Изменённый список вопросов (всего 66):

1. Архитектура МП 8086 и 80386.
2. Характеристики регистров
3. Флаги.
4. Сегментные регистры по умолчанию. Образование физического адреса.
5. Сегментный префикс.
6. Структура одномодульной программы. Повторные описания сегментов.
7. Возможные структуры кодового сегмента.
8. Возможные способы начала выполнения и завершения программы типа .EXE
9. Структура программы из нескольких исходных модулей.
10. Переменные, метки, символические имена и их атрибуты
11. Виды предложений в языке ассемблер
12. Директивы, псевдооператоры: форма записи, назначение
13. Директивы описания сегментов: формат записи, назначение параметров
14. Возможные комбинации сегментных директив и умолчания
15. Директива ASSUME
16. Структура процедуры
17. Директива END
18. Внешние имена.
19. Листинг программы.
20. Типы данных и задание начальных значений.
24. EQU
25. =
32. Команды передачи управления
33. Команды условных переходов при работе с Целыми Без Знака и Целыми Со Знаком
34. Организация циклов:
35. Директива INCLUDE
37. Способы адресации
38. Арифметические команды (для ЦБЗ и ЦСЗ)
39. Связывание подпрограммы
40. Команда CALL. Использование прямой и косвенной адресации
41. Способы передачи параметров подпрограммы
42. Способы сохранения и восстановления состояния вызывающей программы (кто выполняет и в чьей памяти).
44. Соглашения о связях в Turbo Pascal
45. Соглашения о связях в Turbo C
46. Команды сдвига
47. Команды логических операций
48. Макросредства
49. Описание макроопределений и макрокоманд
50. Рекурсия в макроопределениях
51. Параметры в макросах
52. Директива PURGE
53. Директива LOCAL
54. Директивы условного ассемблирования IF, IFE, IF1, IF2
55. IFDEF, IFNDEF и связанные с ними конструкции
56. IFB и IFNB в макроопределениях
57. Директивы IFIDN И IFDIF в макроопределениях
58. Операции ;; ! & % < > в макроопределениях
59. Блок повторения REPT
60. Блок повторения IRP
61. Блок повторения IRPC
62. Команды обработки строки и префиксы повторения
63. Команда пересылки строк
64. Команда CMSx (для байтов CMSB)
65. Команда сканирования строк
66. Команда загрузки строки
67. Организация рекурсивных подпрограмм
68. Общая схема памяти 8086 под управлением DOS
69. Назначение прерываний
70. Классификация прерываний
71. Как выполняется обработка прерываний
72. Прерывание BIOS и векторы
73. Способы маскирования внешних прерываний.
78. Директива .MODEL
81. Функции DOS 35h и 25h
82. Замена оригинального обработчика прерываний
83. Дополнение оригинального обработчика прерываний
XX. Возможные дополнительные вопросы
XXX. Советы
XXXX. Примеры для строковых команд и прочая лабуда

Ребят! Вы все затащите этот экзамен.

MASM — MACRO Assembler (нет, не microsoft)
TASM — TURBO Assembler

Создатели: Fgs Fds, KINGGOLDrus, Druide

1. Архитектура МП 8086 и 80386.

8086 — 16-разрядный микропроцессор. Имеет 14 16-разрядных регистров, 16-разрядную шину данных, 20-разрядную шину адреса (поддерживает адресацию до 1 Мб памяти), поддерживает 98 инструкций. 8088 отличается от 8086 только тем, что у него 8-разрядная шина данных.

80386 — расширение архитектуры 8086 до 32-разрядной. Появился защищенный режим, позволяющий 32-битную адресацию; все регистры, кроме сегментных, расширились до 32 бит (приставка E, н-р EAX); добавлены два дополнительных 16-разрядных сегментных регистра FS и GS, а также несколько специальных 32-битных регистров (CRx, TRx, DRx). В защищенном режиме используется другая модель памяти, которую мы не рассматриваем (?).

Написать еще, какие конкретно регистры есть и их названия (см. второй вопрос). Дальше как он спросит.

2. Характеристики регистров

Регистр процессора — сверхбыстрая память внутри процессора, предназначенная для хранения адресов и промежуточных результатов вычислений (регистр общего назначения/регистр данных) или данных, необходимых для работы самого процессора (указатель команды).

Всего архитектуры 8086 и 8088 содержат 14 16-битных регистров:

Регистры общего назначения	Аккумулятор	**AX** (**AH** | **AL**)
	База	**BX** (**BH** | **BL**)
	Счётчик	**CX** (**CH** | **CL**)
	Данные	**DX** (**DH** | **DL**)
Указательные регистры	Указатель базы	**BP**
	Указатель стека	**SP**
Указатель команды	Указатель команды	**IP**
Индексные регистры	Индекс источника	**SI**
	Индекс получателя	**DI**
Регистр состояния	Флаги	**FLAGS**

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

Сегментные регистры	Сегмент кода	**CS**
	Сегмент данных	**DS**
	Дополнительный сегмент	**ES**
	Сегмент стека	**SS**

При этом, например, AL - нижние 8 бит регистра AX, AH - верхние (это не отдельные регистры!).

В архитектуре 80386 все регистры, кроме сегментных, были расширены до 32 бит (приставка E), при этом, например, AX стал нижней половиной EAX и т.д., а так же были добавлены два новых 16-битных сегментных регистра FS и GS и несколько специальных регистров (CRx, DRx, TRx), о которых Евтихыч не говорил.

<table>
<tr><td rowspan="4">Регистры общего назначения</td><td>Аккумулятор</td><td><table><tr><td colspan="4">EAX</td></tr><tr><td colspan="2"></td><td colspan="2">AX</td></tr><tr><td></td><td></td><td>AH</td><td>AL</td></tr></table></td></tr>
<tr><td>База</td><td><table><tr><td colspan="4">EBX</td></tr><tr><td colspan="2"></td><td colspan="2">BX</td></tr><tr><td></td><td></td><td>BH</td><td>BL</td></tr></table></td></tr>
<tr><td>Счётчик</td><td><table><tr><td colspan="4">ECX</td></tr><tr><td colspan="2"></td><td colspan="2">CX</td></tr><tr><td></td><td></td><td>CH</td><td>CL</td></tr></table></td></tr>
<tr><td>Данные</td><td><table><tr><td colspan="4">EDX</td></tr><tr><td colspan="2"></td><td colspan="2">DX</td></tr><tr><td></td><td></td><td>DH</td><td>DL</td></tr></table></td></tr>
<tr><td rowspan="2">Указательные регистры</td><td>Указатель базы</td><td><table><tr><td colspan="4">EBP</td></tr><tr><td colspan="2"></td><td colspan="2">BP</td></tr></table></td></tr>
<tr><td>Указатель стека</td><td><table><tr><td colspan="4">ESP</td></tr><tr><td colspan="2"></td><td colspan="2">SP</td></tr></table></td></tr>
<tr><td>Указатель команды</td><td>Указатель команды</td><td><table><tr><td colspan="4">EIP</td></tr><tr><td colspan="2"></td><td colspan="2">IP</td></tr></table></td></tr>
<tr><td rowspan="2">Индексные регистры</td><td>Индекс источника</td><td><table><tr><td colspan="4">ESI</td></tr><tr><td colspan="2"></td><td colspan="2">SI</td></tr></table></td></tr>
<tr><td>Индекс получателя</td><td><table><tr><td colspan="4">EDI</td></tr><tr><td colspan="2"></td><td colspan="2">DI</td></tr></table></td></tr>
<tr><td>Регистр состояния</td><td>Флаги</td><td><table><tr><td colspan="18">EFLAGS</td></tr><tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td></tr><tr><td colspan="14">Резерв для INTEL</td><td>VM</td><td>RF</td></tr><tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr><tr><td>-</td><td>NT</td><td>IOPL</td><td>OF</td><td>DF</td><td>IF</td><td>TF</td><td>SF</td><td>ZF</td><td>-</td><td>AF</td><td>-</td><td>PF</td><td>-</td><td>CF</td></tr></table></td></tr>
<tr><td rowspan="6">Сегментные регистры</td><td>Сегмент кода</td><td>CS (16 бит)</td></tr>
<tr><td>Сегмент данных</td><td>DS (16 бит)</td></tr>
<tr><td>Дополнительный сегмент</td><td>ES (16 бит)</td></tr>
<tr><td>F сегмент</td><td>FS (16 бит)</td></tr>
<tr><td>G сегмент</td><td>GS (16 бит)</td></tr>
<tr><td>Сегмент стека</td><td>SS (16 бит)</td></tr>
<tr><td rowspan="9">Специальные регистры
(Он забыл упомянуть об этих регистрах на лекции, поэтому он будет, скорее всего, рад, если вы их будете знать)</td><td>Управляющий регистр машины</td><td>CR0</td></tr>
<tr><td>Резервный регистр для INTEL</td><td>CR1</td></tr>
<tr><td>Линейный адрес прерывания из-за отсутствия страницы</td><td>CR2</td></tr>
<tr><td>Базовый регистр таблицы страниц</td><td>CR3</td></tr>
<tr><td>Регистр сегмента состояния задачи</td><td>TSSR</td></tr>
<tr><td>Регистр таблицы глоб. дескриптора</td><td>GDTR</td></tr>
<tr><td>Регистр таблицы дескр. прерывания</td><td>IDTR</td></tr>
<tr><td>Регистр таблицы лок. дескриптора</td><td>LDTR</td></tr>
<tr><td>и т.д.</td><td>...</td></tr>
</table>

3. Флаги (регистр (E)FLAGS)

Бит 0. CF (Carry Flag) - флаг переноса; = 1 при переносе из/заёме в (при вычитании) старший бит результата и показывает наличие переполнения в беззнаковой целочисленной арифметике.	**Бит 10. DF (Direction Flag) - флаг направления;** если = 0, то команды вроде **MOVS** будут увеличивать текущий адрес (идти по памяти слева направо), иначе - уменьшать (справа налево).
Бит 2. PF (Parity Flag) - флаг четности; = 1 если младший байт результата содержит чётное число единичных битов.	**Бит 11. OF (Overflow Flag) - флаг переполнения;** = 1, если целочисленный результат слишком длинный для размещения в целевом операнде (регистре или ячейке памяти), т.е. переполнение.
Бит 4. AF (Aux. Carry Flag) - доп. флаг переноса; = 1 при переносе или заёме из бита 4 результата.	**Биты 12, 13. IOPL (I/O Privilege Level) - уровень приоритета ввода-вывода;**
Бит 6. ZF (Zero Flag) - флаг нуля; = 1 если результат последней операции = 0.	**Бит 14. NT (Nested Task) - флаг вложенности задач;** = 1, если текущая задача вложена в другую.
Бит 7. SF (Sign Flag) - флаг знака; = значению старшего бита результата (в знаковой арифметике старший бит - знаковый: если = 1, то число < 0)	**Бит 16. RF (Resume Flag) - флаг возобновления;** флаг маскирования ошибок отладки.
Бит 8. TF (Trap Flag) - флаг трассировки; если = 1, разрешена пошаговая отладка.	**Бит 17. VM (Virtual 8086 Mode Flag) -** установка в защищенном режиме вызывает переход в режим виртуального 8086.
Бит 9. IF (Interrupt Enable Flag) - если = 1, прерывания разрешены (устанавливается и снимается командами **STI** и **CLI**)	**Биты 18-31. Резерв для расширений Intel.**

Прим.: не указанные биты не используются (зарезервированы). Всего 12 флагов (используемых битов) в регистре FLAGS, 2 в регистре EFLAGS. Старший бит - 31-ый в случае двойного слова (отсчет с нуля), 15-ый в случае слова, 7-й в случае байта. Под “результат” имеется в виду результат последней операции/команды. Регистр FLAGS кладется в стек и берется из него командами **PUSHF** и **POPF,** EFLAGS - командами **PUSHFD, POPFD.**

Флаги, помеченные в таблице серым цветом, существуют начиная с архитектуры 80386.

4. Сегментные регистры по умолчанию. Образование физического адреса.

Память представляет собой линейную последовательность байтов, поделенную на параграфы - последовательности из 16 идущих подряд байт, у первого из которых адрес кратен 16. Параграф является минимальным возможным в x86 сегментом. Полный физический адрес составляется из номера сегмента и смещения относительно начала этого сегмента (“байт 5 сегмента 3”).

<полный адрес> = <номер сегмента> * <размер сегмента> + <смещение>

Сегментные регистры (CS, DS, ES, SS) нужны для хранения номера сегмента. Они используются вместе с адресными регистрами (SI, DI, IP, SP), которые указывают на смещение относительно начала сегмента, для получения полного адреса.

CS (регистр сегмента кода) используется с IP для адресации инструкций программы (IP указывает на следующую к выполнению команду), при этом по умолчанию при выполнении команд передачи управления (н-р **JMP**) их операнд воспринимается как смещение относительно текущего сегмента (ближний адрес). Можно передавать управление и в другой сегмент (дальний адрес: номер сегмента и смещение).

DS (регистр сегмента данных) используется по умолчанию для адресации данных (н-р **MOV A, AX** преобразуется в **MOV DS:A, AX**).

ES (регистр доп. сегмента) используется для тех же целей, если требуется доп. сегмент данных или связь приемник-передатчик (DS:SI → ES:DI, например). Требуется некоторыми командами (н-р **MOVS**).
SS (регистр сегмента стека) вместе с SP (указателем стека) используется для организации стека. SP при запуске программы содержит размер стека, если он не равен 0 или 64, и 0 в противном случае.
SS и CS устанавливаются автоматически при запуске программы, DS и ES надо устанавливать вручную.

5. Сегментный префикс.

Конструкция вида AS:X, где AS — какой-то сегментный регистр (DS, ES, SS, CS), а X - какая-то метка (ближний адрес) или другой регистр (SI, DI, SP, IP). Явно указывает, какой регистр мы используем для получения номера сегмента, к которому мы обращаемся. Н-р: **MOV AX, ES:X**

6. Структура одномодульной программы. Повторные описания сегментов.

Пример одномодульной программы:

```
SEG_STACK SEGMENT PARA STACK 'STACK'
  DB 64 DUP('STACK-')
SEG_STACK ENDS

SEG_DATA SEGMENT PARA 'DATA'
  SHIT DW 42
SEG_DATA ENDS

SEG_CODE SEGMENT PARA 'CODE'
  ASSUME CS:SEG_CODE, DS:SEG_DATA, SS:SEG_STACK

  PROGSTART:
    MOV AX, SEG_DATA
    MOV DS, AX
    ; ... DO SOME SHIT
    MOV AH, 4Ch
    INT 21h
SEG_CODE ENDS
END PROGSTART
```

Повторные описания сегментов обычно используются, когда один и тот же сегмент описывается в нескольких модулях, или чтобы расположить данные рядом с операциями над ними. При этом в случае разных модулей указываются видимые снаружи (в других модулях) элементы (**PUBLIC name**), а в модулях, где эти элементы надо увидеть, в том же сегменте указываются их имена и тип (**EXTRN name: type**):

```
; МОДУЛЬ 1
SEG_DATA SEGMENT PARA 'DATA'
  PUBLIC ASS
  EXTRN SHIT: BYTE
  ASS DW 666
SEG_DATA ENDS
```

```
; МОДУЛЬ 2
SEG_DATA SEGMENT PARA 'DATA'
  PUBLIC SHIT
  EXTRN ASS: WORD
  SHIT DB 42
SEG_DATA ENDS
```

7. Возможные структуры сегмента кода.

```
ИМЯ_СЕГМЕНТА_КОДА SEGMENT [<вид_выравнивания>] ['CODE']
  ASSUME CS:ИМЯ_СЕГМЕНТА_КОДА, DS:ИМЯ_СЕГМЕНТА_ДАННЫХ, SS: ИМЯ_СЕГМЕНТА_СТЕКА

  ПРОЦ1 PROC [FAR|NEAR]
    ...
    CALL ПРОЦ2
```

```
    ...
    CALL ПРОЦ3
    ...
    RET|RETF
  ПРОЦ1 ENDP

  ПРОЦ2 PROC
    ...
    RET
  ПРОЦ2 ENDP

  ПРОЦ3 PROC FAR
    ...
    RET         ; ЭТОТ RET АВТОМАТИЧЕСКИ ЗАМЕНИТСЯ НА RETF
  ПРОЦ3 ENDP

  ...

  МЕТКА_ТОЧКИ_ВХОДА:
    ; ...
ИМЯ_СЕГМЕНТА_КОДА ENDS
END МЕТКА_ТОЧКИ_ВХОДА
```

FAR — подпрограмма дальнего вызова (можно вызывать из других сегментов), NEAR — ближнего (вызов только из этого сегмента). **RET** — возврат из подпрограммы ближнего, **RETF** — дальнего вызова. В отличие от обычного **RET**'а, который вытаскивает из стека только значение IP, **RETF** вытаскивает из стека значения CS и IP. По умолчанию стоит NEAR.
<вид_выравнивания> может быть PARA, BYTE, WORD, PAGE. Указывает, чему должны быть кратны адрес начала и конца сегмента. По умолчанию PARA.
Еще вместо метки точки входа можно использовать FAR-процедуру. Тогда можно заканчивать программу точно так же по INT 21H / AX: 4Ch, а можно просто делать в конце **RET**, перед этим запушив в стек 0, а в начале программы перед выставлением регистра DS запушить его старое значение. Остальное не отличается.

8. Возможные способы начала выполнения и завершения программы типа .EXE

Способы входа: `END <метка в кодовом сегменте>, END <процедура (far)>`
Ставится в коде программы после сегмента кода (только ОДИН РАЗ ВО ВСЕЙ ПРОГРАММЕ). См. выше.
Способы выхода: при помощи **INT 21h / AH = 4Ch** и при помощи **RET**'а, если программа началась со входа в подпрограмму.
При этом если выход с помощью **RET**, то на верхушке стека должен лежать 0, а за ним DS, положенный туда в начале программы, а AX должен быть 0. **Почему:** при старте программы DOS кладет в DS адрес своей системной структури, называемой Program Segment Prefix (PSP). Эта структура содержит аргументы командной строки, адрес возврата в DOS и прочую информацию. Первые два байта ее ВСЕГДА равны машинному коду команды **INT 20h.** Поэтому, когда мы кладем в стек DS, а затем 0 прямо в самом начале программы, по **RET(F)** в CS положится адрес PSP, а в IP - 0. А нулевая инструкция в этом нашем новом CS и есть **INT 20h**, которая является одним из прерываний, завершающих программу в DOS, и она будет выполнена процессором следующей после **RET.** Это прерывание берет из AX код ошибки (0 = нет ошибки).
Здесь написать пример кода входа и выхода (см. пример одномодульной программы выше).

9. Структура программы из нескольких исходных модулей.

```
; МОДУЛЬ 1 ГЛАВНЫЙ                          ; МОДУЛЬ 2
```

```
PUBLIC SHIT1

DSEG SEGMENT PARA PUBLIC 'DATA'
  ASS DB 22
DSEG ENDS

SSEG SEGMENT PARA STACK 'STACK'
  DB 64 DUP ('ASS-')
SSEG ENDS

CSEG SEGMENT PARA PUBLIC 'CODE'
  ASSUME CS:CSEG, DS:DSEG, SS:SSEG

  EXTRN SHIT2:NEAR
  SHIT1 PROC FAR
    CALL SHIT2
    MOV AH, 4Ch
    INT 21h
  SHIT1 ENDP
CSEG ENDS
END SHIT1 ; Это определяет точку входа
          ;   во всю программу
```

```
DSEG2 SEGMENT PARA PUBLIC 'DATA'
  FUCK DB "HELLO AVGN!$"
DSEG2
EXTRN ASS:BYTE

CSEG SEGMENT PARA PUBLIC 'CODE'
  ASSUME CS:CSEG, DS:DSEG2

  PUBLIC SHIT2
  SHIT2 PROC NEAR
    MOV AX, SEG ASS
    MOV ES, AX

    MOV AH, 09h
    MOV DX, FUCK
    INT 21h
    INC ES:ASS
    RET
  SHIT2 ENDP
CSEG ENDS
END ; Здесь нихрена нет, поэтому модуль
    ;   не главный
```

Сначала создаются объектники отдельных модулей с помощью компилятора (masm.exe), а затем всё компонуется в экзешник линковщиком (link.exe).

10. Переменные, метки, символические имена и их атрибуты

Конструкции ассемблера формируются из идентификаторов и ограничителей. Идентификатор — набор символов (буквы, цифры, "_", ".", "?", "$", "@"). Символ "." может находиться только в начале идентификатора. Идентификатор не может начинаться с цифры, может размещаться только на одной строке и содержит только от 1 до 31 символа. С помощью идентификаторов можно представить переменные, метки и имена.

Переменные идентифицируют хранящиеся в памяти данные и имеют три атрибута:

- Сегмент
- Смещение
- Тип (DB(1 байт), DW(2), DD(4), DF/DP(6), DQ(8), DT(10))

Метка — частный случай переменной. На неё можно ссылаться посредством переходов и вызовов. Имеет два атрибута:

- Сегмент
- Смещение

Символические имена — символы, определённые директивой EQU, имеющие значение типа "символ" или "число".

Примеры:

```
DB ?                  ; Просто выделена память в сегменте под 1 байт
DB 64 DUB (B, C, D)   ; 64 байта, заполненные поочерёдно B, C и D
A DB 123              ; Переменная A
B EQU 456             ; Константа B
C = 902               ; Константа C, которую можно переобъявлять
```

11. Виды предложений в языке ассемблер

Предложения бывают трех типов:

1. Команды (и их операнды) — символические аналоги машинных инструкций (мнемоники). В процессе трансляции они преобразуются в машинный код ассемблером (н-р **`MOV AX, 0`**)

2. Директивы — указания ассемблеру на выполнение определенных действий; не имеют аналога в машинном коде (н-р **A EQU 9**).
3. Комментарии — строка или часть строки из любых символов, начинающаяся с символа ‘;’.

12. Директивы, псевдооператоры: форма записи, значение

Директивы — указания ассемблеру на выполнение определенных действий; не имеют аналога в машинном коде.

- Директивы определения набора инструкций: н-р, строка **.386** в начале файла с кодом говорит ассемблеру использовать только наборы команд из архитектуры 80386 и ниже.
- Упрощенные директивы сегментации:
  - **.MODEL <модель адр.> [<модиф. адр.>][,<язык>][,<модиф. языка>]:**
    <модель адр>: модель адресации (FLAT - 32-битная стандартная).
    <модиф. адр>: тип адресации: use16 | use32 | dos
    <язык>, <модиф. языка>: определяют особенности соглашений о вызовах
    C, PASCAL, BASIC, FORTRAN и **STDCALL, SYSCALL**
  - **.CODE [<имя сег>]:** задает положение начала сегмента кода.
  - **.STACK [<размер>]:** задает размер стека. По дефолту — 1024б.
  - **.DATA:** начало или продолжение сегменты инициализированных данных.
  - **.DATA?:** начало или продолжение сегмента неиниц. данных.
  - **.CONST:** начало или продолжение сегмента констант.

Также директивы объявления данных DB, DW, DD, …

Псевдооператоры:

- **<ИМЯ_ИДЕНТИФИКАТОРА> EQU <ВЫРАЖЕНИЕ>:** используется для задания констант. Например, **ASS EQU 10** вызовет последующую замену каждого слова ASS в коде на 10 транслятором. Замена “тупая”, то есть во всем тексте программы каждое вхождение того что слева будет заменено на то что справа. Выражения могут быть и текстовые, и числовые, и даже куски текста программы.
- **<ИМЯ_ИДЕНТИФИКАТОРА> = <ЧИСЛОВОЕ_ВЫРАЖЕНИЕ>:** то же, что и EQU, но константы, заданные =, можно переопределять позднее в коде и работает он только для числовых выражений. При этом числовое выражение может включать что угодно, что может в числовом виде вычислить компилятор, н-р: **3 + 2, ASS + 4** (если ASS — заданная выше числовая константа).

13. Директивы описания сегментов: форма записи, параметры

```
<ИМЯ_СЕГМЕНТА> SEGMENT [<ВЫРАВНИВАНИЕ>] [<ТИП>] [<КЛАСС>]
  . . .
<ИМЯ_СЕГМЕНТА> ENDS
<ВЫРАВНИВАНИЕ>: BYTE - без
                WORD - по границе слова
                PARA - по границе параграфа
                PAGE - по 256 байт (странице)
<ТИП>: PUBLIC   - объединение сегментов с одним именем
       STACK    - сегмент является частью стека
       COMMON   - расположение на одном адресе с другими COMMON-сегментами
       AT <ADDR> - расположить по абсолютному адресу ADDR
<КЛАСС>: идентификатор (‘STACK’, ‘CODE’, ‘DATA’, . . .)
```

14. Возможные комбинации сегментных директив и умолчания

Если комбинация не указана, то по умолчанию сегменты считаются разными, **даже если у них одно имя и класс**.
Комбинировать объявления сегментов (сегментные директивы) можно с помощью указания ТИПА сегмента.

Написать общую форму из вопроса выше или пример кода, и типы из вопроса выше (PUBLIC, STACK, …).

15. Директива ASSUME

ASSUME <СЕГ_РЕГ1>:<ИМЯ_СЕГ1>[, <СЕГ_РЕГ2>:<ИМЯ_СЕГ2>[, . . .]]

Пример: **ASSUME DS:DATA_SEG, ES:EXTRA_SEG**

Директива специализации сегментов. Обычно пишется в сегменте кода второй строчкой.

Пример сообщает ассемблеру, что для сегментирования адресов из сегмента DATA_SEG выбирается регистр DS, из EXTRA_SEG — регистр ES. Теперь префиксы DS:, ES: при использовании переменных из этих сегментов можно опускать, и ассемблер их будет ставить самостоятельно.

16. Структура процедуры

Вызов процедуры производится при помощи инструкции CALL. Если процедура NEAR, то в стек заносится только значение IP следующей за CALL'ом инструкции. Если процедура FAR, то дополнительно в стек заносится CS, так как FAR процедура обычно лежит в другом кодовом сегменте.

```
...
CALL A; После выполнения этой строчки в стеке будет лежать адрес CS:[MOV AX, BX]
MOV AX, BX
...
```

Общий вид процедуры:

```
ИМЯ_ПРОЦ PROC [NEAR|FAR]
  ...
  RET              ; Если PROC FAR, то ассемблер заменит RET на RETF
ИМЯ_ПРОЦ ENDP
```

Пример:

Возможное начало и завершение процедуры, которой передается управление при старте программы.

- Процедура должна быть дальней
- Возврат командой RET

```
ИМЯ_ПРОЦ   PROC FAR   ; дальняя процедура
      PUSH  DS        ; помещаем в стек
      MOV   AX, 0     ; сегментную часть PSP
      PUSH  AX        ; и число 0

      MOV   AX, ИмяСегментаДанных      ;  настройка DS на
      MOV   DS, AX                     ; сегмент данных
      ; ASSUME   DS:ИмяСегментаДанных  ; помогаем ассемблеру разобраться с сегм-ами
      ...
      RET   ; тоже дальняя команда
ИМЯ_ПРОЦ   ENDP
```

17. Директива END

```
CSEG SEGMENT PARA PUBLIC 'CODE'
  ; ...
  START:
    ; ...
CSEG ENDS
END START
```

Здесь директива END указывает на то, что точка входа в программу находится на метке START, а так же на окончание файла с исходным кодом. Строки после нее игнорируются. Если меток в коде нет, можно после END ничего не указывать. Если опустить END, MASM выдаст ошибку.

18. Внешние имена.

Объявление внешних имён производится при помощи директивы PUBLIC <имя>, а декларация производится при помощи директивы EXTRN <имя>:<тип>. Работает на метках, именах процедур и переменных/константах.

Пример:

```
; Модуль 1
...
A DW 21
PUBLIC A  ; Теперь A доступна из других
...       ; модулей
```

```
; Модуль 2
...
EXTRN A:WORD  ; Объявление внешней
              ;   переменной, находящейся
...           ;   другом файле
```

19. Листинг программы.

Файл с исходным кодом программы, в котором развернуты все макросы, метки заменены на адреса, константы — на их значения. Генерируется ассемблером в процессе подготовки к трансляции. Для получения у MASM можно указать флаги /Zi /L, у ML — флаг /Fl.

20. Типы данных и задание начальных значений.

- Целые числа:
  - [+|-]XXX - обычно десятичное
  - [+|-]XXXb - (BIN) в двоичной системе
  - [+|-]XXXq - (OCT) в восьмеричной системе (q, чтобы не спутать с 0)
  - [+|-]XXXo - (OCT) в восьмеричной системе
  - [+|-]XXXd - (DEC) в десятичной системе
  - [+|-]XXXh - (HEX) в шестнадцатеричной системе
- Символы и строки (символ — один чар, строка — 2 и более):
  - 'Символы'
  - "Символы"

  (Строки в DOS'е $-терминированные)
- Структуры: См. вопрос 10 (структуры) **(от 16/06 Евтих сказал убрать структуры)**

Начальное значение может быть неопределённым. В таком случае вместо начального значения ставится "?".

24. EQU См. вопрос 12 (псевдооператоры).

25. = См. вопрос 12 (псевдооператоры)

32. Команды передачи управления (еще можно упомянуть INT)

Самая простая: безусловный переход —

JMP <метка_или_адрес>: кладет свой операнд в IP, т.е. следующей будет выполнена инструкция по адресу **<метка_или_адрес>** (передает управление по адресу в операнде).

Команда передачи управления в подпрограмму:

CALL <имя_метки_или_процедуры_или_адрес>: кладет (только если процедура FAR) текущее значение CS и (всегда) адрес возврата (адрес следующей за CALL командой) в стек и делает **JMP <операнд>.**

Предназначена для использования с одной из команд возврата управления вызывающей стороне:

RET [<N>]: если текущая процедура NEAR, то **POP**'ает IP (т.е. передает управление по адресу на верхушке стека). Предполагается, что на верху стека при этом адрес возврата. Если процедура FAR, то вместо себя выполняет **RETF**. Если указан операнд (N), то еще прибавляет к SP эту N.

RETF [<N>]: **POP**'ает IP и CS. Для возврата из FAR-процедур (в другой сегмент). Если указан операнд (N), то еще прибавляет к SP эту N.

33. Команды условных переходов при работе с Целыми Без Знака и Целыми Со Знаком

Имеется 19 команд, имеющих 30 мнемонических кодов операций. (по правде, команд и мнемоник дохрена)
Команды проверяют состояние отдельных флагов или их комбинаций, и при выполнении условия передают управление по адресу, находящемуся в операнде команды, иначе следующей команде.

Команда	Мнемоника	С чем работает	Прыжок, когда
JE |JZ	Equal (Zero)	все	`ZF == 1`
JNE|JNZ	Not Equal (Not Zero)	все	`ZF == 0`
JG |JNLE	Greather	ЦСЗ	`(ZF == 0) && (SF == OF)`
JGE|JNL	Greather or Equal	ЦСЗ	`SF == OF`
JL |JNGE	Less	ЦСЗ	`SF != OF`
JLE|JNG	Less or Equal	ЦСЗ	`(ZF == 1) || (SF != OF)`
JA |JNBE	Above	ЦБЗ	`(CF == 0) && (ZF == 0)`
JAE|JNB	Above or Equal	ЦБЗ	`CF == 0`
JB |JNAE	Below	ЦБЗ	`CF == 1`
JBE|JNA	Below or Equal	ЦБЗ	`(CF == 1) || (ZF == 1)`
JO	Overflow	все	`OF == 1`
JNO	Not overflow	все	`OF == 0`
JC	Carry	все	`CF == 1`
JNC	Not Carry	все	`CF == 0`
JS	Sign (< 0)	ЦСЗ	`SF == 1`
JNS	Not Sign (>= 0)	ЦСЗ	`SF == 0`
JP	Parity	все	`PF == 1`
JNP	Not Parity	все	`PF == 0`
JCXZ	CX is Zero	все	`CX == 0`

34. Организация циклов

`LOOP`*x* метка(параметр), где метка(параметр) — определяет адрес перехода от -128 до +127 от команды `LOOP`.

Мнемоника	Описание
`LOOP`	--CX. Если (CX != 0) — повторяем цикл (переходим к метке, указанной в команде)
`LOOPE |LOOPZ`	--CX. Если (CX != 0) && (ZF == 1) — повторяем цикл.
`LOOPNE|LOOPNZ`	--CX. Если (CX != 0) && (ZF == 0) — повторяем цикл.

Примеры использования:
Обычный цикл:

Ассемблерный код	Аналог на C (просто для понимания)

`MOV CX, 5`
`SUMMATOR:`
`  ADD AX, 10`
	`for (int CX = 5; CX > 0; --CX)`
	`  AX += 10;`

`LOOP SUMMATOR`	

Вложенный цикл:

Ассемблерный код	Аналог на C (просто для понимания)

`MOV CX, 5`
`SUMAX:`
`  PUSH CX  ; Сохранение в стеке CX`
`  ;  внешнего цикла`
`  ADD AX, 10`
`  MOV CX, 7`
`  SUMBX:`
`  ADD BX, 10`
`  LOOP SUMBX`
 	
`  POP CX`
`LOOP SUMAX`
	`for (int CX = 5; CX > 0; --CX)`
	`{`
	`  AX += 10;`
	
	`  // Здесь вместо запушивания я юзаю`
	`  //  другую переменную`
	`  for (int CX2 = 7; CX2 > 0; --CX2)`
	`  BX += 10;`
	`}`

35. Директива INCLUDE

`INCLUDE <имя файла без кавычек>`

Вставляет содержимое файла в код (прям как в сях)

`<имя файла без кавычек>` — ссылка на файл

37. Способы адресации

В инструкциях программ операнды строятся из объектов, представляемых собой имена переменных и меток, представленных своим сегментом смещения (регистры, числа, символические имена с численным значением).

Запись этих объектов с использованием символов: **[ ]**, **+**, **-**, **.** в различных комбинациях называют способами адресации.

1. **Непосредственная адресация** (значение): операнд задается непосредственно в инструкции и после трансляции входит в команду, как составная часть 1 и 2 байта.

   Пр.:
   ```
   MOV  AL, 2          ; (1 байт)
   MOV  BX, 0FFFFH     ; (2 байта)
   ```
2. **Регистровая адресация**: значение операнда находится в регистре, который указывается в качестве аргумента инструкции.

   Пр.:
   ```
   MOV  AX, BX
   ```
3. **Прямая адресация**: в инструкции используется имя переменной, значение которой является операндом.

   Пр.:
   ```
   MOV  AX, X       ;  в AX значение X
   MOV  AX, CS:Y    ;  в AX - CS по смещению Y
   ```
4. **Косвенная регистровая адресация**: в регистре, указанном в инструкции, хранится смещение (в сегменте) переменной, значение которой и является операндом. Имя регистра записывается в [ ].

   Пр.:
   ```
   MOV  BX, OFFSET Z
   ...
   MOV  BYTE PTR[BX], 'ASS'    ; в байт, адресованный регистром BX,
                                   запомнить 'ASS' в сегменте DS
   ```
   Пр.:
   ```
   MOV  AX, [BX]               ; в AX запомнить слово из сегмента, адрес
                                   BX со смещением DS
   ```
   Пр.:
   ```
   MOV  BX, CS:[SI]            ; записать данные в BX, находящиеся в CS по
                                   смещению SI
   ```

5. **Косвенная базово-индексная адресация (адресация с индексацией и базированием)**: смещение ячейки памяти, где хранятся значения операндов, вычисляется, как сумма значений регистров, записанных в качестве параметра в инструкции.
   Форма записи команды:
   `[R1+R2], где  □1 ∈ {□□,□□},□2 ∈ {□□,□□}`
   Пр.: `MOV AX, [BX + DI]`
   Пр.: `MOV AX, [BP + SI]`
   `Для BX по умолчанию сегменты из регистра DS.`
   `Для BP по умолчанию сегменты из регистра SS.`
6. **[Косвенная] базово-индексная адресация со смещением**: (прямая с базированием и индексированием), отличается от п.5 тем, что ещё прибавляется смещение, которое может быть представлено переменной или ЦБЗ.
   Форма записи команды:
   `[R1 + R2 ± ЦБЗ]`
   `[R1][R2 ± ЦБЗ]`
   `имя[R1 + R2 ± Абс. выражение]`
   `имя[R1][R2 ± Абс. выражение]`

   Пр.: `MOV AX, ARR[EBX][ECX * 2] ; ARR + (EBX) + (ECX) * 2 - вот такой монстр`
   `[Имя][База][Индекс [* масштаб]][Абс. выражение]`
   `Смещение операнда = Смещение имени + значение базы + значение индекса * масштаб + значение абсолютного выражения`

38. Арифметические команды (для ЦБЗ и ЦСЗ)

Мнемоника	Описание
NEG	Обратить знаковый бит (* -1)
ADD	Сложение операндов
SUB	Вычитание операндов
ADC	Сложение операндов + флаг переноса (<При> + <Ист> + <CF>)
SBB	Вычитание операндов - флаг переноса (<При> - <Ист> - <CF>)
MUL	Умножение (E)AX на операнд (для беззнаковых)
DIV	Деление (E)AX с остатком на операнд (для беззнаковых). В (E)AX кладётся - результат деления, в (E)DX - остаток
IMUL	**MUL** для знаковых
IDIV	**DIV** для знаковых

Ахтунг:

```
; Пример 1:
      MOV AX, 8000h    ; Думали, переполнение вызывается только ADD'ом или MUL'ом?
      NEG AX           ; Переполнение.
; Пример 2:
      MOV AX, 1234h    ; Думали, что с делением в асме всё нормально? Ну-ну.
      MOV BH, 1        ;
      DIV BH           ; Так как делитель — 1 байт, то будет происходить деление AX
                       ;   на BH, при этом результат будет в AL, а остаток — в AH.
                       ;   Здесь будет переполнение, т.к. 1234h не влезет в AL :D
```

39. Связывание подпрограммы

Связывание подпрограмм — совокупность действий по:

- передаче управления подпрограмме
- передаче параметров через регистры, общую область памяти, области параметров
- возврату управления из подпрограмм
- запоминанию и восстановлению состояния вызывающей программы

Все это — связывание подпрограмм в исполняемом модуле.

В более широком плане, когда речь идет о динамическом вызове подпрограмм или программ, хранящиеся на диске в виде отдельных модулей, к этому перечню добавляется:

1) загрузка модуля подпрограмм в ОЗУ и разрешение внешних ссылок (по передаче управления и по передаче данных).
2) освобождение памяти, после завершения работы вызываемого модуля и возврат управления из вызываемого модуля.

40. Команда CALL. Использование прямой и косвенной адресации

CALL передает управление в другую строку программы, при этом сохраняя в стеке адрес возврата для команды **RET** (он указывает на следующую после **CALL** инструкцию). Если переход в другой сегмент, также сохраняется текущий сегмент кода (CS), в который нужно будет вернуться.

Передача управления подпрограмм с помощью команды **CALL**:

- **CALL** `имя_процедуры` или `имя_метки` в подпрограмме (прямая адресация)
- **CALL** `[переменная]` (косвенная адресация)

41. Способы передачи параметров подпрограммы

1. Через регистры:
   плюсы: быстро, просто
   минусы: мало регистров, нужно постоянно следить за значениями в регистрах, регистры маленькие
2. Через общую область памяти, которая организуется при помощи COMMON сегментов:
   плюсы: удобно возвращать большое кол-во данных
   минусы: можно испортить данные извне + данные перекрываются в памяти, что не есть хорошо
3. Через общий стек (используется чаще всего):
   плюсы: можно передавать большое количество параметров
   минусы: количество параметров ограничено размерами стека и разрядностью BP + требуется наличие стека
4. Через области параметров, адреса которых передаются через регистры:
   (по сути то же, что и 1, но передаем адреса начала списков параметров и может быть их количество)
   плюсы: можно передавать большое количество параметров или даже переменное число параметров
   минусы: должна быть заранее подготовленная под параметры и заполненная область памяти, надо иметь в виду, что вызываемый код может работать с другой областью памяти

Пример второго способа передачи:

```
PARSEG SEGMENT COMMON     ; потом грузим этот сегмент в DS или ES
  px DW ? ; Параметр 1    ; вызывающей и вызываемой стороны и пишем и
  py DW ? ; Параметр 2    ; читаем параметры
  re DW ? ; Результат
PARSEG ENDS
```

42. Способы сохранения и восстановления состояния вызывающей программы (кто выполняет и в чьей памяти).

Запоминание и восстановление состояния вызывающей программной единицы (т.е. регистров).

Вариант запоминания регистров определяется следующим:

- кто (вызывающая / вызываемая программная единица) является владельцем области сохранения

- кто из них фактически выполняет сохранение регистров

Вариант 0:

Хранилище — область (стек) вызываемой программы.

Сохранитель (восстановитель) — вызывающая программа

"+" минимум затраты времени и памяти на сохранение, т.к. вызываемая программа знает, какие нужно сохранить регистры

"-" если много вызовов подпрограммы, то команды, выполняющие сохранение регистров, будут занимать значительную часть кода

Вариант 1:

Хранилище — область (стек) вызываемой программы.

Сохранитель (восстановитель) — вызываемая подпрограмма

"+" сокращение кода основной подпрограммы

"-" подпрограмма не знает, какие регистры сохранять, сохранение регистров описывается в специальном соглашении.

44. Соглашения о связях в Turbo Pascal

Передача параметров через стек в порядке загрузки слева направо.

Возврат результата:

- скалярные результаты по 2, 1 байт через `AX (AL)`, указатели (4 байта) через `DX:AX` (строка - указатель)
- Real **(6-байтовые паскальные)** - в `DX:BX:AX` (это не адресация, просто возврат через эти регистры -- делится на куски по два байта в таком порядке, сверху такой же смысл)

Сохранение результатов (`SI, DI, BP`) в подпрограмме.

За уничтожение параметров в стеке отвечает вызываемая программа, например используя

`RET N` (`N` — количество байт, которые надо очистить)

45. Соглашения о связях в Turbo C

Передача параметров через общий стек. Помещаются параметры из списка параметров функции в стек в порядке справа налево. Если результат возвращается через имя функции, то он помещается, как правило, в регистр `AX` или `DX:AX`.

Сохранение регистров внутри подпрограммы (`BP, SP, CS, DS, SS, SI, DI`), т.е. их поместить в стек в начале подпрограммы, если их значения могут изменяться в подпрограмме.

За уничтожение параметров в стеке отвечает вызывающая сторона. Освобождение стека выполняет вызывающая сторона командой **`ADD SP, N`** (`N` — количество байт, которые надо очистить).

46. Команды сдвига

Мнемоника	Описание		Мнемоника	Описание
SHL/SHR	Логический сдвиг		**ROR/ROL**	Циклический сдвиг
SAL/SAR	Арифметический сдвиг		**RCL/RCR**	**ROR/ROL** с установкой CF

47. Команды логических операций

Мнемоника	Описание		Мнемоника	Описание
NOT	Поразрядное НЕ		**XOR**	Поразрядный XOR
AND	Поразрядное И		**TEST**	Поразрядное И без возврата
OR	Поразрядное ИЛИ			

48. Макросредства

При написании любой программы на ассемблере, возникают некоторые трудности: повторяемость некоторых идентичных или незначительно отличающихся участков программы; необходимость включения в каждую программу участков кода, которые уже были использованы в других программах; ограниченность набора команд.
Для решения этих проблем в языке ассемблер существуют макросредства.
Отличие макросредств от подпрограмм:

- подпрограммы требуют подготовки для вызова (это бывает дольше, чем сам код)
- если в программе много макровыводов, то увеличивается размер кода
- параметры при записи макрокоманды должны быть известны уже на стадии ассемблирования
- в макрокомандах можно пропускать параметры

49. Описание макроопределений и макрокоманд

Макроопределения — специальным образом оформленная последовательность предложений языка ASM. Код управления которой ASM (точнее его часть порождает — макрогенератор) макрорасширение макрокоманд.

Макрорасширения (результат макроподстановки)— последовательность предложений языка, порождающаяся макрогенератором при обработке макрокоманды. /* под управлением макроопределения.*/

Макрокоманда (ссылка на макрос) — предложение в исходном тексте программы, которое воспринимается макрорасширением, как приказ построить макрорасширение и вставить на её место (“вызов” макроса).

Описание макроопределения:

```
ИМЯ MACRO [форм.пар.1[,форм.пар.N]]
  <предложения языка Ассемблер (тело)>
ENDM
```

Формат макрокоманды:

```
<имя макроса> <фактические параметры через запятую и/или пробел>
```

Пример*:*

Описание макроса:

```
PRINTSTR MACRO STR
  MOV  DX, STR
  MOV  AH, 9
  INT  21H
ENDM
```

Вывод строки по смещению R.
Пример вызова:

```
LEA      BX, SOME_STR
PRINTSTR BX ; макрокоманда
```

Замечания по исполнению макроопределений:

- длина формальных параметров ограничена длиной строки
- нет ограничений, кроме физических
- макроопределения рекомендуется размещать в начале программы
- макрос можно определять внутри другого макроса, но тогда вложенный макрос будет доступен только после первого вызова внешнего макроса
- определение макроса с именем, которое ранее уже было использовано (переопределение макроса) затирает предыдущее определение
- макрос можно уничтожить с помощью PURGE <имя_макроса>{, <имя макроса>}
- макроопределения можно хранить в отдельном txt файле и включать в программу с помощью `INCLUDE`

50. Рекурсия в макроопределениях

В теле макроопределения может содержаться вызов другого макроопределения или даже того же самого. Пример:

```
RECUR MACRO P
  MOV AX, P
  IF P
    RECUR %P-1
  ENDIF
ENDM
```

Такой макрос будет вызывать сам себя, пока P не станет равно 0.
Например, `RECUR 3` развернется в

```
MOV AX, 3
MOV AX, 2
...
```

51. Параметры в макросах

Описание макроопределения:

```
ИМЯ MACRO [форм.пар.1[,форм.пар.N]]
  <предложения языка Ассемблер (тело)>
ENDM
```

Пример вызова макроса с фактическими параметрами 1, 2: **`MACRONAME 1, 2`**
Фактические параметры перечисляются через запятую, список параметров не обязательно должен присутствовать. Запятые писать обязательно, даже в том случае, если 1 из параметров отсутствует. Так же в качестве параметров можно использовать выражения.

Пример макроса с параметром:

```
PRINT_STR MACRO STR    ; STR - параметр
  MOV AH, 9
  MOV DX, STR
  INT 21h
ENDM
```

Пример макроса без параметров:

```
EXIT_APP MACRO
  MOV AH,4C
  INT 21h
ENDM
```

Число фактических параметров может отличаться от формальных параметров в макрокоманде.
Если фактических больше, чем формальных, то лишние фактические параметры игнорируются.
Если фактических меньше, чем формальных, то формальные параметры, которые не соответствуют фактическим, заменяются на пустую строку.

52. Директива PURGE

```
PURGE имя_макрокоманды, [имя_макрокоманды]
```

Директива PURGE удаляет определение макрокоманды, состоящей из нескольких строк, с именем "имя_макрокоманды". После использования директивы PURGE ассемблер больше не интерпретирует идентификатор PURGE как макрокоманду.

53. Директива LOCAL

```
LOCAL v1, …, vk (k >= 1)
```

Указывается, какие имена меток следует рассматривать как локальные.
Формы записи:
В макрокомандах:

LOCAL идентификатор[, идентификатор].

В процедурах:

LOCAL элемент[, элемент].[= идентификатор]

Пример:

```
M MACRO
  ...
  L:
    ...
ENDM
```

Если макрос будет вызван несколько раз, то в коде появятся несколько меток L, что недопустимо.
LOCAL L заставляет макрогенератор заменять метки L на имена вида ??xxxx, xxxx - 4-значное hex число. [??0000 - ??FFFF]. Макрогенератор запоминает номер, который он использовал последний раз при подстановке (??n) и в следующий раз подставит n+1 (??(n+1))

54. Директивы условного ассемблирования IF, IFE, IF1, IF2

IF выр блок внутри будет включен в расширение, если выражение ≠ 0

IFE выр блок будет включен в расширение, если выражение = 0

IF1 блок будет включен в расширение при 1 проходе ассемблера.

IF2 блок будет включен в расширение при 2 проходе ассемблера.

Если в качестве имени задана ссылка вперед, она считается неопределенной на 1-м проходе и определенной на 2-м.

55. IFDEF, IFNDEF и связанные с ними конструкции

Работают так же, как и обычный IF, но:

IFDEF name

истинно (блок внутри будет в расширении), если имя name было объявлено выше,

IFNDEF name

истинно, если name не было объявлено выше.

Также можно использовать с ELSE, как и обычный IF:

```
IFDEF A
  ADD AX, A
ELSE
  EXITM
ENDIF
```

В примере к AX будет прибавлено A только если имя A определено, иначе совершится выход из макроса.

56. IFB и IFNB в макроопределениях

IFB arg

Если аргумент не задан, то условие является истинно. Можно сказать, что данная директива проверяет значение аргумента на равенство пустой строке.

IFNB arg

Действие IFNB обратно IFB. Если аргумент задан, то условие истинно.

Пример:

```
SHOW MACRO REG
  IFB REG
    DISPLAY 'НЕ ЗАДАН РЕГИСТР'
    EXITM
  ENDIF
...
ENDM
```

57. Директивы IFIDN И IFDIF в макроопределениях

IFIDN <S1>, <S2> — истинно, если строки (любой текст) S1 и S2 совпадают. (if IDeNtical)

IFDIF <S1>, <S2> — истинно, если строки S1 и S2 различаются. (if DIFferent)

<, > - не метасимволы, они обязательны.
Пример:
```
SOMETHING MACRO
IFIDN <A>, <B>
  DISPLAY  'ЭТО НИКОГДА НЕ ВЫПОЛНИТСЯ'
  EXITM
ENDIF
...
ENDM
```

58. Операции ;; ! & % < > в макроопределениях

1) `;;` — комментарий.
   формат: `;; text`
2) `%` — если требуется вычисление в строке некоторого константного выражения или передача его по значению в макрос.
   формат: `%ВЫРАЖЕНИЕ,`
   например: `K EQU 5, %K+2  => 7`
3) **!** — символ, идущий после данного знака будет распознан, как символ, а не как операция или директива.
   формат: `!c,` например, если нам надо задать строку &A&: "`!&A!&`"
4) **&** — склейка текста (склеить к параметру). Используется для задания модифицируемых идентификаторов и кодов операций.
   формат: `&par | &par& | par&,` например "`Something &A&: &B& something`" - в этой строке &A& и &B& будут заменены на фактические значения A и B. Алсо, это интерполяция строк.
5) **<>** — содержит часть текста программы (в макрорасширении заменяется ровно на то что в скобках). Внутри скобок последовательность любых символов.
   формат: `<text>,` например `<5+2>`

Пример:
```
DEBUGMSG MACRO WHERE, WHAT
  DBGMSG&WHERE DB "&WHAT& occured at &WHERE&$"
ENDM
```
Вызов **`DEBUGMSG 135, <A huge error>`** развернется в
```
  DBGMSG135 DB "A huge error occured at 135$"
```

59. Блок повторения REPT

```
REPT  WWW
 ; ...тело блока
ENDM
```
`WWW` — выражение, значение которого (ЦБЗ) определяет число копирований макроопределения в макрорасширение.

60. Блок повторения IRP

```
IRP FORM, <FACT1, FACT2, … > (угловые скобки обязательны)
 ; ...тело блока
ENDM
```
`FORM` — формальный параметр, который используется в теле блока: `FACT1,..,FACTi` — подставляемый на место формального при **i**-ом копировании.

Пример:
```
IRP X, <0,1,2>
  A&X  DB  X
ENDM
```

После развертки (макрорасширение):
```
A0 DB 0
A1 DB 1
A2 DB 2 ...
```

61. Блок повторения IRPC

```
IRPC param, str
  ; ...тело блока
ENDM
```

`param` — формальный параметр, который используется в теле блока, `str` — набор символов, **i**-ый символ которой при i=1, 2,..,<длина_этого_набора> подставляется на место формального параметра при **i**-ом копировании.

Пример:

```
IRPC X, ABC
  X  DB  'X'
ENDM
```

После развертки (макрорасширение):

```
A  DB  'A'
B  DB  'B'
C  DB  'C' ...
```

62. Команды обработки строки и префиксы повторения

Этими командами могут обрабатываться строки (последовательности байтов или слов). Максимальная длина такой строки - 64 кб (так как регистры 16-битные). С ними часто используются префиксы повторения.

Существуют пять основных операций или примитивов, выполняющих обработку одного элемента - слова (байта) за один приём.

Команды-примитивы (текущий элемент = на него сейчас указывают SI или DI):

- сравнение (**CMPS**): сравнивает поэлементно источник и приемник;
- сканирование (**SCAS**): ищет значение AX(AL) в приемнике (сравнивает текущий элемент с AX(AL));
- пересылка (**MOVS**): копирует поэлементно из источника в приемник
- загрузка (**LODS**): копирует текущий элемент из источника в AX(AL);
- сохранение в ОЗУ (**STOS**): заменяет текущий элемент в приемнике на AX(AL).

Каждая команда имеет 4 разновидности (на примере **MOVS**):

- без суффикса размера элементов, но с явным указанием приёмника и источника: **MOVS <ПРИ>, <ИСТ>**
- ключевое слово с суффиксом **B** и без параметров (команда побайтовой обработки): **MOVSB**
- ключевое слово с суффиксом **W** и без параметров (команда обработки слов): **MOVSW**
- ключевое слово с суффиксом **D** и без параметров (команда обработки двойных слов): **MOVSD**

В версиях этих команд без параметров считается, что источник находится в сегменте DS по смещению SI (**DS:SI**), приемник — в **ES:DI**. После выполнения команды автоматически изменяется значение в SI и DI, причём SI и DI возрастают на 1 (если в конце **B**), 2 (если в конце **W**) или 4 (если в конце **D**) при флаге DF = 0, уменьшаются на 1/2/4 при DF = 1.

Состоянием DF можно управлять командами **CLD** (DF=0) и **STD** (DF=1). В версии с параметрами 1 или 2 выбирается автоматически в зависимости от разрядности аргумента.

Префиксы повторения:

Описание	Мнемоника	Обычно используется с	Условие окончания
Повторять до CX=0	**REP**	**MOVS, LODS, STOS**	CX=0
Повторять, пока равно и CX≠0	**REPE, REPZ**	**CMPS, SCAS**	CX=0 или ZF=0
Повторять, пока не равно и CX≠0	**REPNE, REPNZ**	**CMPS, SCAS**	CX=0 или ZF=1

63. Команда пересылки строк

Команды пересылки байтов **MOVSB**

Функция:

а) при DF = 0 (установить командой CLD)

байт [DS:SI] → байт [ES:DI]

SI = SI + 1

DI = DI + 1

б) при DF = 1 (установить командой STD)

```
байт [DS:SI] → байт [ES:DI]
SI = SI - 1
DI = DI - 1
```

Флаги не меняются.

Пересылка слов **MOVSW**

Всё то же, но копируется по два байта (слово) и

а)
```
SI = SI + 2
DI = DI + 2
```

б)
```
SI = SI - 2
DI = DI - 2
```

Аналогично с двойными словами (4 байта) **MOVSD**

Общая команда

MOVS <Приемник>, <Источник>

Функция: то же, что и команды выше, но для адресов начала источника и приемника, указанных в операндах:

а) то же, что и у **MOVSB**, если тип источника и приемника BYTE

б) то же, что и у **MOVSW**, если тип источника и приемника WORD,

в) то же, что и у **MOVSD**, если тип источника и приемника DWORD

Пример:

```
P1    DD    STR1
P2    DD    STR2
...
LDS   SI,P1
LES   DI,P2
...
MOV   CX,5
REP   MOVSB
```

64. Команда CMPSx

Сравнение байтов **CMPSB**

Функция: сравнение байта по адресу DS:SI с байтом по адресу ES:DI

а) DF = 0,

```
CMP [DS:SI], [ES:DI] ; (т.е. вычитаем из источника приемник и ставим флаги)
SI = SI + 1
DI = DI + 1
```

б) DF = 1

```
CMP [DS:SI], [ES:DI]
SI = SI - 1
DI = DI - 1
```

Сравнение слов **CMPSW**

Функция: сравнение слова по адресу DS:SI со словом по адресу ES:DI

То же самое, но сравниваются слова и

а)
```
SI = SI + 2
DI = DI + 2
```

б)
```
SI = SI - 2
DI = DI - 2
```

Аналогично с двойными словами (4 байта) **CMPSD**

Общая команда

CMPS <Приемник>, <Источник>

Функция: то же, что и команды выше, но для адресов начала источника и приемника, указанных в операндах:

а) ~**CMPSB**, если <Приемник> и <Источник> - BYTE

б) ~**CMPSW**, если <Приемник> и <Источник> - WORD
в) ~**CMPSD**, если <Приемник> и <Источник> - DWORD

Примечание: CMPS выполняет CMP <ИСТОЧ>, <ПРИЕМ>, а не <ПРИЕМ>, <ИСТОЧ>! То есть вычитается из источника приемник, а не наоборот.

65. Команда сканирования строк

Сканирование байтов **SCASB**
Функция: поиск байта в байтовой строке
а) DF = 0,

```
CMP AL, [ES:DI]
DI = DI + 1
```

б) DF = 1,

```
CMP AL, [ES:DI]
DI = DI - 1
```

Сканирование слов **SCASW**
Функция: то же самое, но сравниваются слова и с AX, а еще
а) `DI = DI + 2`
б) `DI = DI - 2`
Аналогично сканирование двойных слов и с EAX **SCASD**

Общая команда
SCAS <Приемник>
Функция: то же, что и команды выше, но в качестве начала приемника используется операнд:
а) ~**SCASB,** если <Приемник> - BYTE
б) ~**SCASW,** если <Приемник> - WORD
в) ~**SCASD,** если <Приемник> - DWORD

66. Команда загрузки строки

Загрузка байтов **LODSB**
Функция: загрузка текущего байта источника в AL
а) DF = 0

```
байт [DS:SI] → AL
SI = SI + 1
```

б) DF = 1

```
байт [DS:SI] → AL
SI = SI - 1
```

Загрузка слов **LODSW**
Функция: то же, что и выше, но с AX и словами и
а) `SI = SI + 2`
б) `SI = SI - 2`
Аналогично загрузка двойных слов и с EAX **LODSD**

Общая команда
LODS <Источник>
Функция: то же, что и команды выше, но в качестве источника используется операнд:
а) ~**LODSB,** если <Источник> - BYTE
б) ~**LODSW,** если <Источник> - WORD
в) ~**LODSD,** если <Источник> - DWORD
Примечание: еще есть серия команд **STOSx**, которые отличаются от **LODSx** тем, что они кладут **ИЗ** AX в **ПРИМНИК** (ES:DI).

67. Организация рекурсивных подпрограмм

Рекурсивные алгоритмы предполагают реализацию в виде процедуры, которая сама себя вызывает.

При этом необходимо обеспечить, чтобы каждый последовательный вызов процедуры не разрушал данных, полученных в результате предыдущего вызова. Поэтому необходимо сохранять используемые в процедуре регистры (например, EBP) и прочие промежуточные данные в стеке. Если процедура работает по соглашениям вызова Turbo C или Turbo Pascal, за этим также надо проследить (ADD SP, передача параметров через стек и прочие операции).

Пример рекурсивной процедуры по соглашениям Turbo C типа `void recur(int);`

```
; вызов:
PUSH CX ; евтих вообще так не делает
MOV  DX, 5
PUSH DX
CALL RECUR
ADD  SP, 2
POP  CX ; но GCC делает
```

```
; процедура (SI, DI, DS не сохраняем потому что
RECUR PROC FAR          ; они не меняются)
  PUSH BP
  MOV BP, SP
  MOV CX, [BP + 6] ; получаем аргумент
  CMP CX, 0 ; если он 0,
  JE RECEND ; то выходим, иначе
  ; ... делаем с ним что-то
  PUSH CX   ; вызываем эту же процедуру
  CALL SUMM
  ADD SP, 2 ; не забываем про аргументы
  RECEND:
    POP BP
  RETF
RECUR ENDP
```

68. Общая схема памяти ПК, основанных на МП 8086, под управлением DOS

- Таблица векторов прерываний
- Данные BIOS
- Данные DOS
- Коды низкого уровня
- Резидентные программы
- Область памяти для загрузки программ
- Графический адаптер
- Ещё какие-то адаптеры
- Видео память
- ---
- Переходы (JMP) на перегрузку

69. Назначение прерываний

Прерывание — приостановление выполнения процессором текущей последовательности команд по сигналу от внешнего устройства, устройства контроля или по специальной команде программы и передача управления специальной процедуре (обработчик прерывания) для обслуживания сделанного запроса на соотв. услугу.
Прежде чем передать управление программе обработки прерывания, центральный процессор разрешит выполнение текущей команды, затем:

1) Записать в стек флаги регистра.
2) Сбрасывать флаг `TF` и `IF` (запрет прерывания).
3) Сохранить в стеке `CS` и `IP`.
4) Загрузить в `CS` и `IP` адрес обработчика прерываний.

70. Классификация прерываний

Каждое прерывание (кроме RESET и NMI) порядковый № (0..255) для вызова командой **INT**.

Внешние прерывания (*асинхронные* - могут возникнуть в любой момент):
Линия RESET (по сути тоже немаскируемое)**:** вектор прерывания не исполнится (нет смысла, ресет ведь).
CPU выполняет обнуление `DS, ES, SS, IF, IP. CS` ставит в 0FFFF (в FFFF:0 хранится **`JMP`**).

Линия NMI (немаскируемое прерывание) - к ней относится прерывания, сигнализирующие о внешних событиях особой важности: отключение питания, сбой памяти. Нельзя запретить сбросом флага `IF`. Имеют высший приоритет.
Линия INTR (маскируемое прерывание): запрещается сбросом флага `IF` в ноль (отсюда и название).
При возникновении такого прерывания его источник посылает CPU т.н. **IRQ (Interrupt Request)** - запрос на обработку прерывания, содержащий уникальный номер асинхронного прерывания (IRQ number, отличается от номера прерывания!). Т.к. они могут возникать во время обработки других прерываний, у каждого номера асинхронного прерывания есть приоритет:

Приоритеты внешних маскируемых прерываний (по убыванию):

IRQ№	**Номер прерывания (INT)**	**Описание**
0	8h	Прерывание интервального таймера
1	9h	Прерывание клавиатуры
...		

Внутренние прерывания (*синхронные* - вызываются действиями программы):
Вызываются либо командой **`INT`**, либо самим CPU при делении на 0 или при переполнении регистров. Также запрещаются сбросом флага `IF`. Вызов: команда **`INT N`** (`N` — номер вектора прерывания).
Для установки `IF` используется **`STI`** (разрешить прерывание), для сброса - **`CLI`** (запретить прерывание).
Пользователь может создавать свои обработчики прерывания, но во время выполнения обработчика `IF` должен быть = 0.
Для выхода из обработчика прерываний используется команда **`IRET`**, которая восстанавливет из стека сохраненные значения регистров `CS`, `IP` и `FLAGS`.

71. Как выполняется обработка прерываний

Программные прерывания осуществляются по команде **`INT N`**, где `N` — число от 0 до 255, либо процессором.
Работа команды в 8086:

1) в стек сохраняются регистры: FLAGS, CS, IP;
2) сбрасываются флаги трассировки и разрешения прерывания (TF и IF);
3) в IP и CS копируется содержимое двух слов памяти, расположенных по физическим адресам 4N и 4N+2 (т.е. в таблице векторов прерываний). В этих словах хранится адрес ISR (обработчика прерывания).

Начинается выполнение ISR (обработчика). Она заканчивается командой возврата из процедуры обработки прерывания (**IRET**), которая читает из стека IP, CS и FLAGS.

72. Прерывания BIOS и векторы

Система BIOS (Basic Input/Output System) находится в ROM и управляет всеми прерываниями в системе. При включении компьютера вызывает прерывание RESET (см. выше).
Векторы прерываний делятся на 5 групп:

- 01-0Fh - прерывания BIOS
- следующие 20 - прерывания DOS
- следующие 20 - зарезервированные (для различных устройств)
- 60h-07Fh - пользовательские прерывания (используются вашей прогой или дровами)
- прерывания диска (80h-FFh)

Векторы прерывания BIOS:

`00h`	Возникает при делении на 0	`01h`	Пошаговое прерывание, происходит, если TF = 1
`02h`	Сбой ОЗУ, откл. питания	`04h`	По команде **INTO**, если OF = 1
`05h`	Печать экрана	`06h/07h`	Резервные векторы
`08h`	Прерывание системного таймера	`09h`	Прерывание клавиатуры

0Dh	Прерывание от жесткого диска	0Eh	Прерывание от гибкого диска
03h	Используется для установки точки останова		
1Ch	Пользовательское прерывание, получающее управление из 8h		

73. Способы маскирования внешних прерываний.

Так как сигнал на входе INTR можно подавить, то внешние прерывания, вызывающие сигнал на этом входе, носят название маскируемых. "Маскируемые прерывания" - которые можно запретить, сбросив флаг IF.
Единственный данный нам способ маскирования внешних прерываний - **с помощью флагов маскирования и трассировки.** Тут можно написать про IRQ и приоритет (см выше).

78. Директива .MODEL

Формат: **.MODEL <модель>[,<яз.программирования>] (точка важна вначале)**
Задает модель памяти assembler-го модуля; разрешает использовать упрощенные сегментированные директивы. Модель памяти === расположение сегментов в ней. Может быть FLAT (для 32-битных приложений), TINY, ...
<яз программирования> определяет используемое по умолчанию соглашение о вызовах (C, PASCAL, …)

81. Функции DOS 35h и 25h - получение и замена старого обработчика прерываний

Обработчики могут либо полностью заменять старый обработчик, либо дополнять функции оригинального обработчика. Для этого необходимо заменить нужный вектор прерывания своим кодом. Во время замены кода в векторе прерываний (например заменяем адрес сегментной части, и в это время возникает прерывание с неведомо какой адресацией), нужно сбрасывать флаг IF разрешающий обработку маскированных прерываний.

Функция 35h 21-го прерывания **INT 21h / AH=35h**

Вход: AH = 35h
AL = № вектора прерывания

Выход: ES:BX - адрес текущего обработчика

Функция 25h 21-го прерывания **INT 21h / AH=25h**

Вход: AH = 25h
AL = № вектора прерывания
DS:DX - адрес нового обработчика

82. Замена оригинального обработчика прерываний

```
OLD_IP  DW ?
OLD_CS  DW ? ; чтобы сохранить адрес исходного обработчика
…
MOV  AH, 35h     ; получить
MOV  AL, 1Ch     ; в ES:BX
INT  21h         ; адрес
; а теперь сохранить
MOV  OLD_CS, ES
MOV  OLD_IP, BX
…
; установка нового обработчика ROUT
PUSH DS
MOV  DX, OFFSET ROUT
MOV  AX, SEG ROUT
MOV  DS, AX
```

```
MOV  AH, 25h
MOV  AL, 1Ch
INT  21h
POP  DS
… ; работает программа с новым обработчиком
; перед завершением программы следует восстановить старый обработчик
CLI  ; запретить обработку маскируемых прерываний, если новый обработчик
     ; использует регистр DS текущей программы, т.к. иначе при смене DS
     ; может возникнуть ошибка
PUSH DS
LDS  DX, DWORD PTR OLD_IP ; вып. одновременно загрузку DS и указанного регистра
MOV  AH, 25h
MOV  AL, 1Ch
INT  21h
POP  DS
STI  ; разрешить обработку прерываний
; процедура ROUT нового обработчика
ROUT  PROC  FAR
      STI
      ... ; далее сохранение всех изменяемых регистров
      PUSH  AX
      ... ; далее тело нового обработчика
      ... ; далее восстановление регистров
      POP  AX
      CLI
      MOV  AL,  20h    ; данные строки только для обработчиков
      OUT  20h, AL     ; внешних прерываний, в противном случае их нет!!!
      IRET
ROUT  ENDP
```

83. Дополнение оригинального обработчика прерываний

Пример такого обработчика:

```
DOPOBR PROC
      ...              ; сохранение регистров
      PUSHF
      ...              ; дополнительная обработка, возможно и выход по IRET
      MOV  AL,  20h    ; данные строки только для обработчиков
      OUT  20h, AL     ; внешних прерываний, в противном случае их нет!!!
      ...
      POPF
      INT  60h         ; пользовательское прерывание, куда мы повесили старый обр.
      ...              ; дополнительная обработка
      ...              ; восстановление регистров
      IRET
DOPOBR ENDP
```

Пример: Дополнение **INT 16h**

```
;подготовка и установка
MOV   AH, 35h
MOV   AL, 16h
INT   21h      ; в ES:BX теперь вектор оригинального обработчика прерываний 16h
CLI
MOV   DX, BX
MOV   BX, ES
MOV   DS, BX
MOV   AH, 25h ; устанавливаем старый обработчик на пользовательское INT 60h
MOV   AL, 60h
INT   21h
PUSH  DS
MOV   DX, OFFSET DOPOBR
MOV   AX, SEG DOPOBR
MOV   DS, AX
MOV   AH, 25h ; устанавливаем новый обработчик на нужное прерывание (16h)
MOV   AL, 16h
INT   21h
POP   DS
STI
…     ; работа программы и восстановление оригинального обработчика
```

Вариант2:

```
; вместо INT 60h в обработчике (DOPOBR) можно писать так:
PUSHF
CALL <АдресОригОбработчика>
```

Вариант3:

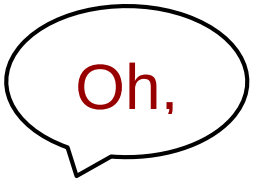
```
; или так:
JMP <АдресОригОбработчика>
```

Логика такова: мы сохраняем старый обработчик прерывания, вешаем его на какое-то свободное пользовательское прерывание (например 60h), затем нужному прерыванию выставляем свой обработчик, который в процессе обработки вызывает **INT 60h**, т.е. старый обработчик, а потом (или перед) делает что-то еще. Выходит, что дополнили.

XX. Возможные дополнительные вопросы

Вопрос 1: нарисовать стек вызовов для какой-либо функции по соглашениям
(функцию Евтих может дать сам)

По соглашениям Turbo C (он такой стек рисовал на доске):



shit(P1, …, Px)

	Сегмент стека
	Px
	…
BP+6	P1
BP+4	Адрес
BP+2	возврата
BP+0	BP
	Лок. перем
	SI
	DI
	DS

Соглашение Turbo Pascal отличается тем, что аргументы в обратном порядке.
Соглашение Turbo Delphi отличается тем, что первые три аргумента слева направо могут положиться в EAX, EDX, ECX, если они влезают, и этих аргументов тогда не будет в стеке.

XXX. Список советов:

1. Евтихычу влом у всех и каждого проверять экзамен, поэтому я не думаю, что он начнёт кого-то валить. Если он что-то начнёт дополнительно задавать, то либо он что-то не понял, либо ответ был дан хреновый.
2. Не злите Евтихыча, так как он, если злится, включает Ban Mode на час, а это никому не надо. Особенно на экзамене.
3. Списывать можно, особенно когда он уже у кого-то проверяет, но осторожно. Могут все-таки быть другие преподы, которые не проверяют у вас экзамен Евтихыча, но просто следят или у них какой-то свой экзамен.
4. Если Евтихыч запалит при списывании, он начнёт злиться. Далее всё пойдёт по пункту 2.
5. Сидеть будем в аудитории для лабораторных, типа той, что была у Градова. Евтихыч обычно не ходит и сидит на троне в торце стола.
6. Чтобы повесить Евтихыча, можно напомнить ему про то, что он не рассказал на лекции или же убрал из вопросов.

XXXX. Примеры для строковых команд и прочая лабуда

Евтих может попросить некий "существенный пример". Вот примеры таких примеров. FLAGS вообще Евтих говорил не сохранять, но без этого на самом деле не работает. Проги по соглашениям Turbo C для 32-битных хуиток. Может попросить расширить до полной проги (с сегментами и т.д.).

MOVS: Скопировать строку.

```
STRCPY PROC FAR
  PUSH EBP
  MOV  EBP, ESP
  ...                   ; тут еще пушнуть регистры ESI, EDI, ES, FLAGS
  MOV  ES, DS           ; предполагаем что строки в том же сегменте
  MOV  ESI, [EBP + 8]   ; откуда
  MOV  EDI, [EBP + 12]  ; куда
  MOV  ECX, [EBP + 16]  ; длина
  REP  MOVSB            ; копируем пока CX не кончится
  ...                   ; тут попнуть FLAGS, ES, EDI, ESI
  POP EBP
  RET
STRCPY ENDP
```

CMPS: Сравнить две строки.

```
STRCMP PROC FAR
  PUSH EBP
  MOV EBP, ESP
  ...                   ; тут еще пушнуть регистры ESI, EDI, ES, FLAGS
  MOV  ES, DS           ; предполагаем что строки в том же сегменте
  MOV  ESI, [EBP + 8]   ; оффсет первой строки
  MOV  EDI, [EBP + 12]  ; второй
  MOV  EAX, [EBP + 16]  ; длина строки 1
  MOV  ECX, [EBP + 20]  ; длина строки 2
  CMP  EAX, ECX         ; если длины не равны, то не равны
  JNE  CMPFAIL
  REPE CMPSB            ; гуляем по строкам пока не дойдем до конца
  JNE  CMPFAIL          ; или не найдем отличающуюся хуитку
  MOV  EAX, 1           ; равны!
  JMP  CMPEND
  CMPFAIL:
    MOV EAX, 0          ; не равны
  CMPEND:
    ...                 ; тут попнуть FLAGS, ES, EDI, ESI
    POP BP
    RET
STRCMP ENDP
```

SCAS: Найти позицию символа в строке (она в DS).

```
STRCHR PROC FAR
  PUSH   EBP
  MOV    EBP, ESP
  ...                     ; тут еще пушнуть регистры ES, FLAGS, ESI, EDI
  MOV    ES,  DS
  MOV    EDI, [EBP + 6]   ; где
  MOV    AX,  [EBP + 10]  ; какой символ (один байт, но в стеке вроде все по два)
```

```
  MOV    ECX, [EBP + 12] ; длина
  REPNE  SCASB           ; идем по строке
  JNE    FOUND           ; пока не найдем
  MOV    EAX, -1         ; не нашли
  JMP    CHREND
  FOUND:
    MOV  EAX, ESI        ; нашли
  CHREND:
    ...                    ; тут попнуть EDI, ESI, FLAGS, ES
    POP    EBP
    RET
STRCHR ENDP
```

STOS: Создать строку данной длины, состоящую из повторяющегося символа.

```
REPCHR PROC FAR
  PUSH   EBP
  MOV    EBP, ESP
  ...                    ; тут еще пушнуть регистры ES, FLAGS, ESI, EDI
  MOV    ES,  DS
  MOV    EBI, [EBP + 6]  ; куда положить
  MOV    AX,  [EBP + 10] ; какой символ (один байт, но в стеке вроде все по два)
  MOV    ECX, [EBP + 12] ; длина
  REP    STOSB           ; повторяем символ ECX раз
  ...                    ; тут попнуть EDI, ESI, FLAGS, ES
  POP    EBP
  RET
REPCHR ENDP
```

LODS: По аналогии со STOS (например, найти символ на позиции N и вернуть его в AL)