

LEA (Load Effective Address) - вычисление эффективного адреса

LEA <приёмник>, <источник>

Вычисляет эффективный адрес источника и помещает его в приёмник.

Эффективный (текущий) адрес - это БАЗА + СМЕЩЕНИЕ + ИНДЕКС , где БАЗА - это базовый адрес, находящийся в регистре (при 16-разрядной адресации могут использоваться только регистры BX или BP); СМЕЩЕНИЕ - это константа (число со знаком), заданная в команде; ИНДЕКС - значение индексного регистра (при 16-разрядной адресации могут использоваться только регистры SI или DI). Любая из частей эффективного адреса может отсутствовать.

Позволяет вычислить адрес, описанный сложным методом адресации (да и просто его загрузить).

Оператор OFFSET позволяет определить смещение только при компиляции, и в отличие от него команда LEA может сделать это во время выполнения программы. Хотя в остальных случаях обычно вместо LEA используют MOV и OFFSET, то есть

LEA ПРИЁМНИК, ИСТОЧНИК - это то же самое, что и MOV ПРИЁМНИК, offset ИСТОЧНИК

Инструкция LEA часто используется для арифметических операций, таких как умножение и сложение. Преимущество такого способа в том, что команда LEA занимает меньше места, чем команды арифметических операций. Кроме того, в отличие от последних, она не изменяет флаги. Примеры:

```
;Умножение с помощью LEA  
MOV BX, 8  
LEA BX, [BX + BX * 4] ;Не поддерживается emu8086
```

```
;Сложение с помощью LEA  
MOV BX, 8  
LEA BX, [BX + 16] ;BX = BX + 16 = 8 + 16
```

Следующий вопрос: [Команды десятичной арифметики](#).

Предыдущий вопрос: [Команда XLAT/XLATB](#).

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

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

За один раз обрабатывается один байт (слово).

MOVS/MOVSB/MOVSW <приёмник>, <источник>

Когда в тексте программы встречается команда MOVS, компилятор вычисляет размерность ее operandов и на основании вычислений подставляет на ее место одну из реальных команд процессора MOVSB, MOVSW или MOVSD (эти команды - без операторов)

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, в зависимости от размера operandов), 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>

Команда SCAS "просматривает" строку на нахождение в ней определенного значения байта или слова. Эта команда сравнивает содержимое области памяти (адресуемой регистрами ES:DI) с содержимым регистра AL или 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.

Предикторы: 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 применяться 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

Следующий вопрос: [Стек. Регистры, связанные со стеком. Команды записи/извлечения из стека.](#)

Предыдущий вопрос: [Команда организации цикла.](#)

CALL и RET

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

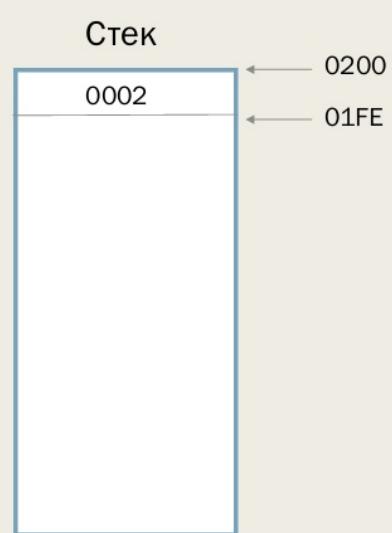
Сохраняет адрес следующей команды в стеке (уменьшает SP и записывает по его адресу IP либо CS:IP, в зависимости от размера аргумента).

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

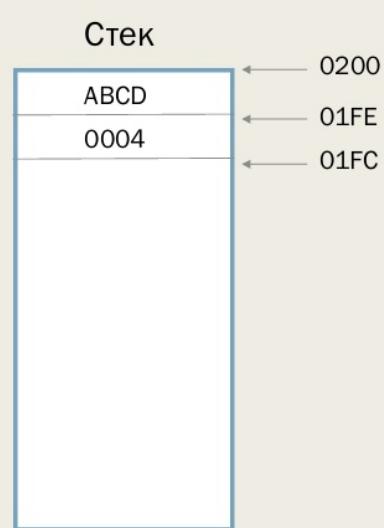
Если указать operand, то можно очистить стек для очистки стека от параметров (<число> будет прибавлено к SP).

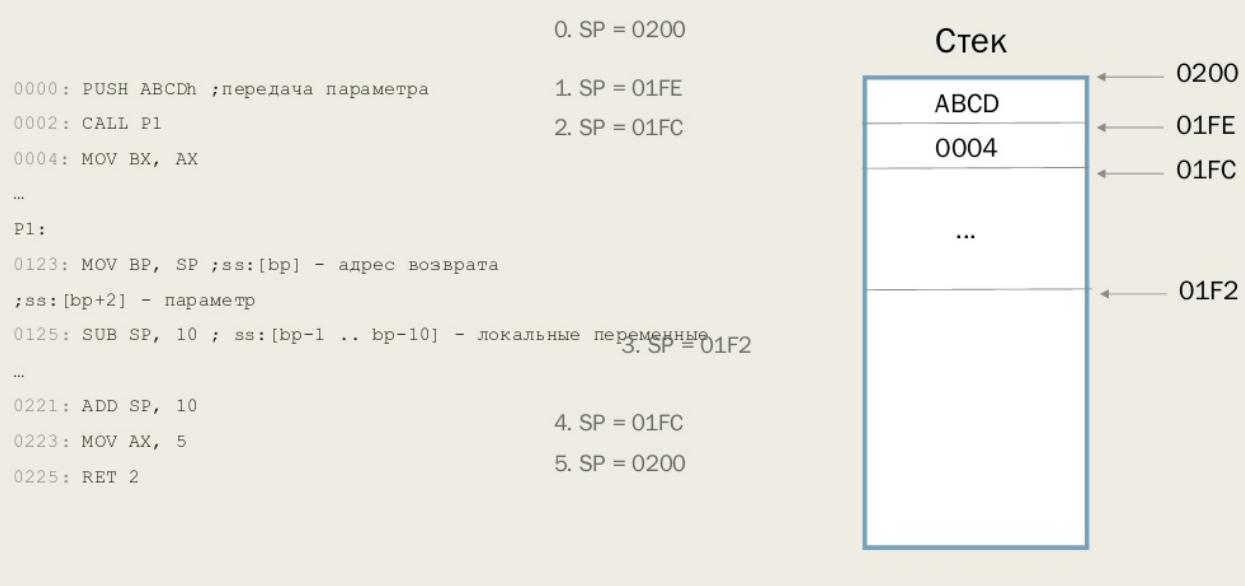
Примеры использования

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



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





Следующий вопрос: Прерывания. Назначение, виды прерываний. Таблица векторов прерываний.

Предыдущий вопрос: Стек. Регистры, связанные со стеком. Команды записи/извлечения из стека.

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

Это ваше жесть какая-то иррациональная...

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 результата 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 $AL = AL + (\text{imm8} * AH)$, 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'
```

Следующий вопрос: [Команды сдвига](#).

Предыдущий вопрос: [Команда LEA](#).

Организация циклов

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).

Следующий вопрос: [Строковые операции. Префиксы повторения.](#)

Предыдущий вопрос: [Команды сдвига.](#)

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.

Следующий вопрос: [Процессор 80386. Разрядность, регистры.](#)

Предыдущий вопрос: [Прерывания. Назначение, виды прерываний. Таблица векторов прерываний.](#)

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 процессоре, например, деление на ноль (номер вектора - 0), переполнение (номер - 4);
- программные - вызванные командой `int`.

Таблица векторов прерываний (Interrupt Descriptor Table, IDT)

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.

(`int 21h` - вызов сервиса DOS, 20-5F зарезервированы под DOS)

Следующий вопрос: Срабатывание прерывания. Обработчик прерывания в реальном режиме. Возврат из обработчика прерывания.

Предыдущий вопрос: Стек. Использование при вызове подпрограмм. Команды вызоваподпрограммы и возврата.

Защищенный режим

В 80386 случилась первая полноценная реализация защищенного режима. (В 80286 тоже был, но сильно отличался способом работы с памятью, т.к. еще не было страничной организации памяти)

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.

Кольцо защищает всего 4. Чем ниже уровень защиты, тем больше возможностей.

Кольцо 0 - всё доступно, режим SuperVisor.

Кольцо 1 - режим HyperVisor для поддержки виртуализации.

Кольцо 2 - режим SMM (системного управления)

Кольцо 3 - режим AMT (доступен чипсетам поддерживающим дополнительный микропроцессор ARCh4)

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)

Следующий вопрос: [Модели памяти в защищённом режиме. Регистры управления памятью. Страницочное преобразование.](#)

Предыдущий вопрос: [Процессор 80386. Разрядность, регистры.](#)

Режимы работы

- **Legacy mode** - совместимость с 32-разрядными процессорами;
- **Long mode** – 64-разрядный режим с частичной поддержкой 32-разрядных программ (32-х разрядный код не должен пересекаться с 64-х разрядным). Рудименты V86 (виртуальный режим для поддержки работы с 8086) и сегментной модели памяти упразднены (DOS программы нельзя теперь запустить, только на 32-х разрядной 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**.

Следующий вопрос: [Математический сопроцессор. Типы данных. Представление вещественных чисел.](#)

Предыдущий вопрос: [Модели памяти в защищённом режиме. Регистры управления памятью. Страницное преобразование.](#)

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 $-\infty$ (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.

Следующий вопрос: [Команда организации цикла.](#)

Предыдущий вопрос: [Команды десятичной арифметики.](#)

Математический сопроцессор

Математический сопроцессор – устройство в виде отдельной макросхемы (до 80486) для обработки чисел с плавающей запятой.

FPU - Floating Point Unit. Обычному процессору для работы с числами с плавающей запятой (вещественными), требуется процедуры поддержки подобного формата. FPU же поддерживает работу с вещественными числами на уровне примитивов. FPU имеет кольцевой стек из 8 регистров по 80 бит каждый R0...R7 (обращение как $st(n)$, где n - номер в стеке)

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

x87 - набор инструкций для работы с математическими вычислениями (подмножество x86 архитектуры).

Типы данных

- Целое слово (16 бит);
 - Короткое целое (32 бита);
 - Длинное слово (64 бита);
 - Упакованное десятичное (80 бит); (упакованный BSD-формат, каждый байт - целое число от 0 до 99, где каждые 4 бита занимает цифра/знак, т.е. $1001b = 9$, $1101b = -$, $1100 = +$)
 - Короткое вещественное (32 бита);
 - Длинное вещественное (64 бита);
 - Расширенное вещественное (80 бит).

Представление вещественных чисел

- Нормализованная форма представления числа ($1 \dots * 2^{\exp}$);
 - Экспонента увеличена на константу для хранения в положительном виде;
 - Формат представления 0,625 в коротком вещественном типе:
 - Знак Мантисса*ОснованиеСистемыСчисления^{Порядок} ([знак][порядок][мантисса])
 - 0 01111110 01000000000000000000000000000000
 - Все вычисления FPU - в расширенном 80-битном формате.

Следующий вопрос: Математический сопроцессор. Регистры.

Предыдущий вопрос: Процессоры x86-64. Регистры. Режимы работы.

37. Математический сопроцессор. Особые числа.

Сопроцессор поддерживает особые числа:

Математический сопроцессор - устройство в виде отдельной макросхемы (до 80486) для обработки чисел с плавающей запятой.

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

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

NaN делятся на тихие и сигнальные.

Тихий - не приводит к исключению (арифметической 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 дальше, чем обработка нормализованных чисел.

Следующий вопрос: [Математический сопроцессор. Классификация команд.](#)

Предыдущий вопрос: [Математический сопроцессор. Регистры.](#)

Стек. Назначение, примеры использования.

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 подпрограмм, адрес возврата при вызове процедур и прерываний.

Регистр SP — указывает на вершину стека

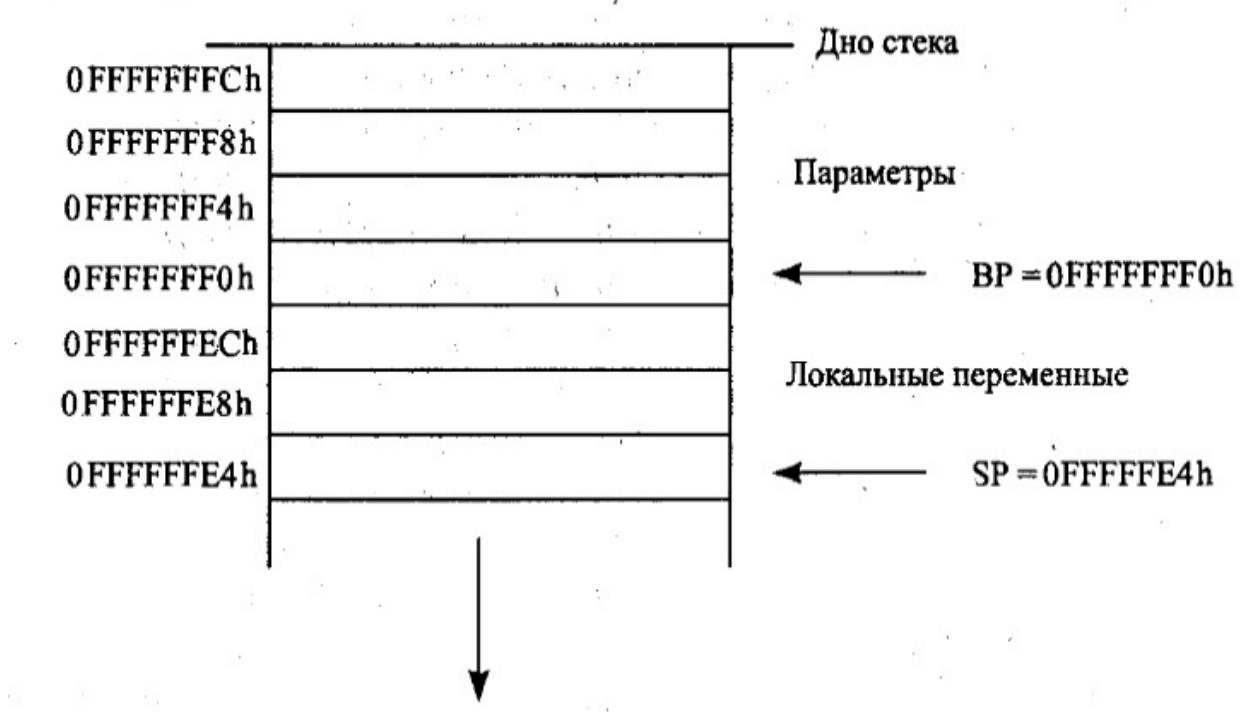
В x86 стек "растёт вниз", в сторону уменьшения адресов (от максимально возможно адреса). При запуске программы SP указывает на конец сегмента.

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 переменых, SP изменится и таким образом можно будет считывать переменные напрямую из стека (их смещения записуются 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)

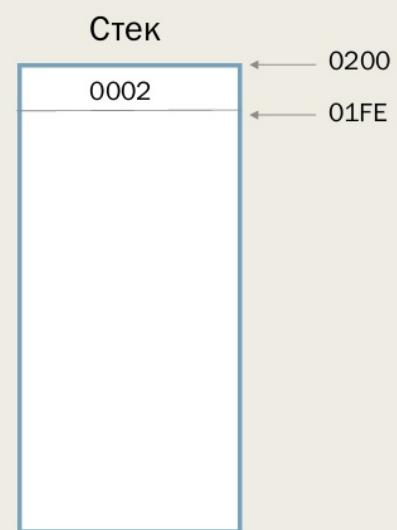
POPA — загрузить регистры из стека (SP игнорируется)

PUSHF — поместить в стек флаги.

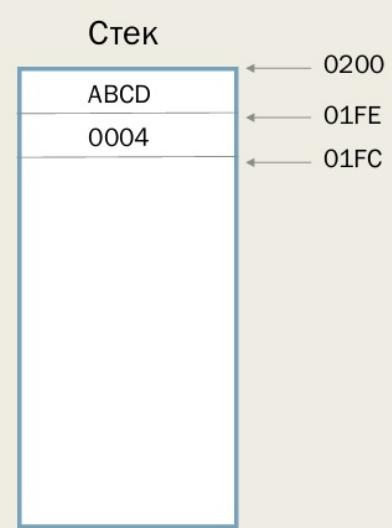
POPF — загрузить флаги из стека

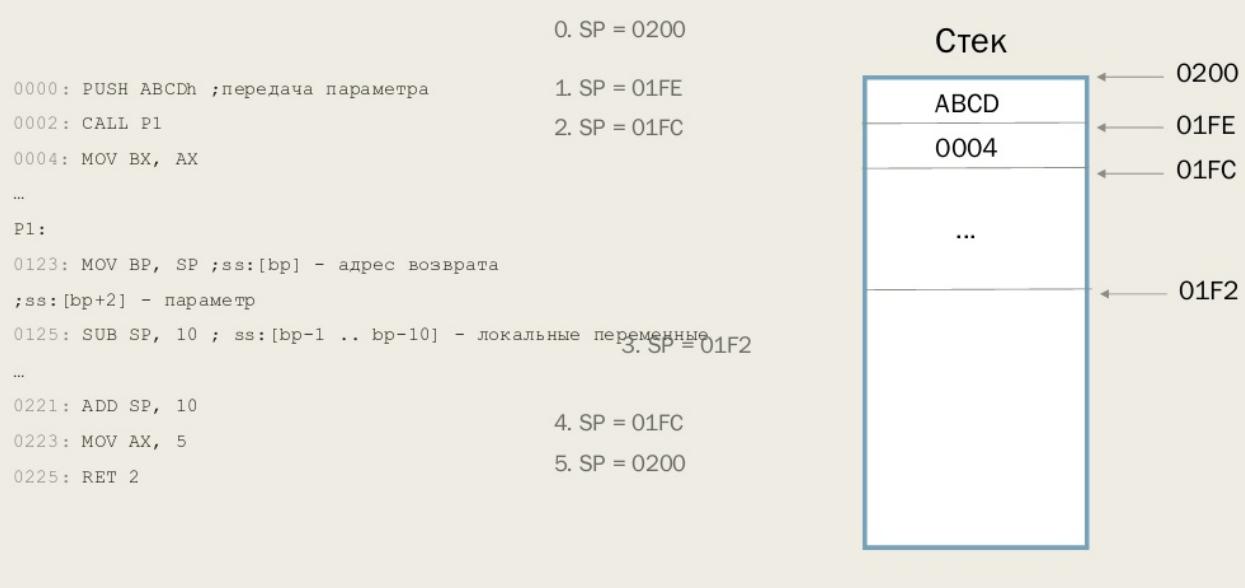
Примеры использования

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



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





Следующий вопрос: Стек. Использование при вызове подпрограмм. Команды вызова подпрограммы и возврата.

Предыдущий вопрос: Строковые операции. Префиксы повторения.

41. Расширение SSE. Назначение. Типы данных. Регистры.

SSE (Streaming SIMD (Single Instruction, Multiple Data) Extensions) - инструкции, пришедшие на смену MMX/

Регистры:

- 8 128-разрядных регистров (XMM0..7)
- Свой регистр флагов (MXCSE)

Основной тип - вещественные одинарной точности (32 бита, в 1 регистре 4 числа), т.е. 128-битный упакованный тип из 4-х 32-х битных с плавающей точкой

Целочисленные команды работают с регистрами MMX

Команд больше чем в MMX, типы:

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

Следующий вопрос: [Макроопределения. Назначение.](#)

Предыдущий вопрос: [Расширение MMX. Регистры. Классификация команд.](#)

40. Расширение MMX. Регистры. Классификация команд.

8 64-битных регистров MM0..MM7 - мантиссы регистров FPU. При записи в MMn экспонента и знаковый бит заполняются единицами (2 байта, знак и экспонента).

Пользоваться одновременно и FPU, и MMX не получится, решать проблему можно при помощи FSAVE и FRSTOR. Это происходит потому что MMX-команды при выполнении "портят" слово состояния (регистра SW) регистров FPU. (Этот регистр во многом схож с FLAGS).

FSAVE/FNSAVE - запись в память текущих значения из регистров стека FPU и инициализирует FPU через FINIT, FRSTOR - восстановить регистровый стек FPU из памяти

Насыщение - замена переполнения/антипереполнения (ситуация, когда результат операции с плавающей запятой становится настолько близким к нулю, что порядок числа выходит за пределы разрядной сетки, например $10^{-20} \cdot 10^{-30} = 10^{-50}$) превращением в максимальное/минимальное значение (Светлый цвет + светлый цвет максимум равно белый (но не будет переполнения и темного цвета))

Классификация команд:

1. Команды пересылки данных:

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

2. Арифметические операции:

- поэлементное сложение PADDB (перенос игнорируется, побайтовое сложение)
- сложение/вычитание с насыщением PADDSS , PSUBSB
- беззнаковое сложение/вычитание с насыщением PADDUSB , PSUBUSB
- умножение и сложение (перемножает 4 слова, затем попарно складывает произведения двух старших и двух младших) PMULHW , PMULLW

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)

4. Логические операции:

- логическое И PAND
- логическое ИЛИ POR
- логическое НЕ-И (штрих шеффера)
- XOR PXOR

5. Сдвиговые операции:

- логический влево PSLLW

- логический вправо PSRAW
- арифметический вправо

Следующий вопрос: [Расширение SSE](#). Назначение. Типы данных. Регистры.

Предыдущий вопрос: [Расширение MMX](#). Назначение. Типы данных.

45. Макроопределения. Блоки повторения.

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

Резервирование 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
```

Следующий вопрос: [Макроопределения. Директивы условного ассемблирования.](#)

Предыдущий вопрос: [Макроопределения. Макрооперации.](#)

Процессоры архитектуры 80386 стали 32-разрядные:

- регистры, кроме сегментных
- шина данных
- шина адреса ($2^{32} = 4\text{ГБ ОЗУ}$)

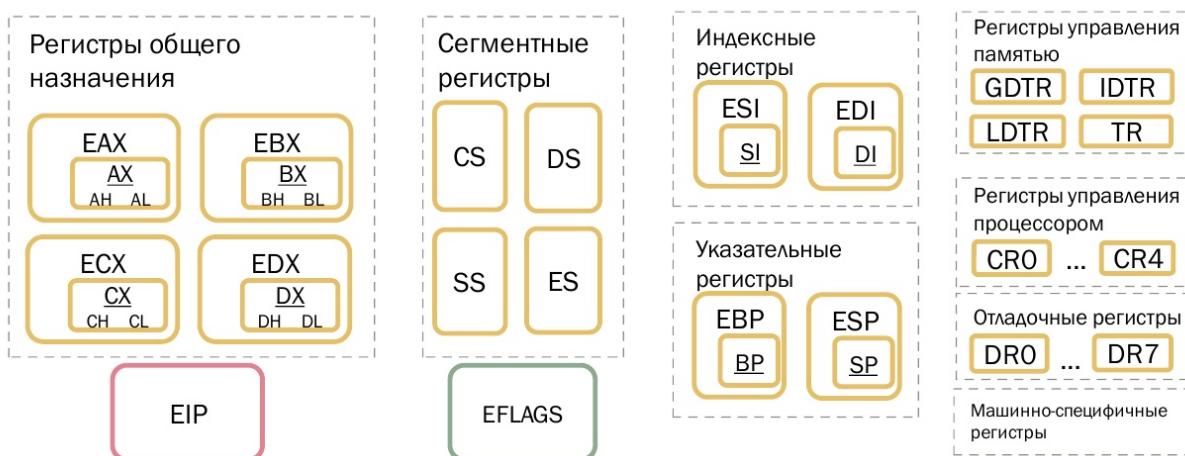
Память теперь делится на страницы по 4 кБайта

Регистры

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

Расширен регистр флагов (теперь EFLAGS): с 18 по 31 - зарезервированы Intel, добавлены VM (Virtual Mode - признак того, что задача выполняет программу под 8086) и RF (Resume Flag - временно отключает ошибки дебага)

Добавлены регистры поддержки работы в защищенном режиме (обеспечивание разделения доступа программ между собой, между программами и ОС и тд; эти регистры справа на картинке). Например добавлены GDTR, IDTR, LDTR и TR регистры для появившихся Descriptor Tables(как раз для новой страничной модели памяти (вопрос 32)). (Подробнее в 33 вопросе)



Следующий вопрос: Защищённый режим работы процессора. Многозадачность.

Предыдущий вопрос: Срабатывание прерывания. Обработчик прерывания в реальном режиме. Возврат из обработчика прерывания.

42. Макроопределения. Назначение.

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 макроопределений 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.

Следующий вопрос: [Макроопределения. Директивы присваивания и отождествления](#).

Предыдущий вопрос: [Расширение SSE. Назначение. Типы данных. Регистры](#).

39. Расширение MMX. Назначение. Типы данных.

Расширение, которое было встроено для увеличения эффективности обработки больших потоков данных (простые операции над массивами однотипных данных (звук, изображения, видеопоток))

Команды технологии MMX работают с 64-разрядными целочисленными данными, а также с данными, упакованными в группы (векторы) общей длиной 64 бита. Такие данные могут находиться в памяти или в восьми MMX-регистрах. Эти регистры называются MM0, MM1, :, MM7.

Типы данных MMX:

- учетверённое слово (quadword) - 64-х разрядное слово
- упакованные двойные слова (packed double word) - 2 32-х битных слова в 64-разрядном
- упакованные слова (packed word) - 4 16-ти битных слова в 64-х разрядном регистре.
- упакованные байты (packed bytes) - 8 байт в одном 64-х разрядном регистре.

Команды MMX перемещают упакованные данные в память или обычные регистры целиком, но арифметические и логические операции выполняют поэлементно.

Примеры команд

PADDB/PADDW/PADDD <регистр>, <регистр память> - команда складывают элементы данных (байты/слова/двойные слова) входного и выходного операнда.

Следующий вопрос: [Расширение MMX. Регистры. Классификация команд.](#)

Предыдущий вопрос: [Математический сопроцессор. Классификация команд.](#)

43. Макроопределения. Директивы присваивания и отождествления.

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
```

Следующий вопрос: [Макроопределения. Макрооперации.](#)

Предыдущий вопрос: [Макроопределения. Назначение.](#)

44. Макроопределения. Макрооперации.

В Ассемблере имеются макрооперации, обеспечивающие большую гибкость в определении макрокоманд.

- `%` - 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 MPасш-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")

[Дополнительные примеры в 45](#)

Следующий вопрос: [Макроопределения. Блоки повторения.](#)

Предыдущий вопрос: [Макроопределения. Директивы присваивания и отождествления.](#)

46. Макроопределения. Директивы условного ассемблирования.

```
IF:  
IF c1  
...  
ELSEIF c2  
...  
ELSE  
...  
ENDIF
```

- IFB <par> - истинно, если параметр не определён (то есть фактический параметр par не был задан в MКоманде)
- 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

Следующий вопрос: [Архитектура фон Неймана. Принципы фон Неймана](#)

Предыдущий вопрос: [Макроопределения. Блоки повторения.](#)

Модели памяти

- Плоская - код и данные используют одно и то же пространство
- Сегментная - сложение сегмента и смещения (используется в реальном режиме; знакома нам)
- Страницчная - 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.

Преимущества страницной 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

В защищенном режиме в 80386 команды ссылаются на сегменты указывая не их адрес, а дескриптор (описание) сегмента. Размер дескриптора - 8 байт. Все дескрипторы хранятся в специально отведенной области памяти - глобальной дескрипторной таблице.

Селектор - указатель на описание сегмента (номер дескриптора в таблице дескрипторов). Селекторы нужны чтобы:

- Не использовать 8-байтные регистры под (8-байтные) дескрипторы
- Чтобы обходиться всего 2 байтами размера (можно использовать в сегментных регистрах)
- Программа, использующая селектор не сможет изменить параметры сегмента (т.к. он находится в отдельной области памяти)
- В сегментных регистрах - селекторы состоят из:
 - 13-битный номер дескриптора;
 - 1-битный "признак" - какую таблицу использовать - глобальную или локальную (таблица текущей программы/ задачи);
 - 2-х битный уровень привилегий запроса 0-3
 - 0 - система
 - 3 - прикладная программа
 - 1-2 - где-то не используется, где-то используется, например, для драйверов
- При включённом страницном 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 защищенным 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.

Страницное преобразование

Страницная адресация - 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).

Следующий вопрос: [Процессоры x86-64. Регистры. Режимы работы.](#)

Предыдущий вопрос: [Защищённый режим работы процессора. Многозадачность.](#)

Математический сопроцессор. Регистры.

Математический сопроцессор - устройство в виде отдельной макросхемы (до 80486) для обработки чисел с плавающей запятой.

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 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

Следующий вопрос: [Математический сопроцессор. Особые числа.](#)

Предыдущий вопрос: [Математический сопроцессор. Типы данных. Представление вещественных чисел.](#)

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`

Следующий вопрос: [Команды условных переходов](#).

Предыдущий вопрос: [Способы адресации](#).

Архитектура фон Неймана



Архитектура фон Неймана — широко известный принцип совместного хранения команд и данных в памяти компьютера. В общем случае, подразумевают принцип хранения данных и инструкций в одной памяти.

(От решения частных вычислительных задач - к универсальным системам)

- Процессор состоит из блоков **УУ** и **АЛУ**
- УУ (Управляющее Устройство) - дискретный конечный автомат, управляет всеми частями компьютера. От управляющего устройства на другие устройства поступают сигналы «что делать», а от других устройств УУ получает информацию об их состоянии.
Структурно состоит из: дешифратора команд (операций), регистра команд, узла вычислений текущего исполнительного адреса, счётчика команд (**регистр IP**).
- АЛУ (Арифметико-Логическое Устройство) - под управлением УУ производит преобразование над данными (операндами). Разрядность операнда - длина машинного слова. (Машинное слово - машинно-зависимая величина, измеряемая в битах, равная разрядности регистров/шинам данных). Программы и данные вводятся в память из устройства ввода через арифметико-логическое устройство. Из арифметико-логического устройства результаты выводятся в память или устройство вывода.

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 пронумерованы.
5. Возможность условного перехода в процессе выполнения программы.

Следующий вопрос: Структурная схема ЭВМ. Виды памяти. Системная шина.

Предыдущий вопрос: Макроопределения. Директивы условного ассемблирования.

Классификация команд процессора x86

- Команды пересылки данных. Например, mov.
- Арифметические и логические команды. ADD (сложение), SUB (вычитание), MUL (умножение), div (деление), and (логическое И), or (лог. ИЛИ), xor (лог. взаимоисключающее ИЛИ) и т.д.
- Команды переходов. То есть, нарушение естественного (последовательного) выполнения команд. **Команды условного перехода** - когда есть какое-то условие (например, je - jump equals, jmp если равно). **Безусловного перехода** - без условия (например, jmp) (JMP сегмент:смещение/метка , флаги не меняет, не пишет в стек)
- Команды работы с подпрограммами. Вызов подпрограммы - call, возврат из неё - ret (который при компиляции превращается в retn/retf для ближнего/дальнего возврата соответственно, ближний вытаскивает из стека только IP, дальний - IP вместе с CS).
- Команды управления процессором. Например, NOP - no operation (ничего не делает). HALT - останавливает работу процессора пока не будет получено внешнее прерывание.

Следующий вопрос: [Команда пересылки данных.](#)

Предыдущий вопрос: [Выполнение программы. Машинный код. Исполняемые файлы. Язык ассемблера.](#)

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.

(Это канал, по которому процессор соединен с другими устройствами компьютера. К шине напрямую подключен только процессор, другие устройства компьютера подключены к ней через разнообразные контроллеры.)

С помощью шины происходит как обмен информацией, так и передача адресов, служебных сигналов.

Общая магистраль представлена совокупностью трех специализированных шин: шины данных, шины адреса и шины управления.

Шина данных предназначена для пересылки кодов обрабатываемых данных, а также машинных кодов команд между устройствами ЭВМ. По шине данных передается информация в микропроцессор и из него.

Шина адреса несет адрес (номер) той ячейки памяти или того порта ввода-вывода, который взаимодействует с микропроцессором.

На шину адреса микропроцессор выводит информацию о номере (адресе) той ячейки памяти или устройства, с которым он собирается производить обмен информацией.

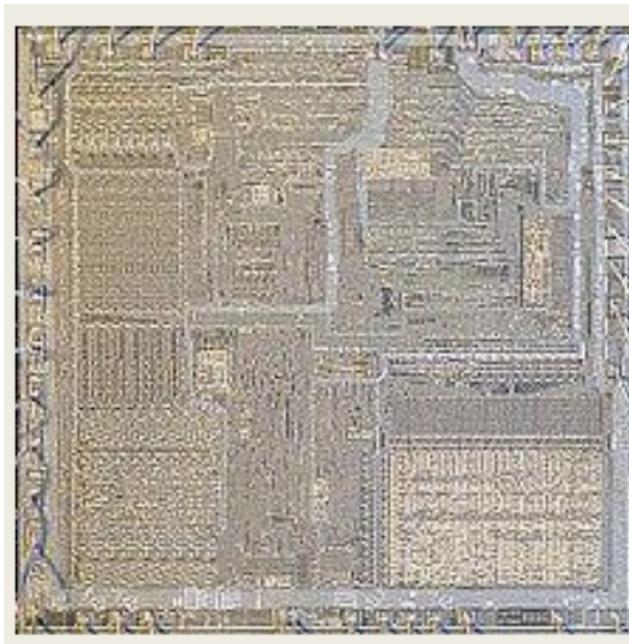
Шина управления несет сигналы управления, обеспечивающие правильное взаимодействие блоков микро ЭВМ друг с другом и с внешней средой.

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

Следующий вопрос: [Процессор 8086. Разрядность. Регистры.](#)

Предыдущий вопрос: [Архитектура фон Неймана. Принципы фон Неймана](#)

Процессор 8086



		MAX MODE	(MIN MODE)
GND	1	40	U_{CC}
AD14	2	39	AD15
AD13	3	38	A16/S3
AD12	4	37	A17/S4
AD11	5	36	A18/S5
AD10	6	35	A19/S6
AD9	7	34	BHE/S7
AD8	8	33	MN/MX
AD7	9	32	RD
AD6	10	8086 CPU	RQ/GT0 (HOLD)
AD5	11		RQ/GT1 (HLDA)
AD4	12		LOCK (WR)
AD3	13	28	S2 (M/I \bar{O})
AD2	14	27	S1 (DT/R)
AD1	15	26	S0 (DEN)
AD0	16	25	QS0 (ALE)
NMI	17	24	QS1 (INTA)
INTR	18	23	TEST
CLK	19	22	READY
GND	20	21	RESET

Разрядность

Разрядность: 16 бит. Производился с 1978 по 1990 годы.

(Незначительно изменённая версия процессора с 8-битной шиной данных, выпущена в 1979 году под названием Intel 8088)

Регистры

Всего в 8086 14 регистров: 4 регистра общего назначения (AX, BX, CX, DX), 2 индексных регистра (SI, DI), 2 указательных (BP, SP), 4 сегментных регистра (CS, SS, DS, ES), программный счётчик или указатель команды (IP) и регистр флагов (FLAGS, включает в себя 9 флагов).

Регистры общего назначения: AX (primary accumulator), BX (base), CX (counter), DX (accumulator, other).

Каждый из регистров имеет старшую и младшую часть, по 1 байту на каждый. AX состоит из AH и AL

Индексные регистры: SI (source index), DI (destination index)

Используются для индексации (например при работе со строками).

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 (instruction pointer).

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 (stack pointer) и 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]

Следующий вопрос: [Процессор 8086. Регистр флагов.](#)

Предыдущий вопрос: [Структурная схема ЭВМ. Виды памяти. Системная шина.](#)

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 зарезервированы при разработке процессора, но так и не были использованы.

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.

Следующий вопрос: [Процессор 8086. Шина адреса. Сегментная модель памяти.](#)

Предыдущий вопрос: [Процессор 8086. Разрядность. Регистры.](#)

Шина адреса в 8086

- Шина адреса - 20 бит, что позволяет адресовать 2^{20} (1мб) памяти, а не 2^{16} б.
- (Шина адреса несет адрес (номер) той ячейки памяти или того порта ввода-вывода, который взаимодействует с микропроцессором.)
- На шину адреса микропроцессор выводит информацию о номере (адресе) той ячейки памяти или устройства, с которым он собирается производить обмен информацией.)

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.

Реальный режим адресации процессора (real-address mode) — режим работы процессоров архитектуры x86, при котором используется сегментная адресация памяти, адрес ячейки памяти состоит из сегмента и смещения, а также любому процессу доступна вся память компьютера.

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

Следующий вопрос: [Выполнение программы. Машинный код. Исполняемые файлы. Язык ассемблера.](#)

Предыдущий вопрос: [Процессор 8086. Регистр флагов.](#)

Команда пересылки данных

MOV <приемник>, <источник>.

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

(MOV не может: писать в IP и CS, копировать данные напрямую между сегментными регистрами, копировать значение в регистр сегментов)

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. Выполняется либо над двумя регистрами, либо регистр + переменная.

Следующий вопрос: [Команды целочисленной арифметики](#).

Предыдущий вопрос: [Классификация команд процессора x86](#).

Команды целочисленной арифметики

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 (signed flag, равен старшему биту результата).

MUL <источник>

Умножение без знака.

- Источник - область памяти, РОН.
- Результат - AL/AX

Умножаются источник и AL/AX, в зависимости от размера источника. Результат помещается в AX либо DX:AX (в DX - старшее слово, в AX - младшее).

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

Следующий вопрос: [Команды побитовой арифметики](#).

Предыдущий вопрос: [Команда пересылки данных.](#)

Команды побитовой арифметики

- AND <приёмник>, <источник> - побитовое "И"
- OR <приёмник>, <источник> - побитовое "ИЛИ"
- XOR <приёмник>, <источник> - побитовое исключающее "ИЛИ"
- NOT <приёмник> - инверсия
- SHL <приёмник>, <счётчик> - сдвиг влево, <счетчик> раз (SAL - эквивалентная команда)
- SHR <приёмник>, <счётчик> - сдвиг вправо
- SAR <приёмник>, <счётчик> - сдвиг вправо с сохранением знакового бита
- ROR <приёмник>, <счётчик> - циклический сдвиг вправо (в CF только копируется, не участвует во вращении)
- RCR <приёмник>, <счётчик> - циклический сдвиг вправо, через флаг переноса (в флаг CF сдвигается старший бит)
- ROL <приёмник>, <счётчик> - циклический сдвиг влево (в CF только копируется, не участвует во вращении)
- RCL <приёмник>, <счётчик> - циклический сдвиг влево, через флаг переноса (в флаг CF сдвигается младший бит)
- TEST <число> <число> - как и AND производит побитовое умножение, но не записывает результат в какой либо регистр, а всего лишь поднимает флаги для каждого бита (ZF, SF, PF и сбрасывает OF и CF), то есть она имитирует выполнение инструкции AND.

<счетчик> - число или регистр CL.

Все эти команды меняют регистр **FLAGS**.

Следующий вопрос: [Команды передачи управления](#).

Предыдущий вопрос: [Команды целочисленной арифметики](#).

Директива - инструкция ассемблеру, влияющая на процесс компиляции и не являющаяся командой процессора. Обычно не оставляет следов в формируемом машинном коде.

Псевдокоманда - директива ассемблера, которая приводит к включению данных или кода в программу, но не 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(?)

Следующий вопрос: [Директива SEGMENT](#).

Предыдущий вопрос: [Структура программы на языке ассемблера. Модули. Сегменты.](#)

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) 2 байта|

|far (дальний)|в другой сегмент (со сменой CS) 4 байта|

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 = 1
JNE/JNZ	Не равно/не ноль	ZF = 0
JP/JPE	Есть чётность/чётное	PF = 1
JNP/JPO	Нет чётности/нечётное	PF = 0
JCXZ	CX = 0	

Команда	Описание	Состояние флагов для выполнения перехода	Знаковый
JB	Если ниже	CF = 1	Нет
JNAE	Если не выше и не равно	CF = 1	Нет
JC	Если перенос	CF = 1	Нет
JNB	Если не ниже	CF = 0	Нет
JAE	Если выше или равно	CF = 0	Нет
JNC	Если нет переноса	CF = 0	Нет
JBE	Есть ниже или равно	CF = 1 или ZF = 1	Нет
JNA	Если не выше	CF = 1 или ZF = 1	Нет
JA	Если выше	CF = 0 и ZF = 0	Нет
JNBE	Если не ниже и не равно	CF = 0 и ZF = 0	Нет
JL	Если меньше	SF <> OF	Да
JNGE	Если не больше и не равно	SF <> OF	Да
JGE	Если больше или равно	SF = OF	Да
JNL	Если не меньше	SF = OF	Да
JLE	Если меньше или равно	ZF = 1 или SF <> OF	Да
JNG	Если не больше	ZF = 1 или SF <> OF	Да
JG	Если больше	ZF = 0 и SF = OF	Да
JNLE	Если не меньше и не равно	ZF = 0 и SF = OF	Да

Следующий вопрос: [Структура программы на языке ассемблера. Модули. Сегменты.](#)

Предыдущий вопрос: [Команды побитовой арифметики.](#)

Структура программы

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

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 исполняемом файле, даже **PRIVATE**)

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 (не выполняется выравнивание, т.е. сегмент может начинаться с любого места в памяти)
- WORD (сегмент начинается по адресу кратному 2, т.е. на границу слова)
- DWORD (сегмент начинается по адресу кратному 4, т.е. на границу двойного слова)
- PARA (по умолчанию, сегмент начинается по кратному 16)
- PAGE (сегмент начинается по адресу кратному 265)

Тип:

- PUBLIC - заставляет компоновщик соединить все сегменты с одинаковым именем. Новый объединенный сегмент будет целым и непрерывным. Все адреса (смещения) объектов, а это могут быть, в зависимости от типа сегмента, команды или данные, будут вычисляться относительно начала этого нового сегмента;

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

Следующий вопрос: [Директивы выделения памяти. Метки.](#)

Предыдущий вопрос: [Команды передачи управления.](#)

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 части))
5. Передача управления на точку входа.

Дальше выполняются инструкции заданные в самой программе.

Машинный код

Машинный код - набор команд, который напрямую интерпретируется 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.
Существуют диалекты - TASM, NASM, MASM, FASM, YASM.

Следующий вопрос: Классификация команд процессора x86.

Предыдущий вопрос: Процессор 8086. Шина адреса. Сегментная модель памяти.

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

Выравнивание:

- BYTE (см 12 билет)
- WORD
- DWORD
- PARA (по умолчанию)
- PAGE

Тип:

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

Следующий вопрос: [Директива ASSUME](#).

Предыдущий вопрос: [Директивы выделения памяти. Метки](#).

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 переменными придется явно указывать селектор сегмента.

Если в данный момент сегментный регистр не указывает ни на какой именованный сегмент, то чтобы сообщить об этом Ассемблеру, можно использовать в директиве ASSUME ключевое слово NOTHING.

Например: ASSUME DS:NOTHING

Следующий вопрос: [Директива END. Точка входа.](#)

Предыдущий вопрос: [Директива SEGMENT.](#)

Этой директивой завершается любая программа на ассемблере.

```
.  
. .  
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 директива END без метки. Операнд может быть опущен, если программа не предназначена для выполнения, например, если ассемблируются только определения данных, или эта программа должна быть скомпилирована с другим (главным) модулем.

Следующий вопрос: [Виды переходов. Условные, безусловные переходы. Короткий, ближний, дальний переход.](#)

Предыдущий вопрос: [Директива ASSUME.](#)

38. Математический сопроцессор. Классификация команд.

1. Команды пересылки данных:

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

2. Арифметические команды:

- сложение `fadd` ;
- вычитание `fsub` ;
- умножение `fmul` ;
- деление `fdiv` ;
- так же есть обратное вычитание и обратное деление (источник - приемник, источник / приемник);
- команда поиска частичного остатка от деления (64 вычитания, если не найдено, то выставляется флаг) `frem/freml` ;
- изменение знака `fchs` ;
- модуль числа;
- округление до целого `frndint` ;
- масштабирование по степеням двойки (умножить на 2 в степени..);
- квадратный корень;
- разделить мантиссу и экспоненту (мантийса дописывается на вершину стека, экспонента - на прошлом месте)

3. Команды сравнений:

- сравнивают 2 числа и выставляют флаги (основные и в регистре SR) `fcom <операнд>` - сравнение с врениной стека, некоторые команды выталкивают после сравнения числа из стека. Также можно сравнить только значащих разрядов (мантийс, то есть без порядков), сравнение только целых частей, аналог TEST `fcomi`, а также команда выставления флагов в зависимости от типа числа.

4. Трансцендентные операции сопроцессора:

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

5. Константы FPU:

- `1`
- `0`
- `pi`
- `log2e`
- `log(2, 10)`
- `ln(2)`
- `lg(2)`

6. Команды управления:

- можно менять напрямую указатель вершины стека (увеличить уменьшить);
- освобождение регистра, инициализация сопроцессора (не очищает, просто помечает все регистры свободными);
- команды обнуления флагов;
- сохранения регистров (в памяти для возможности дальнейшего восстановления);
- команды для обработки исключений;

- пустая команда

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

Исключения (не спрашивается, но лучше упомянуть здесь)

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

Следующий вопрос: Расширение MMX. Назначение. Типы данных.

Предыдущий вопрос: Математический сопроцессор. Особые числа.

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)|

(Метка ближнего перехода состоит только из записи в стек регистра IP, а дальнего - CS:IP (4 байта))

Для короткого и ближнего переходов непосредственный operand (число) прибавляется к **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 (и TEST)

Одна из возможностей чтения флагов - использование условных переходов

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 = 1
JNE/JNZ	Не равно/не ноль	ZF = 0
JP/JPE	Есть чётность/чётное	PF = 1
JNP/JPO	Нет чётности/нечётное	PF = 0
JCXZ	CX = 0	

(OF - регистр)

Команда	Описание	Состояние флагов для выполнения перехода	Знаковый
JB	Если ниже	CF = 1	Нет
JNAE	Если не выше и не равно	CF = 1	Нет
JC	Если перенос	CF = 1	Нет
JNB	Если не ниже	CF = 0	Нет
JAE	Если выше или равно	CF = 0	Нет
JNC	Если нет переноса	CF = 0	Нет
JBE	Есть ниже или равно	CF = 1 или ZF = 1	Нет
JNA	Если не выше	CF = 1 или ZF = 1	Нет
JA	Если выше	CF = 0 и ZF = 0	Нет
JNBE	Если не ниже и не равно	CF = 0 и ZF = 0	Нет
JL	Если меньше	SF <> OF	Да
JNGE	Если не больше и не равно	SF <> OF	Да
JGE	Если больше или равно	SF = OF	Да
JNL	Если не меньше	SF = OF	Да
JLE	Если меньше или равно	ZF = 1 или SF <> OF	Да
JNG	Если не больше	ZF = 1 или SF <> OF	Да
JG	Если больше	ZF = 0 и SF = OF	Да
JNLE	Если не меньше и не равно	ZF = 0 и SF = OF	Да

Следующий вопрос: [Способы адресации.](#)

Предыдущий вопрос: [Директива END. Точка входа.](#)

XLAT/XLATB - трансляция в соответствии с таблицей

XLAT <адрес>

XLATB

Помещает в AL байт из таблицы по адресу DS:BX со смещением относительно начала таблицы, равным AL.

Операнда (адрес) служит только для указания сегментного регистра, который будет использоваться вместо DS (по умолчанию).

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

Следующий вопрос: [Команда LEA](#).

Предыдущий вопрос: [Команды условных переходов](#).

- Регистровая адресация (`mov ax, bx`) - Операнд находится в регистре. Способ применим ко всем программно-адресуемым регистрам процессора
- Непосредственная адресация (`mov ax, 2`) (Непосредственный адрес - вставленный в машинный код) - Операнд может быть представлен в виде числа, адреса, кода ASCII, а также иметь символьное обозначение
- Прямая адресация (`mov ax, ds:0032`) (метка во время компиляции преобразуется в прямую) - В команде указывается ячейка памяти, над содержимым которой требуется выполнить операцию.
- 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]`). Относительный адрес ячейки памяти находится в регистре, обозначение которого заключается в квадратные скобки. Относительный адрес операнда определяется суммой трех величин: содержимого базового, а также дополнительного смещения.
- Адресация по базе с индексированием - Относительный адрес операнда определяется суммой содержимого базового, индексного регистров и дополнительного смещения. (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]`

При адресации через регистры BX, SI или DI в качестве сегментного регистра подразумевается DS; при адресации через BP - регистр SS.

Следующий вопрос: [Команда сравнения](#).

Предыдущий вопрос: [Виды переходов. Условные, безусловные переходы. Короткий, ближний, дальний переход](#).

Команды условных переходов.

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

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

Виды условных переходов

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 количество битов|PF = 1|

|JNP/JPO|Нет четности/нечетное количество битов|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|

Следующий вопрос: [Команда XLAT/XLATB](#).

Предыдущий вопрос: [Команда сравнения](#).

LEA (Load Effective Address) - вычисление эффективного адреса

LEA <приёмник>, <источник>

Вычисляет эффективный адрес источника и помещает его в приёмник.

Эффективный (текущий) адрес - это БАЗА + СМЕЩЕНИЕ + ИНДЕКС , где БАЗА - это базовый адрес, находящийся в регистре (при 16-разрядной адресации могут использоваться только регистры BX или BP); СМЕЩЕНИЕ - это константа (число со знаком), заданная в команде; ИНДЕКС - значение индексного регистра (при 16-разрядной адресации могут использоваться только регистры SI или DI). Любая из частей эффективного адреса может отсутствовать.

Позволяет вычислить адрес, описанный сложным методом адресации (да и просто его загрузить).

Оператор OFFSET позволяет определить смещение только при компиляции, и в отличие от него команда LEA может сделать это во время выполнения программы. Хотя в остальных случаях обычно вместо LEA используют MOV и OFFSET, то есть

LEA ПРИЁМНИК, ИСТОЧНИК - это то же самое, что и MOV ПРИЁМНИК, offset ИСТОЧНИК

Инструкция LEA часто используется для арифметических операций, таких как умножение и сложение. Преимущество такого способа в том, что команда LEA занимает меньше места, чем команды арифметических операций. Кроме того, в отличие от последних, она не изменяет флаги. Примеры:

```
;Умножение с помощью LEA  
MOV BX, 8  
LEA BX, [BX + BX * 4] ;Не поддерживается emu8086
```

```
;Сложение с помощью LEA  
MOV BX, 8  
LEA BX, [BX + 16] ;BX = BX + 16 = 8 + 16
```

Следующий вопрос: [Команды десятичной арифметики](#).

Предыдущий вопрос: [Команда XLAT/XLATB](#).