмент будет целым и непрерывным. Все адреса (смещения) объектов, а это могут быть, в зависимости от типа сегмента, команды или данные, будут вычисляться относительно начала этого нового сегмента;
* STACK - определение сегмента стека. Заставляет компоновщик соединить все одноименные сегменты и вычислять адреса в этих сегментах относительно регистра SS. Комбинированный тип STACK (стек) аналогичен комбинированному типу PUBLIC, за исключением того, что регистр SS является стандартным сегментным регистром для сегментов стека. Регистр SP устанавливается на конец объединенного сегмента стека. Если не указано ни одного сегмента стека, компоновщик выдаст предупреждение, что стековый сегмент не найден. Если сегмент стека создан, а комбинированный тип STACK не используется, программист должен явно загрузить в регистр SS адрес сегмента (подобно тому, как это делается для регистра DS);
* COMMON - располагает все сегменты с одним и тем же именем по одному адресу. Все сегменты с данным именем будут перекрываться и совместно использовать память. Размер полученного в результате сегмента будет равен размеру самого большого сегмента;
* AT - располагает сегмент по абсолютному адресу параграфа (параграф — объем памяти, кратный 16, поэтому последняя шестнадцатеричная цифра адреса параграфа равна 0). Абсолютный адрес параграфа задается выражением хххx. Компоновщик располагает сегмент по заданному адресу памяти (это можно использовать, например, для доступа к видеопамяти или области ПЗУ), учитывая атрибут комбинирования. Физически это означает, что сегмент при загрузке в память будет расположен, начиная с этого абсолютного адреса параграфа, но для доступа к нему в соответствующий сегментный регистр должно быть загружено заданное в атрибуте значение. Все метки и адреса в определенном таким образом сегменте отсчитываются относительно заданного абсолютного адреса;
* PRIVATE (по умолчанию) - сегмент не будет объединяться с другими сегментами с тем же именем вне данного модуля.

Класс:

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

# [Директива ASSUME.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-15%5D-%D0%94%D0%B8%D1%80%D0%B5%D0%BA%D1%82%D0%B8%D0%B2%D0%B0-ASSUME.)

ASSUME *регистр* : *имя сегмента*

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

Data1 SEGMENT WORD 'DATA'

Var1 DW 0

Data1 ENDS

Data2 SEGMENT WORD 'DATA'

Var2 DW 0

Data2 ENDS

Code SEGMENT WORD 'CODE'

ASSUME CS:Code

ProgramStart:

mov ax,Data1

mov ds,ax

ASSUME DS:Data1

mov ax,Data2

mov es,ax

ASSUME ES:Data2

mov ax,[Var2]

.

.

.

Code ENDS

END ProgramStart

Если не написать ASSUME, то при работе с переменнеыми придется явно указывать селектор сегмента.

# [Директива END. Точка входа.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-16%5D-%D0%94%D0%B8%D1%80%D0%B5%D0%BA%D1%82%D0%B8%D0%B2%D0%B0-END.-%D0%A2%D0%BE%D1%87%D0%BA%D0%B0-%D0%B2%D1%85%D0%BE%D0%B4%D0%B0.)

.

.

.

END start

**start** - имя метки (в данном примере имя start, а вообще имя может быть любым), объявленной в сегменте кода и указывающее на команду, с которой начнётся исполнение программы.

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

В остальных модулях должна быть присутствовать дирректива END без метки.

# [Виды переходов. Условные, безусловные переходы. Короткий, ближний, дальний переход.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-17%5D-%D0%92%D0%B8%D0%B4%D1%8B-%D0%BF%D0%B5%D1%80%D0%B5%D1%85%D0%BE%D0%B4%D0%BE%D0%B2.-%D0%A3%D1%81%D0%BB%D0%BE%D0%B2%D0%BD%D1%8B%D0%B5,-%D0%B1%D0%B5%D0%B7%D1%83%D1%81%D0%BB%D0%BE%D0%B2%D0%BD%D1%8B%D0%B5-%D0%BF%D0%B5%D1%80%D0%B5%D1%85%D0%BE%D0%B4%D1%8B.-%D0%9A%D0%BE%D1%80%D0%BE%D1%82%D0%BA%D0%B8%D0%B9,-%D0%B1%D0%BB%D0%B8%D0%B6%D0%BD%D0%B8%D0%B9,-%D0%B4%D0%B0%D0%BB%D1%8C%D0%BD%D0%B8%D0%B9-%D0%BF%D0%B5%D1%80%D0%B5%D1%85%D0%BE%D0%B4.)

## **Виды безусловных переходов**

JMP - оператор безусловного перехода.

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

Для короткого и ближнего переходов непосредственный операнд (число) прибавляется к IP. Регистры и переменные заменяют старое значение в IP (CS:IP).

## **Команда безусловной передачи управления JMP**

JMP <операнд>

* Передаёт управление в другую точку программы, не сохраняя какой-либо информации для возврата.
* Операнд - непосредственный адрес, регистр или переменная.

## **Команды условных переходов J.. (Зубков, Assembler, …, глава 2)**

* Переход типа short или near
* Обычно используются в паре с CMP
* "Выше" и "ниже" - при сравнении беззнаковых чисел
* "Больше" и "меньше" - при сравнении чисел со знаком

| **Команда** | **Описание** | **Состояние флагов для выполнения перехода** |
| --- | --- | --- |
| JO | Есть переполнение | OF = 1 |
| JNO | Нет переполнения | OF = 0 |
| JS | Есть знак | SF = 1 |
| JNS | Нет знака | SF = 0 |
| JE/JZ | Если равно/если ноль | ZF = 0 |
| JNE/JNZ | Если не равно/если не ноль | ZF = 0 |
| JP/JPE | Есть четность/четное | PF = 1 |
| JNP/JPO | Нет четности/нечетное | PF = 0 |
| JCXZ | CX = 0 |  |

* + ***Беззнаковые***

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

* + ***Знаковые***

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

# [Способы адресации.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-18%5D-%D0%A1%D0%BF%D0%BE%D1%81%D0%BE%D0%B1%D1%8B-%D0%B0%D0%B4%D1%80%D0%B5%D1%81%D0%B0%D1%86%D0%B8%D0%B8.)

* Регистровая адресация (mov ax, bx)
* Непосредственная адресация (mov ax, 2)
* Прямая адресация (mov ax, ds:0032) (метка во время компиляции преобразуется в прямую)
* Косвенная адресация (mov ax, [bx]). В 8086 допустимы BX, BP, SI, DI
* Адресация по базе со сдвигом (mov ax, [bx]+2; mov ax, 2[bx]).
* Адресация по базе с индексированием (допустимы BX+SI, BX+DI, BP+SI, BP+DI):
  + mov ax, [bx+si+2]
  + mov ax, [bx][si]+2
  + mov ax, [bx+2][si]
  + mov ax, [bx][si+2]
  + mov ax, 2[bx][si]

# [Команда сравнения.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-19%5D-%D0%9A%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%B0-%D1%81%D1%80%D0%B0%D0%B2%D0%BD%D0%B5%D0%BD%D0%B8%D1%8F.)

## **CMP <приемник>, <источник>**

Источник - число, регистр или переменная.

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

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

## **TEST <приемник>, <источник>**

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

Можно использовать для проверки на ноль, например TEST bx, bx

# [Команды условных переходов.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-20%5D-%D0%9A%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D1%8B-%D1%83%D1%81%D0%BB%D0%BE%D0%B2%D0%BD%D1%8B%D1%85-%D0%BF%D0%B5%D1%80%D0%B5%D1%85%D0%BE%D0%B4%D0%BE%D0%B2.)

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

Команды условных переходов имеют вид J?? <метка>.

* Не зависящие от знака

| **Команда** | **Описание** | **Состояние флагов для выполнения перехода** |
| --- | --- | --- |
| JO | Есть переполнение | OF = 1 |
| JNO | Нет переполнения | OF = 0 |
| JS | Есть знак | SF = 1 |
| JNS | Нет знака | SF = 0 |
| JE/JZ | Если равно/если ноль | ZF = 0 |
| JNE/JNZ | Если не равно/если не ноль | ZF = 0 |
| JP/JPE | Есть четность/четное | PF = 1 |
| JNP/JPO | Нет четности/нечетное | PF = 0 |
| JCXZ | CX = 0 |  |

* + ***Беззнаковые***

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

* + ***Знаковые***

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

# [Команда XLAT/XLATB.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-21%5D-%D0%9A%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%B0-XLAT-XLATB.)

## **XLAT <адрес>**

## **XLATB**

Помещает в AL байт из таблицы по адресу DS:BX 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.

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

# [Команда LEA.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-22%5D-%D0%9A%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%B0-LEA.)

## **LEA <приёмник>, <источник>**

Вычисляет эффективный адрес источника и помещает его в приёмник. Позволяет вычислить адрес, описанный сложным методом адресации (да и просто его загрузить).

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.

# [Команды десятичной арифметики.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-23%5D-%D0%9A%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D1%8B-%D0%B4%D0%B5%D1%81%D1%8F%D1%82%D0%B8%D1%87%D0%BD%D0%BE%D0%B9-%D0%B0%D1%80%D0%B8%D1%84%D0%BC%D0%B5%D1%82%D0%B8%D0%BA%D0%B8.)

## **Десятичная арифметика DAA, DAS, AAA, AAS, AAM, AAD**

Это ваще жесть какая-то...

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

## **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

## **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).

## **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'

# [Команды сдвига.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-24%5D-%D0%9A%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D1%8B-%D1%81%D0%B4%D0%B2%D0%B8%D0%B3%D0%B0.)

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

* SAR <операнд>, 1
* SAR <операнд>, CL
* SHR <операнд>, 1
* SHR <операнд>, CL

Команды сдвига вправо SAR и SHR сдвигают все биты вниз (к младшему), а самый младший бит переносится во флаг CF. Команда SAR фактически выполняет знаковое деление на два, четыре и т.д. с округлением в сторону -∞(не тоже самое, что команда IDIV), старший бит исходного опернада остается неизменным, а его значение копируется при сдвиге в менее значимые биты. Команда SHR выполняет беззнаковое деление, старшие биты при сдвиге заполняются нулями.

* SAL <операнд>, 1
* SAL <операнд>, CL
* SHL <операнд>, 1
* SHL <операнд>, CL

Команды сдвига влево SAL и SHL идентичны, они сдвигают все биты вверх (к старшему), при этом самый старший бит операнда-источника сдвигается во флаг CF. Такое действие равнозначно беззнаковому умножению исходного операнда на два, четыре и т.д, младшие биты заполняются нулями.

* ROR <операнд>, 1
* ROR <операнд>, CL

Циклический сдвиг вправо, флаг CF получает копию сдвинутого бита.

* ROL <операнд>, 1
* ROL <операнд>, CL

Циклический сдвиг влево, флаг CF получает копию сдвинутого бита.

* RCR <операнд>, 1
* RCR <операнд>, CL

Циклический сдвиг вправо через флаг CF, сдвинутый бит помещается в CF, вместо него в число с другой стороны добавляется значение CF.

* RCL <операнд>, 1
* RCL <операнд>, CL

Циклический сдвиг влево через флаг CF, сдвинутый бит помещается в CF, вместо него в число с другой стороны добавляется значение CF.

# [Команда организации цикла.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-25%5D-%D0%9A%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%B0-%D0%BE%D1%80%D0%B3%D0%B0%D0%BD%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D0%B8-%D1%86%D0%B8%D0%BA%D0%BB%D0%B0.)

## **LOOP <метка>**

уменьшает CX и выполняет "короткий" переход на метку, если CX не равен нулю.

## **LOOPE/LOOPZ <метка> - цикл "пока равно"/"пока ноль"**

Декрементируют CX и выполняют переход, если CX не ноль и если выполняется условие (флаг ZF == 0).

## **LOOPNE/LOOPNZ <метка> - цикл "пока не равно"/"пока не ноль"**

Декрементируют CX и выполняют переход, если CX не ноль и если выполняется условие (флаг ZF != 0).

# [Строковые операции. Префиксы повторения.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-26%5D-%D0%A1%D1%82%D1%80%D0%BE%D0%BA%D0%BE%D0%B2%D1%8B%D0%B5-%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%B8.-%D0%9F%D1%80%D0%B5%D1%84%D0%B8%D0%BA%D1%81%D1%8B-%D0%BF%D0%BE%D0%B2%D1%82%D0%BE%D1%80%D0%B5%D0%BD%D0%B8%D1%8F.)

Строка-источник - DS:SI, строка-приёмник - ES:DI. За один раз обрабатывается один байт (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.

В каждой команде источник и приемник можно опустить.

## **Префиксы: REP/REPE/REPZ/REPNE/REPNZ**

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

Префиксы REP, 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

# [Стек. Регистры, связанные со стеком. Команды записи/извлечения из стека.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-27%5D-%D0%A1%D1%82%D0%B5%D0%BA.-%D0%A0%D0%B5%D0%B3%D0%B8%D1%81%D1%82%D1%80%D1%8B,-%D1%81%D0%B2%D1%8F%D0%B7%D0%B0%D0%BD%D0%BD%D1%8B%D0%B5-%D1%81%D0%BE-%D1%81%D1%82%D0%B5%D0%BA%D0%BE%D0%BC.-%D0%9A%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D1%8B-%D0%B7%D0%B0%D0%BF%D0%B8%D1%81%D0%B8-%D0%B8%D0%B7%D0%B2%D0%BB%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F-%D0%B8%D0%B7-%D1%81%D1%82%D0%B5%D0%BA%D0%B0.)

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

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

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

Регистр SP — указывает на вершину стека

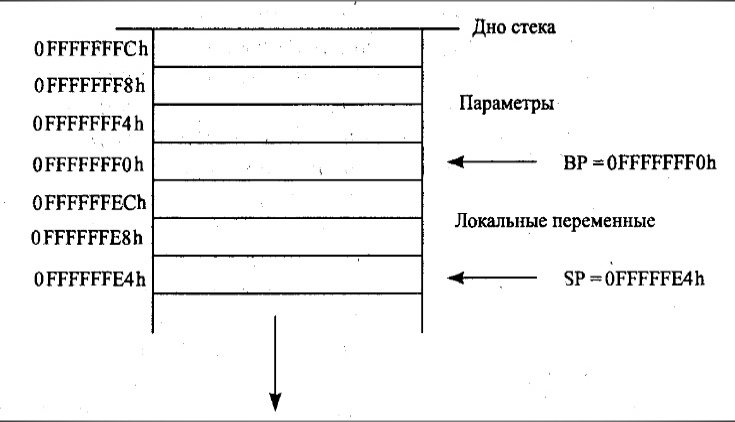
В x86 стек "растёт вниз", в сторону уменьшения адресов (от максимально возможно адреса). При запуске программы SP указывает на конец сегмента.

## **BP (Base Pointer)**

Используется в подпрограмме для сохранения "начального" значения SP.

Так же, используется для адресации параметров и локальных переменых.

При вызове подпрограммы параметры кладут на стек, а в BP кладут текущее значение SP. Если программа использует стек для хранения локальных переменных, SP изменится и таким образом можно будет считывать переменные напрямую из стека (их смещения запишутся как BP + номер параметра)



## **Команды работы со стеком**

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

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

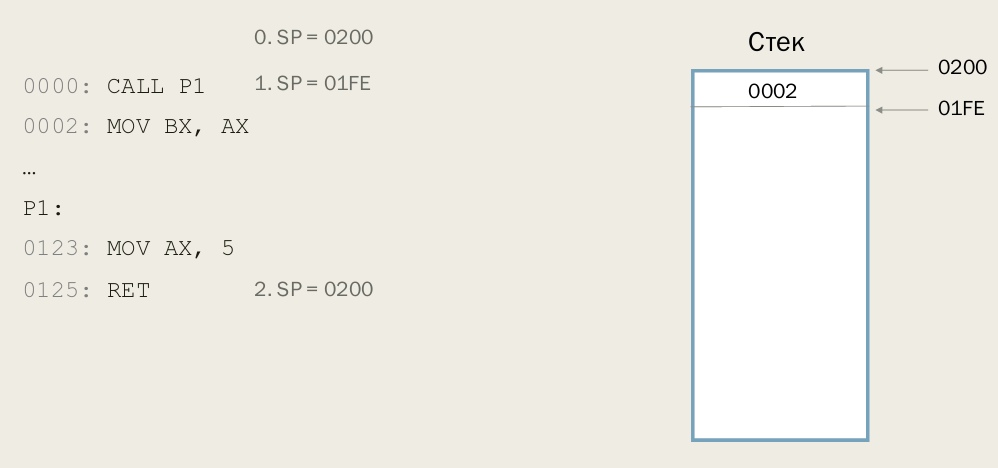
PUSHA — поместить в стек регистры AX, CX, DX, BX, SP, BP, SI, DI. (регистры общего назначения + SP + BP)

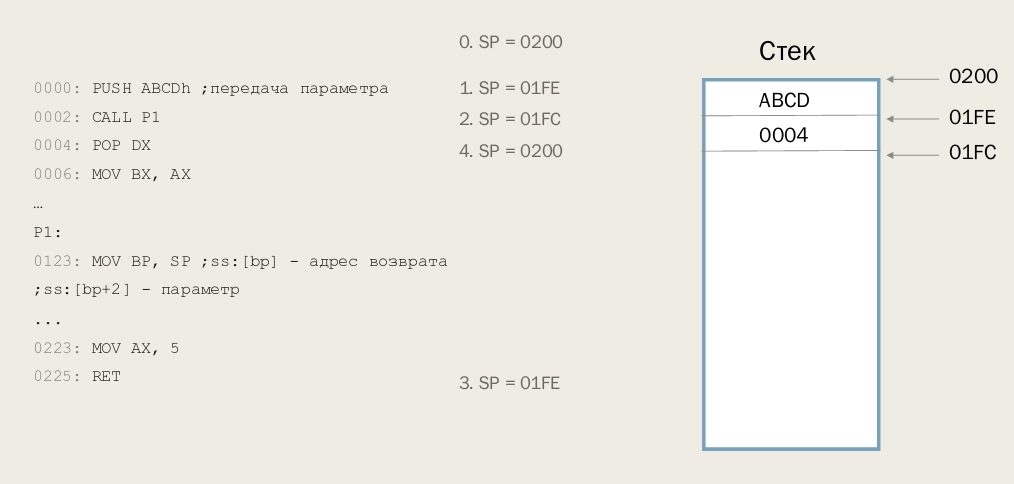
POPA — загрузить регистры из стека (SP игнорируется)

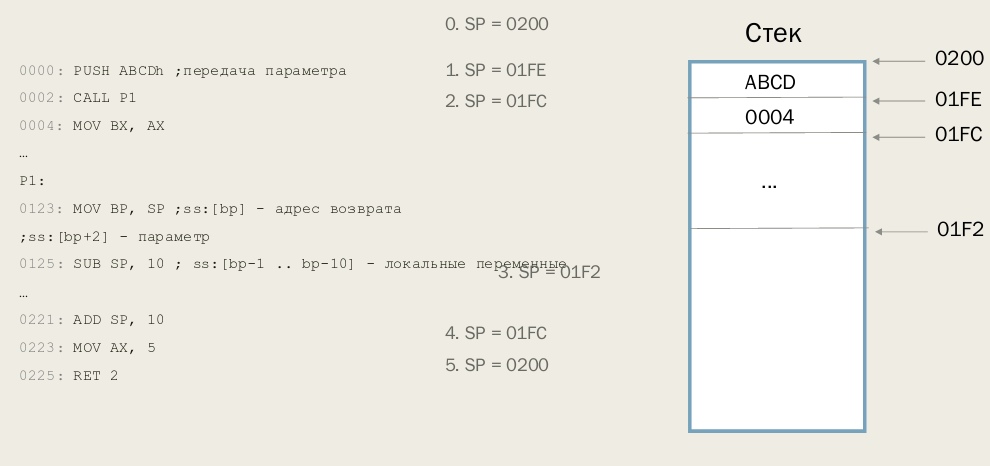
PUSHF — поместить в стек флаги.

POPF — загрузить флаги из стека

## **Примеры использования**

****

****

****

# [Стек. Использование при вызове подпрограмм. Команды вызоваподпрограммы и возврата.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-28%5D-%D0%A1%D1%82%D0%B5%D0%BA.-%D0%98%D1%81%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5-%D0%BF%D1%80%D0%B8-%D0%B2%D1%8B%D0%B7%D0%BE%D0%B2%D0%B5-%D0%BF%D0%BE%D0%B4%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC.-%D0%9A%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D1%8B-%D0%B2%D1%8B%D0%B7%D0%BE%D0%B2%D0%B0-%D0%98-%D0%BF%D0%BE%D0%B4%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D1%8B-%D0%B8-%D0%B2%D0%BE%D0%B7%D0%B2%D1%80%D0%B0%D1%82%D0%B0.)

## **СALL и RET**

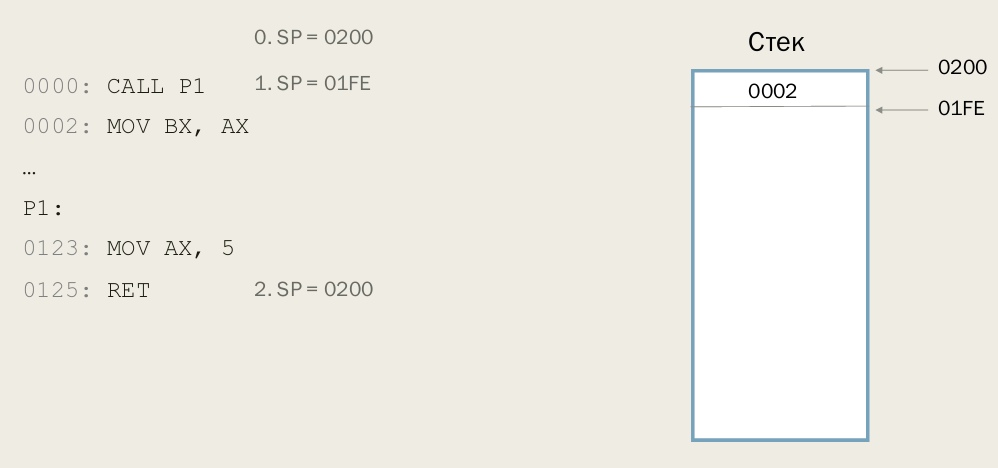
CALL <операнд> — передает управление на адрес <операнд>.

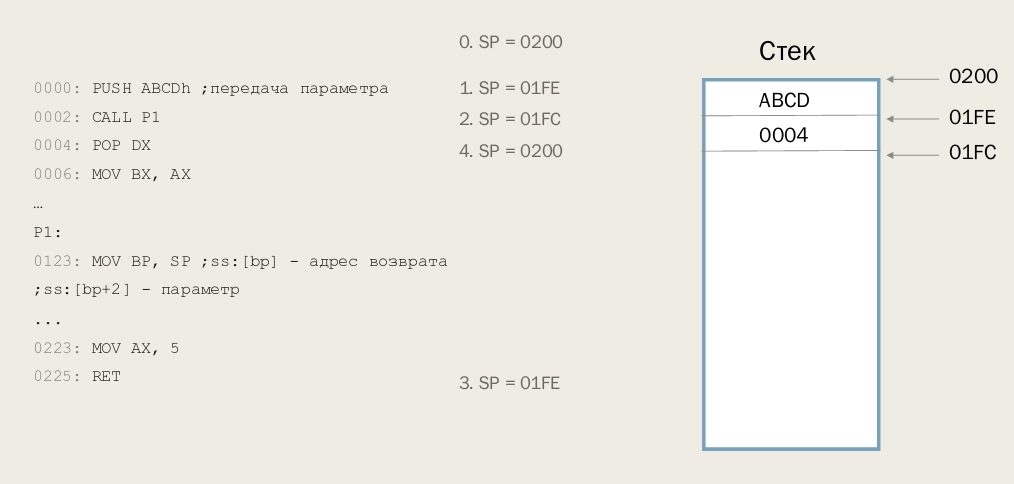
Сохраняет адрес следующей команды в стеке (уменьшает SP и записывает по его адресу IP либо CS:IP, в зависимости от размера аргумента).

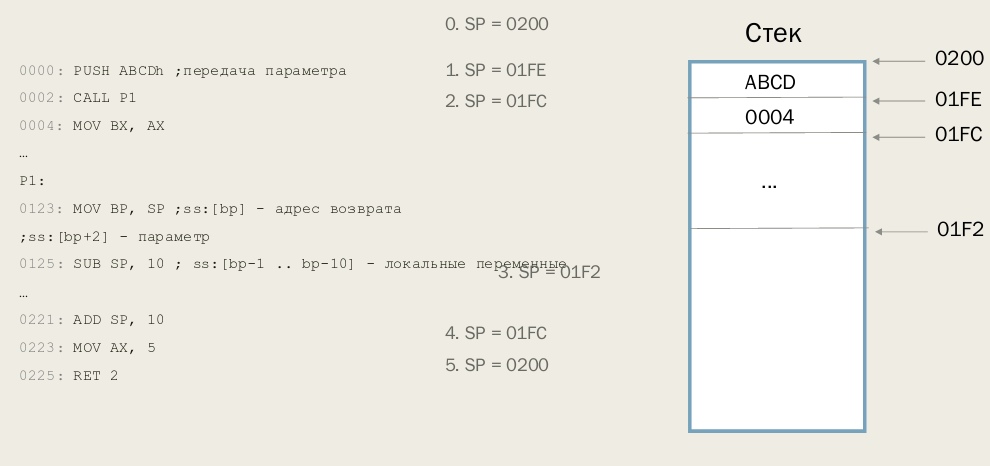
RET/RETN/RETF <число> — загружает из стека адерс возврата, увеличивая SP.

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

## **Примеры использования**

****

****

****

# [Прерывания. Назначение, виды прерываний. Таблица векторов прерываний.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-29%5D-%D0%9F%D1%80%D0%B5%D1%80%D1%8B%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F.-%D0%9D%D0%B0%D0%B7%D0%BD%D0%B0%D1%87%D0%B5%D0%BD%D0%B8%D0%B5,-%D0%B2%D0%B8%D0%B4%D1%8B-%D0%BF%D1%80%D0%B5%D1%80%D1%8B%D0%B2%D0%B0%D0%BD%D0%B8%D0%B9.-%D0%A2%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D0%B0-%D0%B2%D0%B5%D0%BA%D1%82%D0%BE%D1%80%D0%BE%D0%B2%D0%BF%D1%80%D0%B5%D1%80%D1%8B%D0%B2%D0%B0%D0%BD%D0%B8%D0%B9.)

## **Прерывания**

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

Виды прерываний:

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

## **Таблица векторов прерываний**

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

# [Срабатывание прерывания. Обработчик прерывания в реальном режиме. Возврат из обработчика прерывания.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-30%5D-%D0%A1%D1%80%D0%B0%D0%B1%D0%B0%D1%82%D1%8B%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5-%D0%BF%D1%80%D0%B5%D1%80%D1%8B%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F.-%D0%9E%D0%B1%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D1%87%D0%B8%D0%BA-%D0%BF%D1%80%D0%B5%D1%80%D1%8B%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F-%D0%B2-%D1%80%D0%B5%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%BC-%D1%80%D0%B5%D0%B6%D0%B8%D0%BC%D0%B5.-%D0%92%D0%BE%D0%B7%D0%B2%D1%80%D0%B0%D1%82-%D0%B8%D0%B7-%D0%BE%D0%B1%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D1%87%D0%B8%D0%BA%D0%B0-%D0%BF%D1%80%D0%B5%D1%80%D1%8B%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F.)

## **Срабатывание прерывания**

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

## **Обработчик прерывания в реальном режиме**

Располагается в самом начале памяти, начиная с адреса 0. Доступно 256 прерываний. Каждый вектор занимает 4 байта - полный адрес. Размер всей таблицы - 1 Кб.

## **Возврат из обработчика прерываний**

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

# [Процессор 80386. Разрядность, регистры.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-31%5D-%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80-80386.-%D0%A0%D0%B0%D0%B7%D1%80%D1%8F%D0%B4%D0%BD%D0%BE%D1%81%D1%82%D1%8C,-%D1%80%D0%B5%D0%B3%D0%B8%D1%81%D1%82%D1%80%D1%8B.)

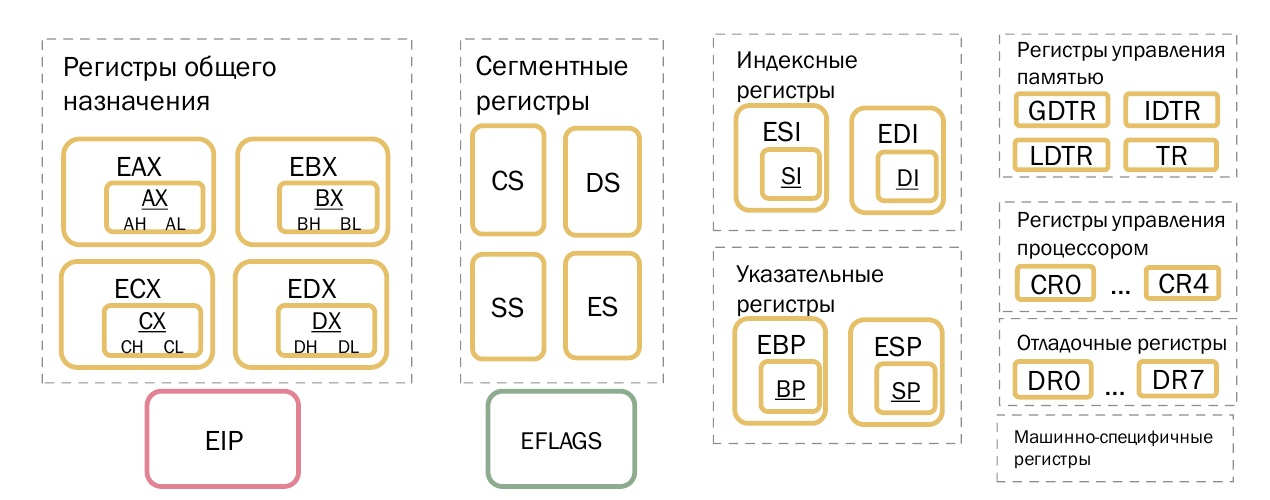
32-разрядные:

* регистры, кроме сегментных
* шина данных
* шина адреса (2^32 = 4ГБ ОЗУ)

## **Регистры**

EDX = Extended DX (обращение к частям остается (DH, DL))

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



Добавлены регистры поддержки работы в защищенном режиме

(обеспечивание разделения доступа программ между собой, между

программами и ОС и тд; эти регистры справа на картинке)

1. **Прежние расширенные**

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

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

* GDTR: (Global Descriptor Table Register) 6-байтный регистр, содержит 32-битный линейный адрес начала таблицы глобальных дескрипторов (GDT) и 16-битный размер (лимит, уменьшенный на 1). Может быть только 1 GDT
* IDTR: (Interrupt Descriptor Table Descriptor; то есть в защищенном режиме таблица векторов прерываний начинается с некоторого произвольного адреса) 6-байтный регистр, содержит 32-битный линейный адрес начала таблицы глобальных дескрипторов обработчиков прерываний (IDT) и 16-битный размер (лимит, уменьшенный на 1);
* LDTR: (Local Descriptor Table Register – для разделения памяти между программами) 10-байтный регистр, содержит 16-битный селектор для GDT (то есть какой именно базовый адрес брать

) и весь 8-байтный дескриптор из GDT, описывающий текущую таблицу

локальных дескрипторов;

* TR: (Task Register) 10-байтный регистр, содержит 16-битный селектор для GDT и весь 8-байтный дескриптор из GDT, описывающий TSS текущей задачи (Сегмент состояния задачи TSS (Task State Segment) - это структура данных, которая определяет состояние (т.е. контекст) задачи. В ней хранится содержимое всех регистров общего назначения, сегментных и некоторых системных регистров а также некоторая дополнительная информация.)

1. **Регистры управления процессором**

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

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

1. **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

1. **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
* Пример:
* mov eax, 12345678h
* xor ebx, ebx
* mov bx, 1

add eax, ebx ; eax=12345679h

**Системные и привилегированные команды**

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

# [Защищённый режим работы процессора. Многозадачность.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-32%5D-%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-%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D1%8B-%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80%D0%B0.-%D0%9C%D0%BD%D0%BE%D0%B3%D0%BE%D0%B7%D0%B0%D0%B4%D0%B0%D1%87%D0%BD%D0%BE%D1%81%D1%82%D1%8C.)

8086 (1978 г.) -> 80186 (1982 г.)-> 80286 (1982 г.) добавлен защищённый режим ->

80386 (1985 г.) архитектура стала 32-разрядной -> 80486 (1989 г.) -> Pentium -> … ->

(современные процессоры)

## **Защищенный режим**

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

Режим V86 - Прикладные программы для 8086 могут исполняться на

32-разрядных процессорах, как в реальном режиме, так и в режиме

виртуального 8086 (V86), кото­рый является особым состоянием задачи защищ

енного режима.

("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)

**Системные и привилегированные команды**

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

**Механизм защиты**

Механизм защиты - ограничение доступа к сегментам или страницам в

зависимости от уровня привилегий

* К типам сегментов реального режима (код, стек, данные) добавляется TSS (Task State Segment) - сегмент состояния задачи. В нём сохраняется вся информация о задаче на время приостановки выполнения. Размер - 68h (104) байт.
* Структура:
  + селектор предыдущей задачи
  + Регистры стека 0, 1, 2 уровней привилегий
  + EIP, EFLAGS, EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI, CS, DS, ES, FS, HS, SS, LDTR
  + флаги задачи
  + битовая карта ввода-вывода (контроль доступа программы к устройствам
* Используется ОС для диспетчеризации задач, в т. ч. переключения на стек ядра при обработке прерываний и исключений.
* Дескриптор сегмента состояния задачи (task state segment descriptor) должен находиться в GDT
* При переключении задач регистры выполняющейся задачи автоматически сохраняются в её TSS, а регистры новой задачи автоматически подгружаются, но уже из её TSS.

## **Исключения**

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

* Отказ (fault) — до, подкачка, на ту же

это исключение, которое обнаруживается и обслуживается до выполнения

инструкции, вызывающей ошибку. После обслуживания этого исключения

управление возвращается снова на ту же инструкцию (включая все префиксы),

которая вызвала отказ. Отказы, использующиеся в системе виртуальной памяти,

позволяют, например, подкачать с диска в оперативную память затребованную

страницу или сегмент.

* Ловушка (trap) — после, прерывания, на следующую

это исключение, которое обнаруживается и обслуживается после выполнения

инструкции, его вызывающей. После обслуживания этого исключения

управление возвращается на инструкцию, следующую за вызвавшей ловушку. К

классу ловушек относятся и программные прерывания.

* Аварийное завершение (abort) — непонятно и серьезно

это исключение, которое не позволяет точно установить инструкцию, его

вызвавшую. Оно используется для сообщения о серьезной ошибке, такой как

аппаратная ошибка или повреждение системных таблиц

## **Многозадачность**

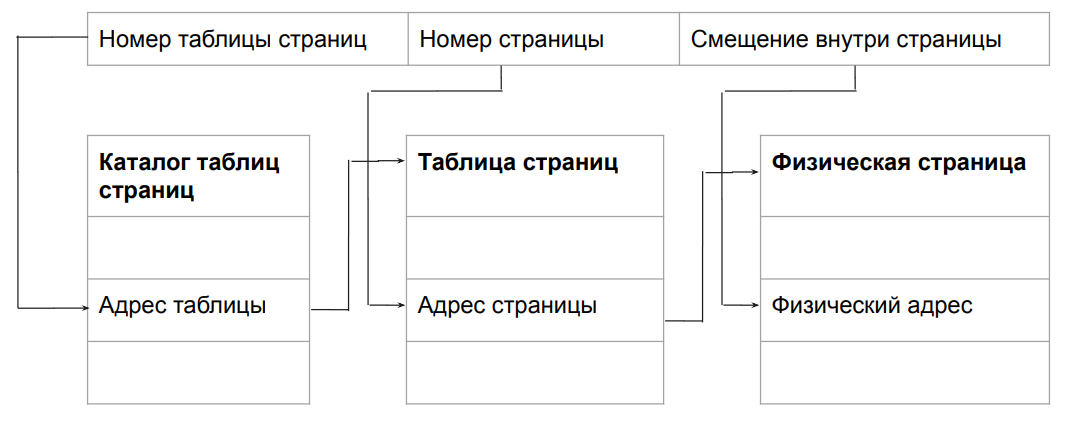
TSS (Task State Segment - сегмент состояния задачи) - специальная структура в архитектуре x86, содержащая информацию о задаче (процессе). Используется ОС для диспетчеризации задач, в т. ч. переключения на стек ядра при обработке прерываний и исключений.

Для работы с TSS имеется отдельный регистр (в котором лежит TSS текущей программы; в TSS сохраняются все регистры при смене задачи)

# [Модели памяти в защищённом режиме. Регистры управления памятью. Страничное преобразование.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-33%5D-%D0%9C%D0%BE%D0%B4%D0%B5%D0%BB%D0%B8-%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D0%B8-%D0%B2-%D0%B7%D0%B0%D1%89%D0%B8%D1%89%D1%91%D0%BD%D0%BD%D0%BE%D0%BC-%D1%80%D0%B5%D0%B6%D0%B8%D0%BC%D0%B5.-%D0%A0%D0%B5%D0%B3%D0%B8%D1%81%D1%82%D1%80%D1%8B-%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F-%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C%D1%8E.-%D0%A1%D1%82%D1%80%D0%B0%D0%BD%D0%B8%D1%87%D0%BD%D0%BE%D0%B5-%D0%BF%D1%80%D0%B5%D0%BE%D0%B1%D1%80%D0%B0%D0%B7%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5.)

## **Модели памяти**

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



* (из интернета) модель памяти в режиме V86. С точки зрения программиста эта модель памяти работает точно также как в обычном реальном режиме. Т.е. память делится на сегменты по 64 Кбайт, ячейки внутри которых адресуются с помощью двух слов, записываемых в виде СЕГМЕНТ:СМЕЩЕНИЕ, максимальная адресуемая память 1 Мбайт и пр. Однако в режиме V86 выполняются все проверки защиты защищенного режима, из-за чего в некоторых случаях не будут работать некоторые инструкции.

### **Преимущества страничной модели:**

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

## **Управление памятью в x86**

<https://frolov-lib.ru/books/bsp/v13/ch2_1.htm>

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

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

## **Регистры управления памятью**

* GDTR: (Global Descriptor Table Register) 6-байтный регистр, содержит 32-битный линейный адрес начала таблицы глобальных дескрипторов (GDT) и 16-битный размер (лимит, уменьшенный на 1);
* IDTR: (Interrupt Descriptor Table Descriptor; то есть в защищенном режиме таблица векторов прерываний начинается с некоторого произвольного адреса) 6-байтный регистр, содержит 32-битный линейный адрес начала таблицы глобальных дескрипторов обработчиков прерываний (IDT) и 16-битный размер (лимит, уменьшенный на 1);
* LDTR: (Local Descriptor Table Register) 10-байтный регистр, содержит 16-битный селектор для GDT и весь 8-байтный дескриптор из GDT, описывающий текущую таблицу локальных дескрипторов;
* TR: (Task Register) 10-байтный регистр, содержит 16-битный селектор для GDT и весь 8-байтный дескриптор из GDT, описывающий TSS текущей задачи.

## **Страничное преобразование**

- преобразование линейного адреса в физический

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

# [Процессоры x86-64. Регистры. Режимы работы.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-34%5D-%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80%D1%8B-x86-64.-%D0%A0%D0%B5%D0%B3%D0%B8%D1%81%D1%82%D1%80%D1%8B.-%D0%A0%D0%B5%D0%B6%D0%B8%D0%BC%D1%8B-%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D1%8B.)

64 разрядные. AMD – c 2001, Intel – c 2003

## **Режимы работы**

* Legacy mode - совместимость с 32-разрядными процессорами;
* Long mode – 64-разрядный режим с частичной поддержкой 32-разрядных программ (32разрядный код не должен пересекаться с 64разрядным). Рудименты V86 и сегментной модели памяти упразднены (DOS программы нельзя теперь запустить, только на 32разрядной системе можно запустить).

## **Регистры**

* Целочисленные 64-битных регистры общего назначения - RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP;
* Новые целочисленные 64-битных регистры общего назначения R8 - R15;
* 64-битный указатель RIP и 64-битный регистр флагов RFLAGS.

## 

## **Соглашения о вызовах**

Соглашение о вызове — формализация правил вызова подпрограмм, которое должно

включать:

* способ передачи параметров;
* способ возврата результата из функции;
* способ возврата управления.

Соглашения о вызовах определяются в рамках отдельных языков высокого уровня, а

также - различных программных API, в т. ч. API операционных систем

# [Математический сопроцессор. Типы данных. Представление вещественных чисел.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-35%5D-%D0%9C%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9-%D1%81%D0%BE%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80.-%D0%A2%D0%B8%D0%BF%D1%8B-%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85.-%D0%9F%D1%80%D0%B5%D0%B4%D1%81%D1%82%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5-%D0%B2%D0%B5%D1%89%D0%B5%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D1%8B%D1%85-%D1%87%D0%B8%D1%81%D0%B5%D0%BB.)

## **Математический сопроцессор**

Отдельное опциональное устройство на материнской плате, с 80486DX встроен в процессор.

## **Типы данных**

* Целое слово (16 бит);
* Короткое целое (32 бита);
* Длинное слово (64 бита);
* Упакованное десятичное (80 бит);
* Короткое вещественное (32 бита);
* Длинное вещественное (64 бита);
* Расширенное вещественное (80 бит).

## **Представление вещественных чисел**

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

# [Математический сопроцессор. Регистры.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-36%5D-%D0%9C%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9-%D1%81%D0%BE%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80.-%D0%A0%D0%B5%D0%B3%D0%B8%D1%81%D1%82%D1%80%D1%8B.)

### **В сопроцессоре доступно 8 80-разрядных регистров (R0..R7).**

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

# [Математический сопроцессор. Особые числа.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-37%5D-%D0%9C%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9-%D1%81%D0%BE%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80.-%D0%9E%D1%81%D0%BE%D0%B1%D1%8B%D0%B5-%D1%87%D0%B8%D1%81%D0%BB%D0%B0.)

## **Сопроцессор поддерживает особые числа:**

* Положительная бесконечность: знаковый - 0, матнисса - нули, экспонента - единицы
* Отрицательная бесконечность: знаковый - 1, мантисаа - 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 нуля, чем наименьшее представимое нормальное число

Бесконечности возникают при делении бесконечности (или числа) на ноль.

NaN делятся на тихие и сигнальные.

***Тихий*** - не приводит к исключению (арифмитической ошибки нет, но получить число без округления нельзя)

***Сигнальный*** - переполнения в большую меньшую сторону (при делении на ноль иногда)

***Денормализованные числа*** - такие, которые не укладываются в заданный формат представления числа и позволяют хранить еще меньшие числа (в экспоненте все нули - специальное значение), мантисса считается умноженной на 2 в отрицательной степени, которая еще меньше, чем минимальное значение экспоненты. Перевод в денормализованные числа может производится аппаратно при получении очень малых значений. Их обработка может производится дольше, чем обработка нормализованных чисел.

**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

# [Математический сопроцессор. Классификация команд.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-38%5D-%D0%9C%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9-%D1%81%D0%BE%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80.-%D0%9A%D0%BB%D0%B0%D1%81%D1%81%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F-%D0%BA%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4)

Все команды сопроцессора оперируют регистрами стека сопроцессора. Если операнд

в команде не указывается, то по умолчанию используется вершина стека сопроцессора

(логический регистр st(0)). Если команда выполняет действие с двумя операндами по

умолчанию, то эти операнды – регистры st(0) и st(1).

### **1. Команды пересылки данных:**

* команды взаимодействия со стеком (загрузка - выгрузка вещественного числа, целого, BCD (упакованного); смена мест регистров)
* FLD - загрузить вещественное число из источника (переменная или ST(n)) в стек. Номер вершины в SR увеличивается
* FST/FSTP - скопировать/считать число с вершины стека в приёмик
* FILD - преобразовать целое число из источника в вещественное и загрузить в стек
* FIST/FISTP - преобразовать вершину в целое и скопировать/считать в приёмник
* FBLD, FBSTP - загрузить/считать десятичное BCD-число
* FXCH - обменять местами два регистра (вершину и источник) стека

### **2. Арифметические команды:**

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

### **3. Команды сравнений:**

* COM, FCOMP, FCOMPP - сравнить и вытолкнуть из стека
* FUCOM, FUCOMP, FUCOMPP - сравнить без учёта порядков и вытолкнуть
* FICOM, FICOMP, FICOMP - сравнить целые
* FCOMI, FCOMIP, FUCOMI, FUCOMIP (P6)

устанавливает биты ZF, PF, CF регистра [EFLAGS](https://prog-cpp.ru/asm-cpu/#eflags) в соответствии с таблицей.

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

### **4. Трансцедентные операции сопроцессора:**

* FSIN
* FCOS
* FSINCOS
* FPTAN
* FPATAN
* F2XM1 – 2^x -1
* FYL2X, FYL2XP1 – y\*log2 x, y\*log2 (x+1)

sin (fsin), cos(fsin) - принимают значение в радианах в некотором диапазоне.

tg, arctg, 2^x - 1, y \* log2x...

### **5. Константы FPU:**

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

### **6. Команды управления:**

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

**Команда CPUID (с 80486)**

Идентификация процессора CPU, предназначена для считывания

программным обеспечением информации о продавце, семействе, модели и

поколении процессора, а также специфической для процессора

дополнительной информации (поддерживаемые наборы команд, размеры

буферов, кэшей, разнообразные расширения архитектуры и т.п.)

Перед выполнением команды CPUID в регистр [EAX](https://www.club155.ru/x86internalreg-common) должно помещаться

входное значение, которое и указывает — какую информацию необходимо

выдать.

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

### **ДОПОЛНИТЕЛЬНАЯ ИНФОРМАЦИЯ ПО FPU, ДЕТАЛЬНО НА ЭКЗАМЕНЕ НЕ БУДУТ СПРАШИВАТЬ.**

**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

# [Расширение MMX. Назначение. Типы данных.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-39%5D-%D0%A0%D0%B0%D1%81%D1%88%D0%B8%D1%80%D0%B5%D0%BD%D0%B8%D0%B5-MMX.-%D0%9D%D0%B0%D0%B7%D0%BD%D0%B0%D1%87%D0%B5%D0%BD%D0%B8%D0%B5.-%D0%A2%D0%B8%D0%BF%D1%8B-%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85.)

### **(1997, Pentium MMX)**

### 

Расширeние, которое было встроено для увеличения

эффективности обработки больших потоков данных

(изображение, звук, видео..) - простые операции над

массивами однотипных чисел

### 

### **8 64-битных регистров MM0..MM7 - мантиссы регистров FPU. При записи в MMn экспонента и знаковый бит заполняются единицами (отрицательная бесконечность получается)**

### **Пользоваться одновременно и FPU, и MMX не получится, требуется FSAVE+FRSTOR**

## **Типы данных MMX:**

* учетверённое слово (64 бита);
* упакованные двойные слова (2);
* упакованные слова (4);
* упакованные байты (8).
* Команды MMX перемещают упакованные данные в память или обычные регистры целиком, но арифметические и логические операции выполняют поэлементно.
* насыщение - замена переполнения/антипереполнения превращением в максимальное/минимальное значение

# [Расширение MMX. Регистры. Классификация команд.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-40%5D-%D0%A0%D0%B0%D1%81%D1%88%D0%B8%D1%80%D0%B5%D0%BD%D0%B8%D0%B5-MMX.-%D0%A0%D0%B5%D0%B3%D0%B8%D1%81%D1%82%D1%80%D1%8B.-%D0%9A%D0%BB%D0%B0%D1%81%D1%81%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F-%D0%BA%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4.)

## **8 64-битных регистров MM0..MM7 - мантиссы регистров FPU. (то есть работает с целыми) При записи в MMn экспонента и знаковый бит заполняются единицами**

## **Пользоваться одновременно и FPU, и MMX не получится, требуется FSAVE+FRSTOR**

## **Насыщение - замена переполнения/антипереполнения превращением в максимальное/минимальное значение (Светлый цвет + светлый цвет максимум равно белый (но не будет переполнения и темного цвета))**

## **Классификация команд:**

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

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



### **1. Команды пересылки данных:**

* пересылка двойных/учетверенных слов;
* упаковка со знаковым насыщением слов (приемник -> младшая половина приемника, источник -> старшая половина приемника, в случае наличия значащих разрядов в отбрасываемых частях происходит насыщение);
* упаковка слов с беззнаковым насыщением, распаковка и объединение старших элементов источника и приемника через 1

• MOVD, MOVQ - пересылка двойных/учетверённых слов

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

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

• PACKUSWB - упаковка слов в байты с беззнаковым насыщением.

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

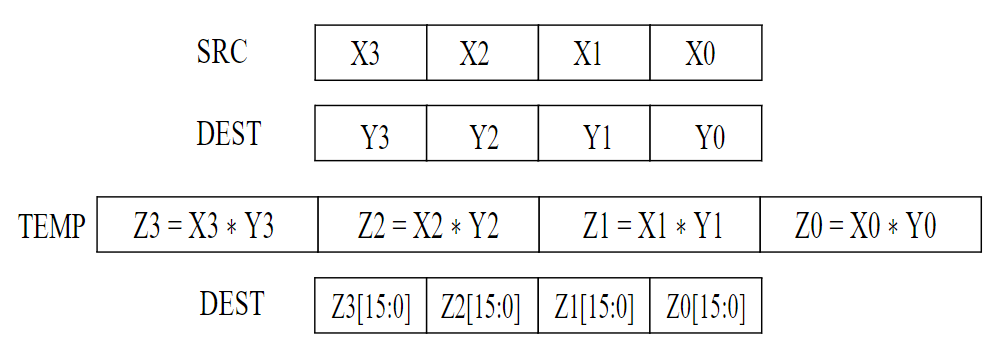
• PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ –

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

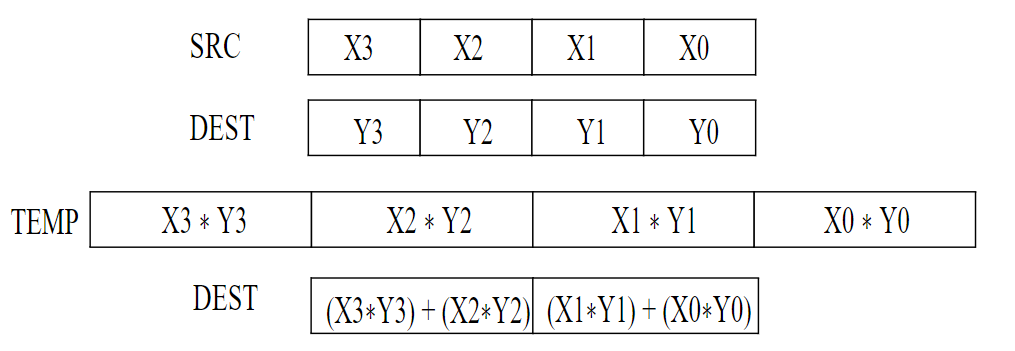
### **2. Арифметические операции:**

* поэлментное сложение (перенос игнорируется, побайтовое сложение)
* сложение/вычитание с насыщением
* беззнаковое сложение с насыщением
* вычитание (заем игнорируется)
* вычитание с насыщением
* старшое/младшее умножение (сохраняет старшую или младшую части результата в приемник)
* умножение и сложение (перемножает 4 слова, затем попарно складывает произведения двух старших и двух младших)
* PADDB, PADDW, PADDD - поэлементное сложение, перенос игнорируется
* PADDSB, PADDSW - сложение с насыщением
* PADDUSB, PADDUSW - беззнаковое сложение с насыщением
* PSUBB, PSUBW, PDUBD - вычитание, заём игнорируется
* PSUBSB, PSUBSW - вычитание с насыщением
* PSUBUSB, PSUBUSW - беззнаковое вычитание с насыщением
* PMILHW, PMULLW - старшее/младшее умножение (сохраняет старшую или младшую части результата в приёмник) (почему?)

(low)



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



### **3. Команды сравнения:**

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

### **4. Логические операции:**

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

### **5. Сдвиговые операции:**

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

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

# [Расширение SSE. Назначение. Типы данных. Регистры.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-41%5D-%D0%A0%D0%B0%D1%81%D1%88%D0%B8%D1%80%D0%B5%D0%BD%D0%B8%D0%B5-SSE.-%D0%9D%D0%B0%D0%B7%D0%BD%D0%B0%D1%87%D0%B5%D0%BD%D0%B8%D0%B5.-%D0%A2%D0%B8%D0%BF%D1%8B-%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85.-%D0%A0%D0%B5%D0%B3%D0%B8%D1%81%D1%82%D1%80%D1%8B.)

Расширение SSE (Pentium III, 1999) - Решение проблемы параллельной работы с FPU

## **Регистры:**

* 8 128-разрядных регистров
* свой регистр флагов
* Основной тип - вещественные одинарной точности (32 бита, в 1 регистре 4 числа)
* Целочисленные команды работают с регистрами MMX

## **Команд больше чем в MMX, типы:**

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

Развитие: SSE2, SSE3..

## **Расширение AES**

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

Цель - ускорение шифрования по алгоритму AES

Команды:

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

# [Макроопределения. Назначение.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-42%5D-%D0%9C%D0%B0%D0%BA%D1%80%D0%BE%D0%BE%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F.-%D0%9D%D0%B0%D0%B7%D0%BD%D0%B0%D1%87%D0%B5%D0%BD%D0%B8%D0%B5.)

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

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

## **Из методички:**

### **Определения:**

1. Макроопределение - специальным образом оформленная последовательность предложений языка ассемблера, под управлением которой ассемблер (точнее, его часть, называемая макрогенератором или препроцессором) порождает макрорасширения макрокоманд.
2. Макрорасширение - последовательность предложений языка ассемблера (обыкновенных директив и команд), порождаемая макрогенератором при обработке макрокоманды под управлением макроопределения и вставляемая в исходный текст программы вместо макрокоманды.
3. Макрокоманда (или макровызов) - предложение в исходном тексте программы, которое воспринимается макрогенератором как команда (приказ), предписывающая построить макрорасширение и вставить его на ее место.

* В макрокоманде могут присутствовать параметры, если они были описаны в макроопределении.
* Макроопределение без параметров однозначно определяет текст макрорасширения.
* Макроопределение с параметрами описывает множество (возможно, очень большое) возможных макрорасширений, а параметры, указанные в макрокоманде, сужают это множество до одного единственного макрорасширения.

### **Определение макроса в программе:**

имя MACRO параметры

...

ENDM

Пример:

load\_reg MACRO register1, register2

push register1

pop register2

ENDM

## **Сравнение макросов с подпрограммами**

### **Плюсы:**

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

### **Минусы:**

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

## **Замечания.**

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

# [Макроопределения. Директивы присваивания и отождествления.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-43%5D-%D0%9C%D0%B0%D0%BA%D1%80%D0%BE%D0%BE%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F.-%D0%94%D0%B8%D1%80%D0%B5%D0%BA%D1%82%D0%B8%D0%B2%D1%8B-%D0%BF%D1%80%D0%B8%D1%81%D0%B2%D0%B0%D0%B8%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F-%D0%B8-%D0%BE%D1%82%D0%BE%D0%B6%D0%B4%D0%B5%D1%81%D1%82%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F.)

## **Директива присваивания =**

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

Макроимя = Макровыражение

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

## **Директивы отождествления EQU, TEXTEQU**

Директива для представления текста и чисел:

Макроимя EQU нечисловой текст и не макроимя ЛИБО число

Макроимя EQU <Операнд>

Макроимя TEXTEQU Операнд

Пример:

X EQU [EBP+8]

MOV ESI,X

# [Макроопределения. Макрооперации.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-44%5D-%D0%9C%D0%B0%D0%BA%D1%80%D0%BE%D0%BE%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F.-%D0%9C%D0%B0%D0%BA%D1%80%D0%BE%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%B8.)

* % - вычисление выражение перед представлением числа в символьной форме (пример ниже - из методички)
  + MP\_REC MACRO P
  + MOV AX,P
  + IF P
  + MP\_REC %(P-1) ;;перед записью в МРасш-ние вычислить P-1
  + ENDIF
  + ENDM
* <> - подстановка текста без изменений (полезно, когда есть вероятность пересечения имени какого-либо макроса с простым (или не очень) текстом, который мы хотим вставить)
* & - склейка текста (A&B ==> AB, если параметры - макропараметры, то они склеятся)
* ! - считать следующий символ текстом, а не знаком операции (A!&B ==> A&B)
* ;; - исключение строки из макроса (После препроцессора эта строчка исчезнет (если одна ";", то комментарий остается); Дословно из методички: "текст не выносится в макрорасширение")

# [Макроопределения. Блоки повторения.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-45%5D-%D0%9C%D0%B0%D0%BA%D1%80%D0%BE%D0%BE%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F.-%D0%91%D0%BB%D0%BE%D0%BA%D0%B8-%D0%BF%D0%BE%D0%B2%D1%82%D0%BE%D1%80%D0%B5%D0%BD%D0%B8%D1%8F.)

## **REPT**

Повтор фиксированное число раз

REPT <число>

...

ENDM

## **IRP или FOR (конкретное имя зависит от компилятора)**

Подстановка фактических параметров по списку на место формального

IRP form,<fact\_1[,fact\_2,...]>

...

ENDM

## **IRPC или FORC (конкретное имя зависит от компилятора)**

Подстановка символов строки на место формального параметра

IRPC form,fact

...

ENDM

## **WHILE**

Классический цикл while

WHILE cond

...

ENDM

## **Примеры (взяты из методички)**

### **REPT**

*Pезеpвиpование 3-х байтов с начальными значениями 0, 3, 6*

A=0

MB0 LABEL BYTE

REPT 3

DB A

A=A+3

ENDM

### **IRP (FOR)**

*Определение переменных A0, A1, A2, A3 с начальными значениями 0,1,2,3 соответственно*

IRP X,<0,1,2,3> ;;параметры - числа

A&X DB X

ENDM

### **IRPC (FORC)**

*Описание переменных полей данных с начальными значениями 'A', 'B', 'C' соответственно*

IRPC X,<"ABC">

DB '&X&'

ENDM

### **WHILE (что-то очень загроможденное, без изменений взятое из методички. здесь нас больше интересует то, как в макросе можно менять параметр и проверять условие)**

*Стандартные макрофункция @SubStr и директива SubStr могут порождать множество подстрок типа text с числовыми и нечисловыми значениями, причём при одних и тех же значениях параметров директива SubStr определит (переопределит) макропеременную типа text, а макрофункция @SubStr вернёт значение, совпадающее со значением макропеременной. Следующий вложенный цикл позволяет перебрать и вывести значения подмножеств строки 1234*

j=1

while j LE 4

i=1

WHILE i le 5-j

names SubStr <ABCD>,i,j

%ECHO 'names SubStr <ABCD>,i,j' out: names , i, j

%ECHO '@SubStr (ABCD,i,j)' out: @SubStr (ABCD,i,j)

i=i+1

endM

j=j+1

endm

# [Макроопределения. Директивы условного ассемблирования.](https://github.com/mRrvz/bmstu-asm/wiki/%5B%D0%AD%D0%9A%D0%97%D0%90%D0%9C%D0%95%D0%9D-46%5D-%D0%9C%D0%B0%D0%BA%D1%80%D0%BE%D0%BE%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F.-%D0%94%D0%B8%D1%80%D0%B5%D0%BA%D1%82%D0%B8%D0%B2%D1%8B-%D1%83%D1%81%D0%BB%D0%BE%D0%B2%D0%BD%D0%BE%D0%B3%D0%BE-%D0%B0%D1%81%D1%81%D0%B5%D0%BC%D0%B1%D0%BB%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F.)

IF:

IF c1

…

ELSEIF c2

…

ELSE

…

ENDIF

* IFB <par> - истинно, если параметр не определён (то есть фактический параметр par не был задан в МКоманде)
* IFNB <par> - истинно, если параметр определён
* IFIDN <s1>,<s2> - истинно, если строки совпадают
* IFDIF <s1>,<s2> - истинно, если строки разные
* IFDEF/IFNDEF <name> - истинно, если имя объявлено/не объявлено

# Макроопределения. Директивы управления листингом

* Листинг - файл, формируемый компилятором и содержащий текст ассемблерной программы, список определённых меток, перекрёстных ссылок и сегментов.
* TITLE (только 1 раз), SUBTTL - заголовок, подзаголовок на каждой странице
* PAGE высота, ширина
* NAME - имя программы
* .LALL - включение полных макрорасширений, кроме ;;
* .XALL - по умолчанию
* .SALL - не выводить тексты макрорасширений
* .NOLIST - прекратить вывод листинга

Комментарии

comment @

… многострочный текст ...

@