

Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
«Казанский национальный исследовательский технический университет им. А.Н. Туполева-КАИ»  
Институт компьютерных технологий и защиты информации

Индекс по учебному плану): Б1.О.12.01

Специальность: 10.05.02 Информационная безопасность телекоммуникационных  
систем

Специализация: "Разработка защищенных телекоммуникационных систем"

Методические указания к лабораторным работам  
по дисциплине

**ОСНОВЫ ПРОГРАММИРОВАНИЯ**

## Лабораторная работа №1

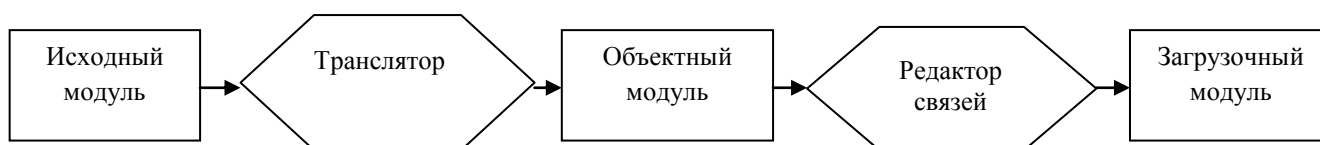
### Регистровая, непосредственная и прямая адресация.

### Арифметические и логические команды

#### Принцип работы ассемблера.

Ассемблер – это программа, которая преобразует программу на языке ассемблера в программу на машинном языке. Исходный файл с программой на языке ассемблера преобразуется ассемблером в объектный модуль (промежуточное представление информации). Редактор связей выполняет преобразование объектного кода в загрузочный модуль (программу на машинном языке).

Для Турбо-ассемблера этот процесс может быть представлен так:



Исходный модуль – это файл с расширением *asm*. Объектный модуль – это файл с расширением *obj*. Загрузочный модуль – это файл с расширением *exe*.

Транслятор – это программа *tasm.exe*. Редактор связей – это программа *Tlink.exe*. Ассемблер наряду с объектным модулем формирует еще 2 файла – файл листинга с расширением *lst* и файл перекрестных ссылок с расширением *sgf*. Листинг содержит исходный и объектный код, а также сообщение об ошибках.

#### Регистры микропроцессора

#### Синтаксис языка ассемблера.

Каждый оператор языка ассемблера может содержать несколько полей. Единственное обязательное поле – это поле кода операции.

Поле метки	Поле кода операции	Поле операндов	Поле комментария
------------	--------------------	----------------	------------------

Поле кода операции определяет ту команду или директиву, которую должна выполнить ЭВМ. Команда транслируется в исполняемый код. Директива не вызывает появления кода, а управляет работой самого ассемблера.

Поле операндов содержит информацию о величинах, участвующих в операции.

Поле метки – необязательное поле, используется для обозначения определенного места в памяти. Метка состоит из букв, цифр и специальных символов. Не может начинаться с цифры. Ес-

Поле комментария – необязательное поле. Служит для указания дополнительной информации о команде или программе. Начинается с символа ; .

Псевдокоманды – это директивы ассемблера, которые указывают, что в этом месте программы располагаются данные (переменные).

*DW* – Определить слово (2 байта).

### 3. ? резервирование памяти.

Допускается указывать список операндов, разделенных запятыми.

$A \quad db \quad 10$  ; определяется байт со значением  $10 = 1010_2 = A_{16}$

Номер бита в байте	7	6	5	4	3	2	1	0
В двоичном коде	0	0	0	0	1	0	1	0
В шестнадцатеричном коде	0				A			

$D \quad dw \quad 256$  ; определяется слово, содержащее значение  $256 = 100000000_2 = 100_{16}$ .

	Старший байт слова								Младший байт слова							
Номер бита в байте	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
В двоичном коде	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
В шестнадцатеричном коде	0				1				0				0			

*dw*    *0FF25h*    ; слово, содержащее значение 65317

	Старший байт слова	Младший байт слова
--	--------------------	--------------------

Номер бита в байте	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
В шестнадцатеричном коде	F				F				2				5			

C db '123' ; 3 байта, содержащие коды символов

	Первый байт								Второй байт								Третий байт							
Номер бита в байте	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
В 16-ричном коде	3				1				3				2				3				3			

Распределение памяти при этом имеет следующий вид:

Адрес	0		1		2		3		4		5		6		7	
Значение	0	A	0	0	0	1	2	5	F	F	3	1	3	2	3	3
	байт		слово				слово				строка символов					

В словах младшие байты заносятся по младшим адресам памяти, а старшие – по старшим.

### Структура программы

Программа состоит из директив и команд.

```
.model tiny ; директива, определяющая модель памяти,
; при которой код и данные находятся в одном сегменте
.code ; директива, определяющая кодовый сегмент программы
; команды, которые загружают сегментные регистры микропроцессора
N: push cs ;
pop ds
; команды, составляющие тело программы
; команды, при помощи которых управление возвращается операционной системе
mov ax,4c00h
int 21h
.data ; директива, определяющая сегмент данных
; директивы распределения данных
end N ; директива, определяющая конец текста программы
```

### Адресация

Команды определяют действия над операндами в программе. Способ указать месторасположение операндов называется адресацией.

**Регистровая адресация.** Операнд находится в регистре. При этом в команде указывается

имя регистра. Например: *ax* или *dl*.

**Непосредственная адресация.** Операнд непосредственно указывается в тексте программы. Операнд может быть указан в десятичном, двоичном или шестнадцатеричном виде. Например: *156*, *0a5h* или *011010b*.

**Прямая адресация.** Операнд находится в памяти. В команде указывается известный адрес операнда, который задается парой значений: <сегментный регистр>: <смещение>. Если данные находятся в сегменте данных *DS*, то указание на сегмент опускают. Смещение указывается числом, не превышающим 16 разрядов для реального режима. Если данные задаются при помощи директив определения данных, тогда можно вместо смещения указывать имя переменной. Например: *es:0001* ; значение по адресу 1 в сегменте *ES*.

*[0F001h]* ; значение по адресу *0F001h* в сегменте *DS*.

*x* ; значение по адресу *x* в сегменте *DS*.

### Команды

**Команда пересылки:** *Mov* **приемник, источник**

Команда копирует содержимое источника в приемник. Источник не изменяется.

Приемник может быть регистром общего назначения, сегментным регистром, кроме регистра *CS*, или ячейкой памяти.

Источник может быть регистром общего назначения, сегментным регистром, ячейкой памяти или непосредственным операндом.

На операнды команды *Mov* накладываются следующие ограничения:

- Оба операнда одновременно не могут находиться в памяти.
- Размеры источника и приемника должны совпадать.
- Нельзя выполнять пересылку из одного сегментного регистра в другой сегментный регистр.
- Нельзя выполнять пересылку в сегментный регистр непосредственного операнда

Пример

Пусть *x* равно 100 (значение длиной 1 байт). Написать программу для копирования переменной *x* в ячейку *y*.

Решение

*X* и *y* – это ячейки памяти. По правилам, нельзя копировать из памяти в память, поэтому мы предварительно скопируем значение *x* в какой-нибудь регистр размером 1 байт, например, регистр *al*. Затем скопируем значение из регистра *al* в ячейку *y*.

Программа

*.model tiny*

*.code*

```

N:   push    cs      ; регистровая адресация операнда
      pop     ds
; адресация первого операнда регистровая, второго – прямая
      mov al,x       ; копируем значение x в al
      mov y,al       ; копируем значение из al в y
; адресация первого операнда регистровая, второго – непосредственная
      mov     ax,4c00h
      int     21h
.data
x db 100 ; определили 1 байт памяти со значением 100
y db ?   ; зарезервировали 1 байт памяти
end N

```

### **Арифметические команды**

**Команда сложения:**            **Add    приемник, источник**

Операнды складываются, и результат помещается в приемник.

Приемник=приемник+источник

**Команда вычитания:** **Sub    приемник, источник**

Из приемника вычитается значение источника, и результат помещается в приемник.

Приемник=приемник-источник

Приемник может быть регистром общего назначения или ячейкой памяти.

Источник может быть регистром общего назначения, ячейкой памяти или непосредственным операндом

Оба операнда одновременно не могут находиться в памяти. Размеры источника и приемника должны совпадать.

Пример

Вычислить значение выражения  $z=x+315-y$  при  $x=1025$ ,  $y=7$ .

Решение

Значение  $x$  больше 255, поэтому для размещения его в памяти требуется 2 байта. Длины операндов должны быть одинаковыми, следовательно, под  $y$  и  $z$  также надо выделить два байта памяти. Оба операнда одновременно не могут находиться в памяти, поэтому будем использовать вспомогательный регистр  $ax$ .

Программа

```

.model tiny
.code
N:   push    cs

```

```

pop    ds
mov ax,x      ; ax=x
add ax,315    ; ax=ax+315=x+315
sub ax,y      ; ax=ax-y =x+315-y
mov z,ax      ; z=ax
mov  ax,4c00h
int   21h

```

*.data*

```

x dw  1025  ; определили 1 слово памяти со значением 1025
y dw  7      ; определили 1 слово памяти со значением 7
z dw  ?      ; зарезервировали 2 байта памяти
end N

```

**Инкремент :**      *Inc*    **приемник**

Увеличивает значение приемника на 1

Приемник это регистр или ячейка памяти.

**Декремент:**      *Dec*    **приемник**

Уменьшает значение приемника на 1.

Приемник это регистр или ячейка памяти.

**Умножение чисел без знака:**      *Mul*    **источник**

Один сомножитель обязательно содержится в аккумуляторе. Другой сомножитель (источник) - это регистр или ячейка памяти. Произведение заносится в аккумулятор (*al* или *ax*).

Источник- 1-й сомножитель	Аккумулятор- 2-й сомножитель	Произведение
байт	<i>al</i>	<i>ax</i>
слово	<i>ax</i>	<i>dx:ax</i>
двойное слово	<i>eax</i>	<i>edx:eax</i>

Если старшая половина результата (*dx* или *edx*) содержит только нули (результат целиком поместился в младшую половину), флаги *CF* и *OF* устанавливаются в 0, иначе - в 1.

Пример

Дано  $x=15$ ,  $y=20$

Вычислить  $z=x*y$ .

Решение

Операнды имеют размер 1 байт. Один из сомножителей, например *y*, заносим в регистр *al*. Произведение будет записано в регистр *ax*. Сохраним его в ячейке *z* размером слово.

Программа

```
.model tiny
.code
N:  push  cs
    pop   ds
    mov  al,y      ; первый сомножитель в аккумуляторе
    mul  x          ; умножаем x на al, произведение будет записано в ax
    mov  z,ax       ; z=ax=x*y
    mov  ax,4c00h
    int  21h

.data
x db 15
y db 20
z dw ?
end N
```

#### **Целочисленное деление без знака: *Div* источник**

Делимое обязательно должно находиться в аккумуляторе. Делитель (источник) – это регистр или ячейка памяти.

Размер делителя	Делимое	Частное	остаток
байт	<i>ax</i>	<i>al</i>	<i>ah</i>
слово	<i>dx:ax</i>	<i>ax</i>	<i>dx</i>
Двойное слово	<i>edx:eax</i>	<i>eax</i>	<i>edx</i>

Результат всегда округляется в сторону нуля. Переполнение или деление на ноль вызывает сообщение об ошибке.

Пример

Дано:

Переменные *x* и *y* размером 1 байт содержат следующие значения: *x*=109, *y*=37. Вычислить частное от деления *x* на *y*.

Решение

Размер делителя - байт. Следовательно, делимое должно находиться в регистре *ax*. Но наше делимое *x* имеет длину 1 байт, поэтому мы занесем его в регистр *al* (младшие разряды реги-



стра *ax*), а регистр *ah* (старшие разряды регистра *ax*) обнулим. Таким образом, значение регистра *ax* будет равно значению *x*.

Программа

```
.model tiny
.code
N:  push  cs
    pop   ds
    mov  al,x
    mov  ah,0
    div  y      ;содержимое ax=x делим на y, частное будет записано в al
    mov  z,al   ;сохраняем частное в ячейке длиной 1 байт
    mov  ax,4c00h
    int  21h

.data
x db  109
y db  37
z db  ?

end N
```

### **Логические команды**

<b>Логическое И:</b>	<b>And</b>	<b>приемник, источник</b>
<b>Логическое ИЛИ:</b>	<b>Or</b>	<b>приемник, источник</b>
<b>Исключающее ИЛИ:</b>	<b>Xor</b>	<b>приемник, источник</b>
<b>Инверсия:</b>	<b>Not</b>	<b>приемник</b>

Приемник может быть регистром общего назначения или ячейкой памяти. Источник может быть регистром общего назначения, ячейкой памяти или непосредственным операндом.

Оба операнда одновременно не могут находиться в памяти. Размеры источника и приемника должны совпадать.

Эти команды выполняют поразрядно соответствующую логическую операцию над приемником и источником и помещают результат в приемник.

Наиболее часто *And* применяют для выборочного обнуления отдельных битов приемника.

Команду *Or* чаще всего используют для выборочной установки отдельных битов приемника.

Команда *Xor* используется для выборочного инвертирования отдельных битов приемника или для обнуления регистров.

### Пример

Установить в единицу значение третьего бита регистра *al*.

### Решение

Для решения поставленной задачи применим оператор *or*. Приемник – это регистр *al*. Источник сформируем следующим образом: в третьем разряде запишем единицу, а остальные обнулим. Таким образом, при выполнении поразрядно операции *or* над операндами, в третьем разряде результата появится единица, а значения остальных разрядов будут такими же, как в *al*.

номер разряда	7	6	5	4	3	2	1	0
Регистр <i>al</i>	x	x	x	x	x	x	x	x
источник	0	0	0	0	1	0	0	0
результат	x	x	x	x	1	x	x	x

*Or al,0000100b*

### Сравнение: *Cmp* приемник, источник

Выполняется так же как команда вычитания *Sub*, но результат не записывается, только устанавливаются флаги. Обычно команду *Cmp* используют вместе с командами условного перехода.

### Тестирование: *Test* приемник. источник

Выполняется так же как команда *And*, но результат не записывается, только устанавливаются флаги. Команда *Test*, так же как и *Cmp* используется в основном в сочетании с командами условного перехода.

### Команды сдвига

Команда	Назначение
SAR приемник, счетчик	Арифметический сдвиг вправо
SAL приемник, счетчик	Арифметический сдвиг влево
SHR приемник, счетчик	Логический сдвиг вправо
SHL приемник, счетчик	Логический сдвиг влево

Приемник – регистр или ячейка памяти. Счетчик – число или регистр *CL*, из которого учитываются только младшие 5 бит, принимающие значения от 0 до 31. Для того, чтобы можно было указывать сдвиг числами, большими единицы, надо включить в программу директиву .386.

Эти четыре команды выполняют двоичный сдвиг приемника вправо (в сторону младшего бита) или влево (в сторону старшего бита) на значение счетчика. Операция сдвига на 1 эквивалентна умножению (сдвиг влево) или делению (сдвиг вправо) на 2.

Команды *SAL* и *SHL* выполняют одну и ту же операцию – на каждый шаг сдвига старший бит заносится в флаг *CF*, все биты сдвигаются влево на одну позицию, и младший бит обнуляется.

Команда *SHR* осуществляет прямо противоположную операцию: младший бит заносится в флаг *CF*, все биты сдвигаются на 1 вправо, старший бит обнуляется.

Команда *SAR* действует по аналогии с *SHR*, только старший бит не обнуляется, а сохраняет предыдущее значение.

Например, команда

*shl ax,1*

выполняет умножение содержимого регистра *ax* на 2.

### Задания

1. Дано:  $X=10$ ,  $Y=16$ . Вычислить  $Z=(X+Y)*2$
2. Дано:  $X=514$ ,  $Y=999$ . Вычислить  $Z=X*Y-1000$
3. Дано:  $X=10$ ,  $Y=1006$ . Вычислить  $Z=X+Y*2$
4. Дано:  $A=4095$ ,  $B=255$ . Вычислить  $C = (A - 200)/B$
5. Дано:  $A=450$ ,  $B=300$ . Вычислить  $C = (A+500)/(B+1)$
6. Дано:  $A=4095$ ,  $B=256$ . Вычислить  $C = (A+B)/7$
7. Дано:  $ah=10110110b$ .
  - а. Обнулить второй и шестой разряды регистра *ah*.
  - б. Установить в единицу нулевой и третий разряды регистра *ah*.
  - в. Инвертировать четвертый и пятый разряды регистра *ah*.

### Порядок работы

1. Запустить **Total Commander**
2. Войти в каталог, указанный преподавателем.
3. Создать свой подкаталог: нажать **F7** и ввести в качестве имени каталога свою фамилию.
4. Нажать **Shift+F4** и набрать имя файла с расширением **asm**.
5. Ввести текст программы (с крайней левой позиции):

*.model tiny*

*.code*

*N: push cs*

```
pop    ds
; ввести команды программы
mov    ax,4c00h
int    21h
.data
; ввести директивы распределения данных
end N
```

6. Сохранить программу: нажать **F2,F10** или **F10, Сохранить**
7. Выполнить компиляцию: дать команду  
**Tasm <имя файла>.asm /L**
8. Если ошибок нет, то появится файл с расширением *obj*: **<имя файла>.obj**. Иначе
  - a. Просмотреть листинг и проанализировать ошибки. Для этого надо выделить файл **<имя файла>.lst** и нажать **F4**.
  - b. Исправить ошибки. Для этого надо выделить файл **<имя файла>.asm** и нажать **F4**, внести необходимые исправления и сохранить файл.
  - c. Повторить пункт 7.
9. Выполнить редактирование связей: дать команду  
**Tlink <имя файла>.obj**
10. Выполнить программу в среде отладчика **Turbo Debugger**: дать команду  
**Td <имя файла>.exe**
11. Подготовить отчет о проделанной работе. Отчет должен содержать задание, текст программы и полученные результаты.

## Лабораторная работа №2

### Адресация косвенная адресация, адресация по базе со сдвигом и адресация по базе с индексированием.

#### Команды передачи управления. Команда цикла.

#### Алгоритмы с разветвлениями. Циклические алгоритмы

### Адресация

#### Косвенная адресация

Выполняется по схеме:

<сегментный регистр>: [<регистр>]

В регистре находится адрес операнда (смещение операнда относительно начала сегмента). Регистр может быть одним из следующих: *BX*, *BP*, *SI* или *DI*. Если смещение хранится в регистре *BP*, то по умолчанию используется сегментный регистр *SS*, иначе - *DS*. Например,

*mov ax,[bx]*

#### Адресация по базе со сдвигом

Выполняется по схеме:

<сегментный регистр>: [<базовый регистр> + <смещение>]

К содержимому базового регистра добавляется смещение, и результат рассматривается как адрес операнда. Базовый регистр – *BX*, *BP*, *SI*, *DI*. Смещение – это число со знаком размером байт или слово.

Допустимые формы записи адреса:

*mov ax, [bx + 2]*

*mov ax, 2[bx]*

*mov ax, [bx] + 2*

Смещение может быть указано не только числом, но и именем переменной (меткой).

*mov al,a[bx]*

Такая адресация применяется, например, при организации доступа к массиву элементов. При этом адрес начала массива соответствует смещению, а содержимое регистра соответствует индексу элемента, к которому производится обращение.

Пусть дан массив из 6 байтов:

*mas db 1, 67, 23, -5, 50, 30*

Тогда для доступа к элементу массива используем обращение вида *[bx+ mas]*, где *bx* – базовый регистр, а *mas* – смещение. Если занести в регистр *bx* значение 2, то после выполнения

команды

*mov ah, mas [bx]*

регистр *ah* получит значение 23 .

### **Адресация по базе с индексированием**

Выполняется по схеме:

<сегментный регистр>: [<баз регистр> + <инд. регистр> + < смещение >]

Адрес операнда вычисляется как сумма значений в 2х регистрах и смещения.

Базовые регистры: *BX* или *BP*. Индексные регистры: *SI* или *DI*. Смещение – это число со знаком размером байт или слово или метка.

Допустимые формы записи адреса:

*2[bx][si]*

*[bx][si+2]*

*[bx+2][si]*

*[bx][si]+2*

*[bx+si+2]*

## **Команды передачи управления**

### **Безусловный переход: *JMP* операнд**

Команда *JMP* передает управление в другую точку программы, не сохраняя какой-либо информации для возврата. Операнд может быть меткой, регистром или переменной, содержащей адрес перехода.

Пример:

*jmp kon*

; команды, которые надо обойти

*kon: mov ax,4c00h*

*int 21h*

### **Команды условной передачи управления**

В общем виде команды условной передачи управления представляются так:

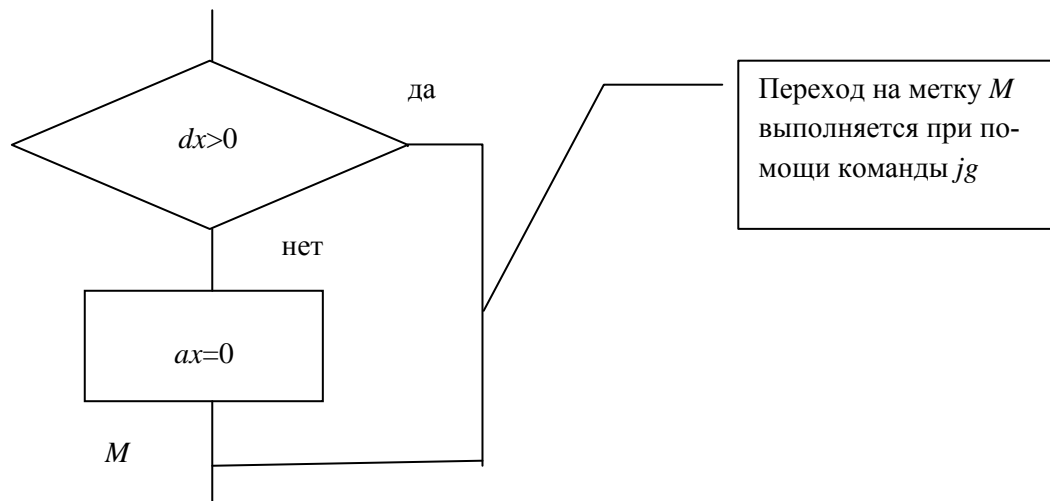
#### ***Jcc* операнд**

Операнд может быть меткой, регистром или переменной, содержащей адрес перехода.

Это группа команд, которые выполняют передачу управления, если выполняется заданное условие. При этом проверяется состояние флагов, которое устанавливаются при выполнении предыдущих команд программы. Адрес перехода должен находиться в пределах -128...+127 байт от команды условной передачи управления.

Код команды	Условие	Условие для <i>СМР</i>
<i>JA</i> <i>JBE</i>	$CF=0$ и $ZF=0$	Если выше Если не ниже и не равно
<i>JAE</i> <i>JNB</i> <i>JNC</i>	$CF=0$	Если выше или равно Если не ниже Если нет переноса
<i>JB</i> <i>JNAE</i> <i>JC</i>	$CF=1$	если ниже если не выше и не равно если перенос
<i>JBE</i> <i>JNA</i>	$CF=1$ или $ZF=1$	Если ниже или равно Если не выше
<i>JE</i> <i>JZ</i>	$ZF=1$	Если равно Если ноль
<i>JG</i> <i>JNLE</i>	$ZF=0$ и $SF=OF$	Если больше Если не меньше и не равно
<i>JGE</i> <i>JNL</i>	$SF=OF$	Если больше или равно Если не меньше
<i>JL</i> <i>JNGE</i>	$SF \neq OF$	Если меньше Если не больше и не равно
<i>JLE</i> <i>JNG</i>	$ZF=1$ или $SF \neq OF$	Если меньше или равно Если не больше
<i>JNE</i> <i>JNZ</i>	$ZF=0$	Если не равно Если не ноль
<i>JNO</i>	$OF=0$	Если нет переполнения
<i>JO</i>	$OF=1$	Если есть переполнение
<i>JNP</i> <i>JPO</i>	$PF=1$	Если нет четности Если нечетное
<i>JP</i> <i>JPE</i>	$PF=1$	Если есть четность Если четное
<i>JNS</i>	$SF=0$	Если нет знака
<i>JS</i>	$SF=1$	Если есть знак

Пример: обнулить регистр *ax* , если содержимое регистра *dx* меньше 0.



Сравнение регистров выполним при помощи команды *cmp*. Далее в таблице найдем команду, которая позволяет передать управление на нужную метку при условии, что *dx* окажется больше 0. Это команда *jg*.

*cmp dx,0* ;устанавливаются флаги

*jg m* ;переход на метку *m*, если  $dx > 0$ , иначе – переход к следующему оператору

*mov ax,0*

*m:*

### Оператор цикла

метка:

; тело цикла

**LOOP** метка

Уменьшает регистр *CX* на 1 и, если *CX* не равен нулю, выполняет переход на метку. Метка должна отстоять от команды *LOOP* не дальше, чем на -128.. +127 байтов. Эта команда используется для организации циклов, в которых регистр *CX* играет роль счетчика.

Пример. Дан массив *A* из 5 элементов типа *byte*. Вычислить сумму элементов массива.

*.model tiny*

*.code*

*start: push cs*

*pop ds*

*mov bx,0* ;смещение (индекс) первого элемента массива=0

*xor al,al* ;начальное значение суммы =0

*mov cx,5* ; количество элементов массива ( счетчик )=5

*m: add al,A[bx]* ; добавить текущий элемент массива к сумме



```

inc bx      ; увеличить смещение на 1, т.к. длина элемента 1 байт
loop m      ; повторять, пока cx не станет равно 0
mov ax,4c00h
int 21h

.data
A          db 3,5,2,7,4
end start

```

### Задания

1. Дано:  $a$  и  $b$  - значения типа *byte*. Присвоить переменной  $C$  значение наибольшего из  $a$  и  $b$ .
2. Дано:  $a$  и  $b$  - значения типа *word*. Присвоить переменной  $C$  значение наименьшего из  $a$  и  $b$ .
3. Дано:  $x$ ,  $y$ ,  $z$  - значения типа *word*. Присвоить переменной  $C$  значение наименьшего из них.
4. Дано:  $x$ ,  $y$ ,  $z$  - значения типа *byte*. Присвоить переменной  $C$  значение наибольшего из них.
5. Дан массив  $X$  из 8 значений типа *byte*. Увеличить в 4 раза все элементы массива.
6. Дан массив  $X$  из 8 значений типа *byte*. Уменьшить в 2 раза все элементы массива.
7. Дан массив  $A$  из 8 значений типа *word*. Вычислить сумму отрицательных элементов массива.
8. Дан массив  $B$  из 16 значений типа *byte*. Вычислить сумму положительных элементов массива.
9. Дан массив  $M$  из 16 значений типа *word*. Вычислить количество положительных элементов массива.
10. Дан массив  $MAS$  из 8 значений типа *word*. Вычислить количество отрицательных элементов массива.
11. Дан массив  $X$  из 8 значений типа *byte*. Увеличить в 2 раза положительные элементы массива и обнулить отрицательные элементы.
12. Дан массив  $Y$  из 8 значений типа *word*. Уменьшить в 2 раза положительные элементы массива и обнулить отрицательные элементы.
13. Даны 2 массива  $A$  и  $B$ , содержащие по 8 значений типа *byte*. Создать новый массив  $C$ , составленный по правилу:  $C_i = \max(a_i, b_i)$
14. Даны 2 массива  $A$  и  $B$ , содержащие по 4 значений типа *word*. Создать новый массив  $C$ , составленный по правилу:  $C_i = \min(a_i, b_i)$

### Лабораторная работа №3

#### Команды работы со стеком PUSH и POP.

#### Команда загрузки исполнительного адреса LEA.

#### Ввод-вывод с использованием функций прерывания int 21h Dos

##### Команды работы со стеком

##### **Поместить данные в стек:      *Push источник***

Источник - регистр, сегментный регистр, непосредственный операнд или переменная (ячейка памяти). Размер источника – слово. Например,

Push cx

Push 100

Push mas

##### **Считать данные из стека:      *Pop приемник***

Приемник - регистр общего назначения , сегментный регистр (кроме CS), переменная (ячейка памяти). Размер источника – слово. Например,

Pop cx

Pop ds

Pop mas

##### Вычисление исполнительного адреса

##### ***Lea    приемник, источник***

Источник – ячейка памяти. Приемник – регистр.

Вычисляет адрес и помещает его в приемник. Например, по команде

*Lea dx, [bx+di]*

В регистр *dx* будет записана сумма регистров *bx* и *di*.

#### Ввод-вывод с использованием функций прерывания int 21h Dos

**Функция 01h - Чтение символа с клавиатуры с ожиданием, эхом и проверкой на ctrl-break.**

Вход	АH	01h
Выход	АL	Код введенного символа

Пример

*mov ah,01h*

*int 21h*; в регистре *al* будет находиться код введенного символа.

**Функция 02h - Вывод символа на экран**

Вход	<i>AH</i> <i>DL</i>	<i>02h</i> код символа
Выход	<i>AL</i>	Код введенного символа

Пример

*mov ah,02h*

*mov dl,'M'*

*int 21h*; на экран будет выведен символ 'M'

### Функция *09h* - Вывод строки символов на экран

Вход	<i>AH</i> <i>DS: DX.</i>	<i>09h</i> Адрес начала строки, которая заканчивается символом "\$".
Выход	<i>AL</i>	Код последнего выведенного символа

Пример

*mov ah,09h*

*lea dx,stroka* ;адрес начала строки

*int 21h* ;будет выведено сообщение « ПРИМЕР \$»

...

*stroka db 'ПРИМЕР \$'*

### Функция *0ah* - Ввод строки символов с клавиатуры

Вход	<i>AH</i> <i>DS:DX</i>	<i>0ah</i> Адрес входного буфера
Выход		Буфер содержит ввод, заканчивающийся символом с кодом <i>0dh</i>

Первый байт буфера содержит максимально допустимое количество символов при вводе *max*. Остальные байты перед вызовом прерывания содержат подсказку - "шаблон".

При вводе во второй байт буфера заносится количество введенных символов (не более *max-1*). Вводимые символы размещаются, начиная с третьего байта. Последний символ в буфере - всегда символ *CR* (с кодом *0dh*), который не учитывается в байте длины.

Таким образом, если предполагается ввод строки длиной не более 10 символов, то *max* принимаем равным 11, а под буфер выделяем 12 байтов.

Пример:

*mov ah,0ah*

*lea dx,text; ;занесли в dx адрес буфера*

*int 21h ;в строку text будет введен текст длиной не  
;более 10 символов*

*...*

*Text db 10,'место ввода' ; выделяем память под буфер*

После ввода строки 'ABC' буфер примет вид:

*Text db 10,3,'ABC',0DH,'о ввода'*

### **Задание**

1. Ввести с клавиатуры четырехзначное целое число, увеличить его в 2 раза и вывести результат на экран. Ввод и вывод числа выполнять посимвольно. Для формирования двоичного числа из введенных десятичных цифр, воспользоваться блок-схемой, приведенной на рис.  
1. Для вывода полученного числа на экран воспользоваться блок-схемой, приведенной на рис. 2. Ввести с клавиатуры свою фамилию, вставить ее в строку вида «Студент <фамилия> выполнил лабораторную работу №3» и вывести на экран полученную строку.

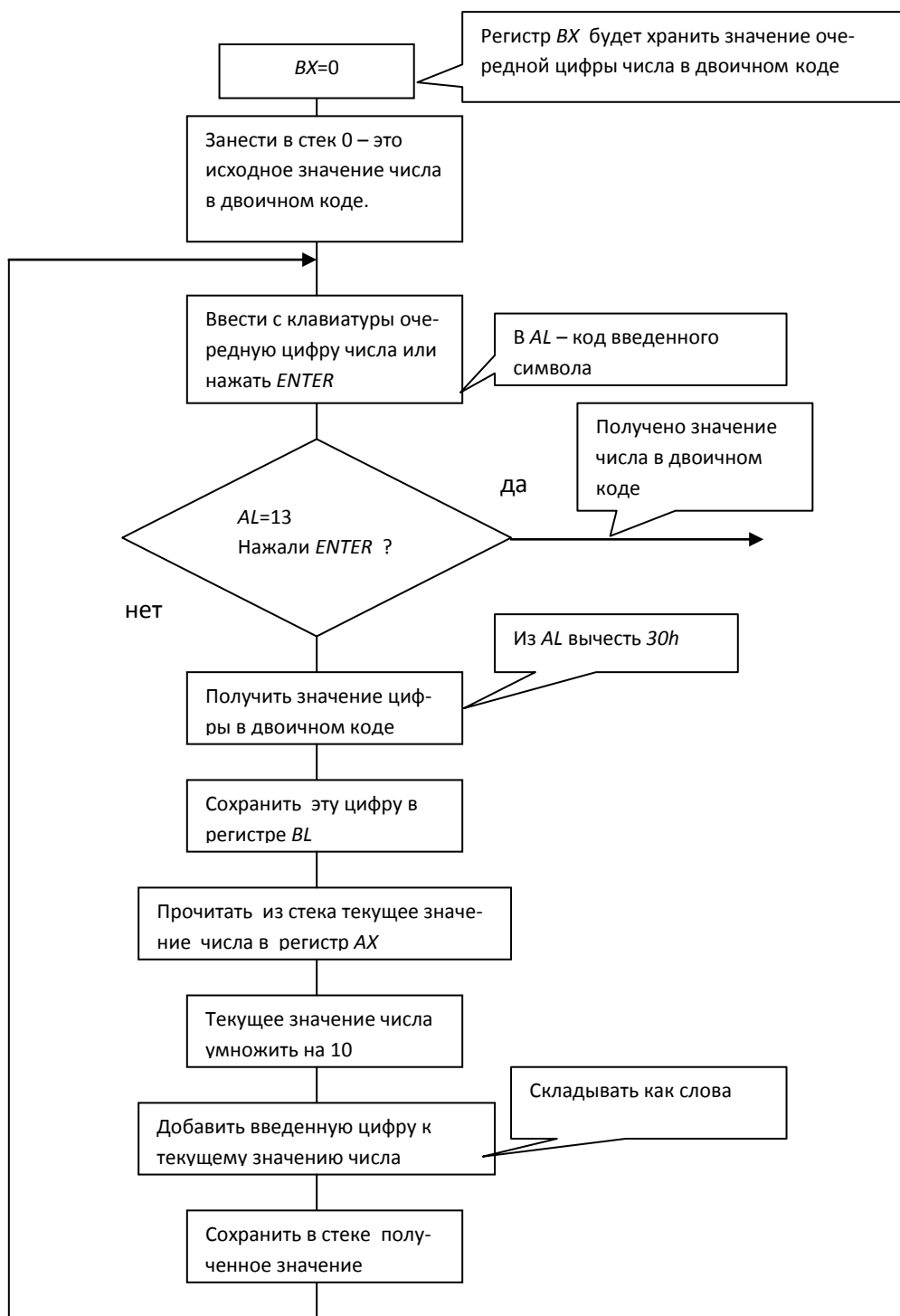


Рис. 1

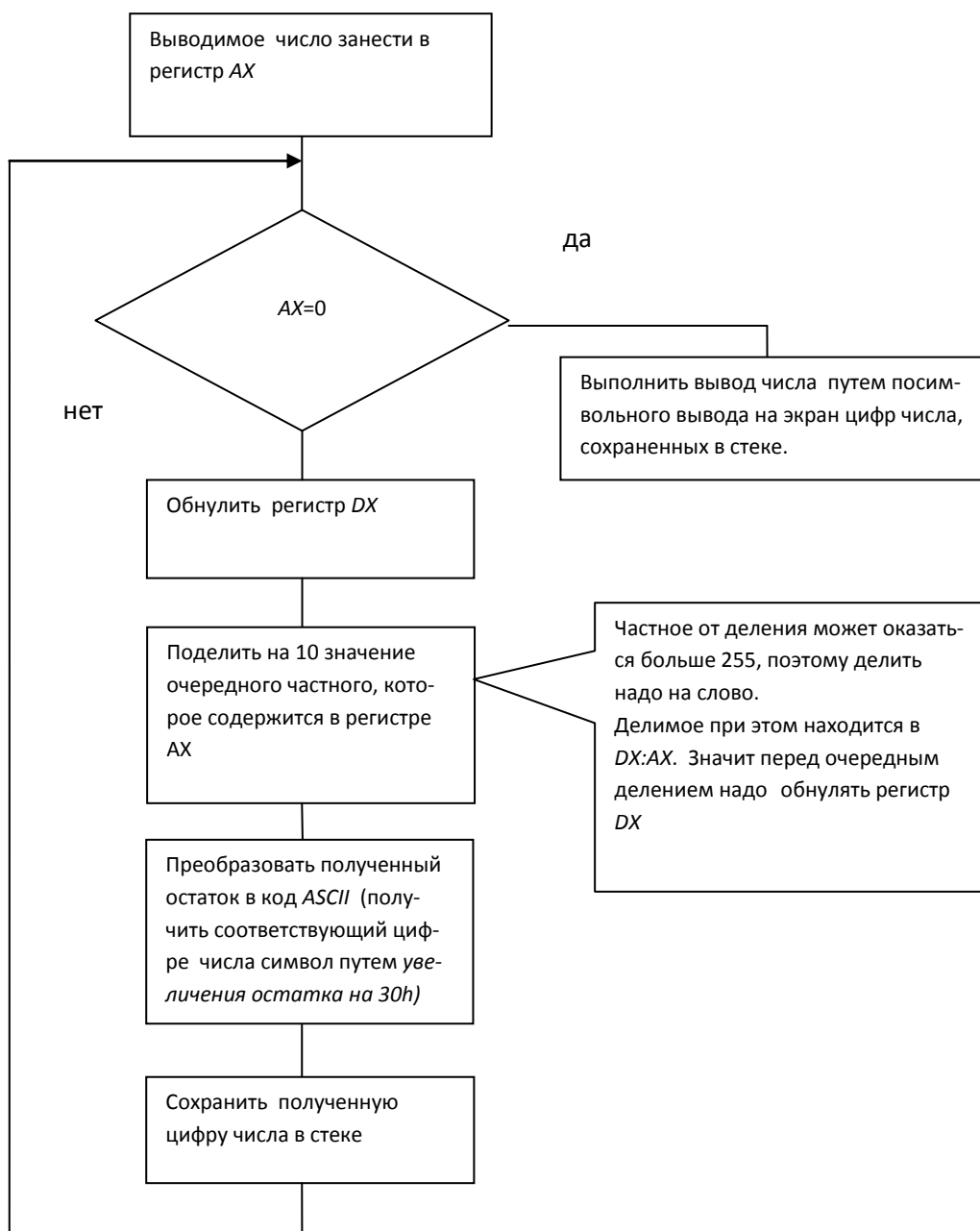


Рис. 2

## Лабораторная работа №4

### Директивы управления сегментами

#### Стандартные директивы

Директива **Segment** стоит в начале описания сегмента. Ее формат:

*Имя сегмента Segment выравнивание тип разрядность класс*

*Имя сегмента* – это метка. Имя используется для получения сегментного адреса и для объединения сегментов в группы.

*выравнивание* может принимать значения

- *Byte* - сегмент может начинаться с любого адреса,
- *Word* - сегмент начинается с адреса, кратного 2,
- *Dword* - сегмент начинается с адреса, кратного 4,
- *Para* - сегмент начинается с адреса, кратного 16 (по умолчанию),
- *Page* - сегмент начинается с адреса, кратного 256.

*Тип* определяет тип комбинирования сегментов и может принимать значения

- *Public* - все сегменты с одинаковыми именами объединяются в один,
- *Stack* – то же, что *Public* , но для стека,
- *Common* - сегменты с одинаковыми именами объединяются в один, но не последовательно, а начиная с одного и того же адреса, т.е. длина сегмента равна длине наибольшего сегмента.
- *At* – сегмент располагается фиксированному абсолютному адресу,
- *Private* - сегмент с другими не объединяется ( по умолчанию)

*Разрядность*

- *Use16* - команды и адреса считаются 16-разрядными, размер сегмента до 64К ( по умолчанию),
- *Use32* команды и адреса считаются 32-разрядными, размер сегмента до 4ГБ.

*Класс* - это любая метка, взятая в одиночные кавычки. Сегменты одного класса располагаются в памяти друг за другом.

Любой из параметров может быть опущен. Например,

*Code segment*

*Prim segment word*

*ABC segment public 'a'*

Директива **ends** стоит в конце описания сегмента. Ее формат:

*Имя сегмента ends*

*Имя сегмента* должно совпадать с именем, указанным в директиве ***segment***.

Директива ***Assume*** имеет формат:

***Assume*** *регистр* : *связь*, *регистр* : *связь*,

где *регистр* – это имя сегментного регистра, *связь* – это имя сегмента.

***Assume*** указывает ассемблеру с каким сегментами связаны сегментные регистры, но ***assume*** не изменяет значения регистров. Сегментные регистры надо загружать в программе.

Директиву ***assume*** следует использовать в каждой программе, в которой используются стандартные директивы определения сегментов. Например,

```
; описание сегмента данных
data segment
; директивы определения данных
data ends
; описание кодового сегмента
cseg segment
assume cs:cseg, ds:data
; команды
cseg ends
```

### ***Упрощенные директивы определения сегментов***

При использовании упрощенных директив определения сегментов, следует определить модель памяти. Она задается директивой ***Model***.

Формат директивы ***Model***:

***Model*** *модель*, *язык*, *модификатор*

Параметр *модель* может принимать следующие значения:

- *Tiny* – код, данные и стек размещаются в одном сегменте до 64КБ,
- *Small* – код размещается в одном сегменте, а данные и стек в другом сегменте,
- *Compact* - код размещается в одном сегменте, а данные в нескольких сегментах,
- *Medium* - код размещается в нескольких сегментах сегменте, а данные в одном,
- *Large*-код и данные могут занимать несколько сегментов
- *Huge* – то же, что Large, но суммарный размер данных может превышать 64 КБ
- *Flat* – то же, что Tiny, но сегменты 32-разрядные.

*Язык* – необязательный параметр. Может принимать значения *C*, *Pascal*, *Basic*, *Fortran* и др.

Показывает, что программа рассчитана на вызов из программ на этих языках.

*Модификатор* – необязательный параметр. Может принимать значения:



- *Nearstack*- сегмент стека объединяется в одну группу с сегментами данных (принимается по умолчанию)

- *Farstack* – сегмент стека не объединяется в одну группу с сегментами данных

Сегмент кода описывается директивой *code*:

*.Code* имя сегмента

Сегмент данных описывается директивой *data*:

*.Data*

Сегмент стека описывается директивой *Stack*:

*.Stack* размер

Во всех моделях памяти сегменты *.data*, *.stack* и сегмент *.code* (в модели памяти *Tiny*) автоматически объединяются в группу *DGROUP*. При этом сегментный регистр *DS* (и *SS*, если не было указано *Farstack*, и *CS* в модели *Tiny*) настраивается на эту группу, как при выполнении директивы *Assume*.

### Загрузка сегментных регистров данных

Директивы управления сегментами позволяют связать сегментные регистры с сегментами программы, но не загружают их нужными значениями. Загрузка сегментных регистров данных должна выполняться программой.

Например, пусть применяются стандартные директивы описания сегментов, а программа содержит два сегмента данных *Data1* и *Data2* и один кодовый сегмент *Code*. Тогда выполнить загрузку сегментных регистров можно так:

;описание сегмента данных

*Data1 segment word 'data'*

...

*Data1 ends*

; описание сегмента данных

*Data2 segment word 'data'*

...

*Data2 ends*

; описание кодового сегмента

*Code segment word 'code'*

*Assume cs: code, ds:data1,es:data2*

*S: mov ax,data1 ; адрес сегмента data1 занести в ax*

*mov ds,ax ;скопировать в ds адрес сегмента*

*mov ax,data2 ;*

```

        mov es,ax
        ...
code    ends
        end S

```

Другой пример. Пусть применяются упрощенные директивы описания сегментов. Модель памяти – *small* (код программы находится в одном сегменте, а стек и данные – в другом). Загрузка сегментного регистра данных в этом случае выполняется так:

```

.model small
.code
begin:  mov ax,@data; адрес сегмента data занести в ax
        ; mov ax, dgroup – другой способ загрузки адреса сегмента данных в ax
        mov ds,ax      ; скопировать в ds адрес сегмента
        ...
.data
...
end begin

```

### Задания

1. Директивы описания сегментов – стандартные. Выравнивание - по словам. Упорядочить массив из 8 элементов по возрастанию. Тип элементов массива – *byte*.
2. Директивы описания сегментов – стандартные. Выравнивание - по двойным словам. Упорядочить массив из 4 элементов по возрастанию. Тип элементов массива – *word*.
3. Директивы описания сегментов – стандартные. Выравнивание – по страницам. Упорядочить массив из 8 элементов по убыванию. Тип элементов массива – *byte*.
4. Директивы описания сегментов – стандартные. Выравнивание – по параграфам. Упорядочить массив из 4 элементов по убыванию. Тип элементов массива – *word*.
5. Директивы описания сегментов – упрощенные. Стек, данные и код – в одном сегменте. Упорядочить массив из 8 элементов по возрастанию. Тип элементов массива – *byte*.
6. Директивы описания сегментов – упрощенные. Стек и данные – в одном сегменте, код – в другом сегменте. Упорядочить массив из 8 элементов по возрастанию. Тип элементов массива – *byte*.
7. Директивы описания сегментов – упрощенные. Стек, данные и код – в одном сегменте. Упорядочить массив из 4 элементов по возрастанию. Тип элементов массива – *word*.

8. Директивы описания сегментов – упрощенные. Стек и данные – в одном сегменте, код – в другом сегменте. Упорядочить массив из 4 элементов по возрастанию. Тип элементов массива – *word*.
9. Директивы описания сегментов – упрощенные. Стек, данные и код – в одном сегменте. Упорядочить массив из 8 элементов по убыванию. Тип элементов массива – *byte*.
10. Директивы описания сегментов – упрощенные. Стек и данные – в одном сегменте, код – в другом сегменте. Упорядочить массив из 8 элементов по убыванию. Тип элементов массива – *byte*.
11. Директивы описания сегментов – упрощенные. Стек, данные и код – в одном сегменте. Упорядочить массив из 4 элементов по убыванию. Тип элементов массива – *word*.
12. Директивы описания сегментов – упрощенные. Стек и данные – в одном сегменте, код – в другом сегменте. Упорядочить массив из 4 элементов по убыванию. Тип элементов массива – *word*.
13. Директивы описания сегментов – стандартные. Выравнивание – по словам. В массиве *A* обнулить все элементы, значения которых встречаются в массиве *B*. Тип элементов массива – *byte*.
14. Директивы описания сегментов – упрощенные. Стек и данные – в одном сегменте, код – в другом сегменте. Даны массивы *A* и *B*. Создать массив *C* из элементов, которые содержатся одновременно в *A* и *B*. В массивах *A* и *B* каждый элемент встречается только один раз. Тип элементов массива – *word*.

## Лабораторная работа №5

### Процедуры

**Процедура** – это отдельная часть кода, которая воспринимает входные данные, выполняет определенные действия и, возможно, возвращает результат в вызывающую программу.

#### Директивы оформления процедур

Заголовок процедуры имеет вид:

метка **Proc** язык тип *uses* регистры

Все параметры необязательны.

**Tun** может быть *near* или *far*. **Tun near** (ближний) означает, что меняется значение только регистра *IP* – это внутрисегментный переход. **Tun far** (дальний) означает, что меняются значения регистров *CS* и *IP* – это межсегментный переход.

**Метка** - имя процедуры.

Директива конца процедуры имеет вид

метка **Endp**

Она отмечает конец процедуры, начатой директивой **Proc** с той же меткой.

Директивы **Proc** и **Endp** не генерируют кода. Они управляют способом вызова процедуры и способом возврата из процедуры.

#### Вызов процедуры и возврат из процедуры

Вызов подпрограмм производится при помощи команды **call**. Формат команды **call** имеет следующий вид:

**call операнд**

Операнд – это либо метка, либо регистр или переменная, содержащие адрес перехода.

Команда **call** сохраняет в стеке *IP*, если указан тип процедуры *near*, или *IP* и *CS*, если указан тип процедуры *far*, и заносит адрес следующей команды в стек.

Возврат из процедуры осуществляется при помощи команды **RET**, которая имеет вид;

**ret число**

Число – необязательный операнд. Если он присутствует, то после считывания адреса возврата из стека будет удалено указанное количество байтов. Это необходимо, если при вызове процедуры ей передавались параметры через стек.

Если указан тип процедуры *near*, то при возврате из процедуры команда **RET** заменяется командой **RETN** (возврат ближнего типа), которая считывает из вершины стека только одно значение – *IP*, занесенное командой **CALL**.

Если указан тип процедуры *far*, то при возврате из процедуры команда *RET* заменяется командой *RETF* (возврат дальнего типа), которая считывает из вершины стека два значения: *IP* и *CS*.

### Передача параметров в процедуру

Параметры в процедуру передаются либо через регистры, либо через стек. При передаче через регистры значения помещаются в соответствующие регистры перед вызовом процедуры. При передаче через стек параметры помещаются в стек до выполнения команды *CALL*.

Для обращения к параметрам, находящимся в стеке, применяется адресация по базе со сдвигом с базовым регистром *BP*, так как при этом по умолчанию используется сегментный регистр стека *SS*. Одной из первых команд в процедуре должна быть команда, которая загружает регистр *BP* текущим значением указателя стека *SP*.

Пример.

Написать подпрограмму, которая вычисляет сумму элементов массива. Тип элементов массива - слово. Тип процедуры – дальний. Код программы находится в одном сегменте, а стек и данные – в другом.

Решение.

Будем передавать параметры через стек. Для этого в основной программе занесем в стек адрес начала массива и количество элементов массива. Заполнение стека в момент вызова процедуры будет таким:

$Bp=Sp$	<i>ip</i>	Так как тип процедуры <i>far</i> , то сохраняются регистры <i>IP</i> и <i>CS</i>
$Bp+2$	<i>cs</i>	
$Bp+4$	Количество элементов	
$Bp+6$	Адрес начала массива	
	...	

В процедуре скопируем в регистр *bp* значение указателя *sp*. Теперь для обращения к параметру в процедуре можно использовать адресацию по базе со сдвигом. Сдвиг – это число, соответствующее расстоянию в байтах от параметра до вершины стека. Сумму элементов массива будем хранить в регистре *ax*, предварительно обнулив его. Для доступа к элементу массива будем применять адресацию по базе с индексированием. Элементы массива находятся в сегменте данных. По умолчанию обращение к этому сегменту выполняется для базового регистра *bx*. Поэтому надо скопировать адрес начала массива из стека в регистр *bx*. Для обращения к текущему элементу массива будем использовать регистр *si*.

Программа

*.model small*

```

.code
;описание процедуры
pp    proc far
      mov bp,sp
      mov cx,[bp+4]      ; в cx – количество элементов массива
      mov bx,[bp+6]      ; в bx – адрес начала массива
      xor ax,ax          ; обнуление суммы
      xor si,si          ; в si – индекс текущего элемента
m:    add ax,[bx+si]      ;добавление элемента массива к сумме
      add    si,2        ; увеличение индекса на длину
                           ;элемента массива

      loop m
      ret 4              ; удаление параметров из стека и возврат управления в
                           ; вызывающую программу
pp    endp
n:    mov ax,@data
      mov ds,ax
      lea dx,mass        ; занесем в регистр dx адрес начала массива
      push dx            ; адрес массива – в стек
      mov ax,10          ; занесем в регистр ax количество элементов массива
      push ax            ; количество элементов массива – в стек
      call pp ; вызов процедуры
      mov sum,ax
      mov ax,4c00h
      int 21h

.data
mass dw 1,2,3,4,5,6,7,8,9,10
sum dw 0
      end n

```

### Задания

Написать процедуру, которая выполняет указанное действие. Параметры передаются через стек. Вызвать процедуру два раза для различных массивов.

1. Удвоить положительные элементы массива. Тип элементов массива –*byte*. Тип процедуры – дальний.
2. Обнулить положительные элементы массива. Тип элементов массива –*word*. Тип процедуры – дальний.
3. Уменьшить в 4 раза положительные элементы массива. Тип элементов массива –*byte*. Тип процедуры – дальний.
4. Обнулить отрицательные элементы массива. Тип элементов массива –*word*. Тип процедуры – дальний.
5. Удвоить отрицательные элементы массива. Тип элементов массива –*byte*. Тип процедуры – дальний.
6. Заменить положительные элементы массива единицами. Тип элементов массива –*byte*. Тип процедуры – дальний.
7. Заменить отрицательные элементы нулями. Тип элементов массива –*word*. Тип процедуры – ближний.
8. Найти количество положительных элементов массива. Результат вернуть через аккумулятор. Тип элементов массива –*word*. Тип процедуры – дальний.
9. Найти количество отрицательных элементов массива. Результат вернуть через аккумулятор. Тип элементов массива –*byte*. Тип процедуры – ближний.
10. Найти сумму положительных элементов массива. Результат вернуть через аккумулятор. Тип элементов массива –*byte*. Тип процедуры – дальний.
11. Найти сумму отрицательных элементов массива. Результат вернуть через аккумулятор. Тип элементов массива –*word*. Тип процедуры – ближний.
12. Найти максимальный элемент массива. Результат вернуть через аккумулятор. Тип элементов массива –*word*. Тип процедуры – дальний.
13. Найти минимальный элемент массива. Результат вернуть через аккумулятор. Тип элементов массива –*word*. Тип процедуры – дальний.
14. Найти количество элементов массива, которые больше заданного значения. Результат вернуть через аккумулятор. Тип элементов массива –*word*. Тип процедуры – дальний.

## Лабораторная работа № 6

### Прерывания

Прерывание – это временное прекращение выполнения текущей программы с передачей управления другой программе (обработчику прерывания).

#### Внутренние прерывания

Причины внутренних прерываний – это либо ненормальное внутренне состояние, возникающее при выполнении некоторых команд микропроцессора, либо обработка команды *INT*.

Система обработки прерывания включает в себя следующие программные средства:

1. Таблица векторов прерываний. Таблица векторов прерываний находится по адресу 0000:0000. Вектор прерывания - это адрес программы обработчика прерывания. Каждый вектор представляет собой двойное слово. Первое слово содержит смещение, а второе слово - сегментный адрес обработчика. Всего таблица содержит 256 векторов.
2. Команда *INT*. Служит для вызова программного прерывания. Имеет вид  
 $INT\ n$ , где  $n$  - это номер прерывания.
3. Команда *IRET*. Служит для возврата из обработчика прерывания.
4. Флаг *IF*. Служит для разрешения или запрета прерываний. При  $IF=1$  (аппаратные) прерывания разрешены, при  $IF=0$  - прерывания запрещены.
5. Команда *STI*. Служит для установки *IF* в 1.
6. Команда *CLI*. Служит для сброса *IF* в 0.

Прерывание должно быть выполнено так, чтобы после возврата из обработчика прерывания прерванная программа выполнялась так, как если бы не было никакого прерывания. Для этого требуется сохранение контекста прерванной программы. Это значит, что все регистры микропроцессора, включая регистр флагов, должны иметь те же значения, которые они имели перед прерыванием. При этом сохранение значений регистра флагов, регистра *CS* и регистра *IP* выполняется системой, а сохранение регистров, изменяемых программой обработки прерывания возлагается на программиста.

#### Перехват прерывания

Перехват прерывания – это замена стандартного обработчика собственным обработчиком прерывания. Замену можно выполнить, обращаясь непосредственно к таблице векторов прерываний, или при помощи специальных функций прерывания *21h Dos*.

Пример:



Заменить обработчик прерывания номер 87h собственным обработчиком, который выводит на экран символ '1', обращаясь напрямую к таблице векторов прерываний. По завершении работы восстановить стандартный обработчик прерывания.

#### Решение

Мы хотим заменить обработчик прерывания 87h. Вначале надо сохранить старый вектор прерывания. Длина каждого вектора 4 байта. Значит, смещение вектора в таблице векторов прерываний можно получить, умножив номер вектора на 4. В момент, когда выполняется запись в таблицу векторов нового значения, должны быть запрещены все прерывания. После того, как в таблицу будет занесено новое значение, прерывания можно разрешить. В конце работы восстановим в таблице вектор стандартного обработчика.

*.model tiny*

*.code*

```
sofs    dw    ?      ; здесь сохраним смещение стандартного обработчика
sseg    dw    ?      ;здесь сохраним сегментный адрес стандартного обработчика
s:      push cs
        pop  ds
```

```
mov ax,0
```

```
mov es,ax      ; в es – сегментный адрес таблицы векторов прерываний
```

```
mov dx,0
```

*;сохранение старого вектора прерывания*

```
mov ax,es:[87*4]      ; в ax – смещение вектора прерывания 87h
```

```
mov sofs,ax
```

```
mov ax,es:[87*4+2] ;в ax – сегментный адрес вектора прерывания 87h
```

```
mov sseg,ax
```

```
mov ax,offset pp ; в ax – смещение нового обработчика прерывания
```

```
mov bx,seg pp ; в dx – сегментный адрес нового обработчика прерывания
```

*;замена вектора прерывания*

```
cli      ; запрет прерываний
```

```
mov es:[87*4],ax ; занесли в таблицу векторов прерываний
```

```
mov es:[87*4+2],bx
```

```
sti      ; разрешение прерываний
```

*int 87 ; вызов нового обработчика*

*mov ax,sseg*

*mov bx,sofs*

*;восстановление вектора прерывания стандартного обработчика*

*cli*

*mov es:[87\*4+2],ax*

*mov es:[87\*4],bx*

*sti*

*; завершение программы*

*mov ax,4c00h*

*int 21h*

*; новый обработчик прерывания 87h*

*nov\_int proc*

*mov ah,02h*

*mov dl,35h*

*int 21h*

*iret*

*nov\_int endp*

*end s*

Для замены обработчика прерывания применяются две функции прерывания *INT 21h*.

***Функция 35h - получить адрес обработчика прерывания.***

Вход	<i>AH</i>	<i>35h</i>
	<i>AL</i>	номер обработчика.
Выход	<i>ES</i>	сегментный адрес обработчика прерывания.
	<i>BX</i>	смещение адреса обработчика прерывания

***Функция 25h - установить обработчик (заменить адрес обработчика прерывания в таблице векторов прерывания).***

Вход	<i>AH</i>	<i>25h</i>
	<i>AL</i>	номер заменяемого обработчика
	<i>DS</i>	сегментный адрес обработчика прерывания
	<i>DX</i>	смещение адреса обработчика прерывания

Выход	Замена обработчика в таблице векторов прерываний
-------	--

Пример:

Заменить обработчик прерывания номер *87h* собственным обработчиком, который выводит на экран символ '1'. По завершении работы восстановить стандартный обработчик прерывания.

Решение

Сначала получим адрес обработчика прерывания *87h* при помощи функции функций *35h* и сохраним этот адрес. Замену обработчика прерывания будем выполнять при помощи функций *25h* прерывания *21h DOS*. В конце работы при помощи этой же функции восстановим в таблице вектор стандартного обработчика.

Программа

*.model tiny*

*.code*

*st\_off dw 0; здесь сохраним смещение стандартного обработчика*

*st\_seg dw 0; здесь сохраним сегментный адрес стандартного обработчика*

*n: push cs*

*pop ds*

*; получить адрес обработчика 87h*

*; в es будет записан сегментный адрес стандартного обработчика*

*; в bx будет записано смещение стандартного обработчика*

*mov ah,35h*

*mov al,87h*

*int 21h ;вызов функции 35h*

*mov st\_seg,es ;сохранить сегментный адрес обработчика87h*

*mov st\_off,bx ;сохранить смещение обработчика 87h*

*;перед вызовом функции 25h в регистрах ds и dx должны находиться*

*;соответственно сегментный адрес и смещение нашего обработчика*

*;в ds уже находится сегментный адрес нашего обработчика*

*lea dx,prog ;в dx занесем смещение нашего обработчика*

*;заменить обработчик нашим обработчиком*

*mov ax,2587h*

*int 21h ;вызов функции 25h*

*int 87h ;вызов прерывания 87h*

*;восстановить стандартный обработчик*

*mov ax,2587h*

*mov dx,st\_off ; в dx – смещение стандартного обработчика*

*mov ds,st\_seg ; в dx - сегментный адрес стандартного обработчика*

*int 21h ;вызов функции 25h*

*mov ax,4c00h*

*int 21h*

*;Это наш собственный обработчик прерывания*

*prog proc*

*; вывод на экран цифры '1'*

*mov ah,02h ; функция вывода символа на экран*

*mov dl,31h ; в dl занесли код выводимого символа*

*int 21h*

*iret ;возврат управления прерванной программе*

*prog endp*

*end n*

### **Задание**

1. Написать собственный обработчик программного прерывания *87h* и заменить им стандартный обработчик, обращаясь непосредственно к таблице векторов прерываний. Необходимо предусмотреть сохранение и восстановление регистров, изменяемых обработчиком прерывания. Этот обработчик должен выводить на экран символ 'А'.
2. Написать собственный обработчик программного прерывания *5h* и заменить им стандартный обработчик, используя специальные функции прерывания *21h DOS*. Необходимо предусмотреть сохранение и восстановление регистров, изменяемых обработчиком прерывания. Этот обработчик должен выводить на экран сообщение вида:

*«Этот обработчик написал студент фамилия»*

### **Прерывания от внешних устройств**

Прерывания от внешних устройств или аппаратные прерывания вызываются причинами, внешними по отношению к микропроцессору. Аппаратное прерывание – это запрос от некоторого периферийного устройства на обслуживание. Обработчик прерывания выполняет необходимые

действия для получения данных от периферийного устройства или для управления им и возвращает управление в прерванную программу.

Всего используется 15 аппаратных прерываний. Например:

*INT 8* – прерывание системного таймера.

*INT 9* - прерывание клавиатуры.

Аппаратные обработчики целиком не заменяются. Заменяющие обработчики обычно передают управление стандартному обработчику, который вызывается либо до, либо после тела заменяющего обработчика.

Пусть адрес стандартного обработчика сохраняется в ячейках *st\_off* и *st\_seg*:

*st\_off dw 0*; смещение стандартного обработчика

*st\_seg dw 0*; сегментный адрес стандартного обработчика

Если стандартный обработчик вызывается до выполнения тела заменяющего обработчика, то процедура обработки прерывания имеет следующий вид:

*nov\_int proc*

*pushf* ; сохранение регистра флагов

*call cs:dword ptr st\_off* ; вызов стандартного обработчика

*<тело заменяющего обработчика>*

*iret* ; возврат управления прерванной программе

*nov\_int endp*

Если стандартный обработчик вызывается после выполнения тела заменяющего обработчика, то процедура обработки прерывания имеет следующий вид:

*nov\_int proc*

*<тело заменяющего обработчика>*

*jmp cs:dword ptr st\_off* ; безусловная передача управления стандартному ;обработчику

*nov\_int endp*

### Обращение к видеопамяти

В теле обработчика, заменяющего аппаратный обработчик прерывания, нельзя использовать функции прерывания *INT 21H*. Потому для вывода сообщений можно применять прямое обращение к видеопамяти. Видеопамять – это область памяти по адресу *0b800h:0000h*. Каждое слово видеопамяти соответствует одному знакоместу на экране. Младший байт слова определяет выводимый символ, а старший байт – цвет символа. Например, байт по адресу *0b800h:0000h* – это символ, отображаемый в верхнем левом углу экрана. Байт по адресу *0b800h:0001h* задает цвет этого символа. Старшие четыре бита определяют цвет фона, а младшие – цвет символа.

Название цвета фона	номер	16-ричный номер	Название цвета фона или символа	номер	16-ричный номер
Черный	0	0	Темно-серый	8	8
Синий	1	1	светло-синий	9	9
зеленый	2	2	светло-зеленый	10	A
бирюзовый	3	3	светло-бирюзовый	11	B
красный	4	4	светло-красный	12	C
малиновый	5	5	светло-малиновый	13	D
Коричневый	6	6	желтый	14	E
серый	7	7	белый	15	F

Например, вывести букву «А» в верхнем левом углу экрана можно при помощи следующих команд:

```
mov ax,0b800h      ; в ax – адрес видеопамати
mov es,ax          ; занесли адрес видеопамати в сегментный регистр es
mov byte ptr es:[0000], 'a' ; вывели на экран символ
mov byte ptr es:[0001], 1fh ; задали цвет символа - белый на синем
```

Для вывода информации в заданную позицию на экране необходимо вычислить смещение от начала видеопамати до нужной ячейки. Например, надо вывести символ '\*' в начале шестой строки экрана. Длина строки – 80 символов. Под каждый символ отводится одно слово видеопамати, т.е. 2 байта. Значит, под одну строку отводится  $80 \times 2 = 160$  байтов. До начала шестой строки надо пропустить 5 строк. Таким образом, смещение, соответствующее началу шестой строки равно  $160 \times 5 = 800$  байтов.

```
mov ax,0b800h      ; в ax – адрес видеопамати
mov es,ax          ; занесли адрес видеопамати в сегментный регистр es
mov byte ptr es:[80*2*5], '*' ; вывели на экран символ
mov byte ptr es:[80*2*5+1], 4eh ; цвет символа - желтый на красном фоне
```

### Задание

1. Написать собственный обработчик прерывания от клавиатуры. Стандартный обработчик вызывается после тела собственного обработчика. Необходимо предусмотреть сохранение и восстановление регистров, изменяемых обработчиком прерывания. Этот обработчик должен выводить на экран цифру «5»

- a. Белым на красном фоне
  - b. Синим на бирюзовом фоне
  - c. Желтым на красном фоне
  - d. Белым на синем фоне
  - e. Желтым на синем фоне
2. Написать собственный обработчик прерывания от клавиатуры. Стандартный обработчик вызывается до тела собственного обработчика. Необходимо предусмотреть сохранение и восстановление регистров, изменяемых обработчиком прерывания. Этот обработчик должен выводить на экран символ «\*»
- a. в центре экрана,
  - b. в левом верхнем углу,
  - c. в правом верхнем углу,
  - d. в левом нижнем углу,
  - e. в правом нижнем углу,
3. Написать собственный обработчик прерывания от клавиатуры. Стандартный обработчик вызывается до тела собственного обработчика. Необходимо предусмотреть сохранение и восстановление регистров, изменяемых обработчиком прерывания. Этот обработчик должен выдавать звуковой сигнал.

## **Лабораторная работа № 7**

### **Макросы**

Макрос – это участок программы, которому присвоено имя и который ассемблируется каждый раз, когда ассемблер встречает это имя в тексте программы.

Макрорасширение – это последовательность строк, которая может вставляться в текст программы при помощи специальных команд.

#### **Синтаксис макроопределения:**

Имя ***macro*** аргументы

Тело макроопределения

***Endm***

Макрорасширение вставляется в текст программы каждый раз когда встречается макрос. Текст макрорасширения может изменяться в соответствии с параметрами макроса.

### **Макрооператоры**

В макроопределениях можно применять макрооператоры, которые позволяют модифицировать текст макрорасширения.

Макрооператор **&**. Параметры, обрамленные знаками **&**, должны заменяться значениями до обработки строк ассемблером (до ассемблирования). Если эти параметры указываются в конце оператора, то второй знак **&** можно опустить. Например, если макроопределение имеет вид

```
move      macro p1,p2,p3
    mov p1&p3,p2&p3
endm,
```

то можно получить следующие макрорасширения:

Вызов макроса	Макрорасширение
<i>move a, b, h</i>	<i>mov ah,bh</i>
<i>move d c x</i>	<i>mov dx,cx</i>

Макрооператор **%** ставят перед выражениями, значения которых должны быть вычислены и переданы в макрос в качестве параметров. Например, если макроопределение имеет вид

```
sum macro p1,l, %p2
    add p1&l,%p2
endm,
```

то можно получить следующие макрорасширения:

Вызов макроса	Макрорасширение
<i>sum a, x, 100/4</i>	<i>add ax,100/4.</i>
<i>sum d h %25*3</i>	<i>add dh,75</i>

Макрооператор **< >** предписывает рассматривать текст в скобках как текстовую строку.

Например, если макроопределение имеет вид

```
op macro p1,l, kom
    kom&l,p1&l
endm,
```

то можно получить следующие макрорасширения:

Вызов макроса	Макрорасширение
<i>op a, x, &lt;sub a&gt;</i>	<i>sub ax,ax</i>
<i>op d h &lt;add b&gt;</i>	<i>add bh,dh</i>

Макрооператор **::** отмечает начало макрокомментария.



Если в макросе встречается метка, то она должна быть объявлена как локальная при помощи директивы ***local***.

#### ***Local*** список меток

Элементы списка разделяются запятыми. Например, макрос для суммирования элементов массива может иметь следующий вид:

```
sum macro sum,mas,ind,s
```

```
    local met
```

```
    mov cx,s
```

```
    xor sum,sum
```

```
    xor ind,ind
```

```
    met:  add sum,mas&[ind]
```

```
    inc ind
```

```
    loop met
```

```
endm
```

#### **Задания**

1. Создать макрос для вывода строки на экран.
2. Создать макрос для ввода строки с клавиатуры.
3. Создать макрос для поиска символа в строке.
4. Создать макрос для поиска максимального элемента в массиве.
5. Создать макрос для поиска минимального элемента в массиве.
6. Создать макрос для упорядочивания элементов по возрастанию.
7. Создать макрос для упорядочивания элементов по убыванию.

## Лабораторная работа № 8

### Включение файлов в текст программы

Директива **INCLUDE** вставляет в текст программы текст указанного файла. Используется обычно для включения файлов, содержащих определения макросов, констант и т.п.

Синтаксис: **Include** имя файла

Например, пусть файл *summa.asm* содержит определение макроса *sum*:

```
Sum macro p1,p2,p3,r
```

```
    Push a&r
```

```
    Mov a&r,p1
```

```
    Add a&r,p2
```

```
    Mov p3,a&r
```

```
    Pop a&r
```

```
endm
```

Для того чтобы включить в текст программы это определение, воспользуемся директивой *include*.

```
.model tiny
```

```
.code
```

```
include summa.asm
```

```
N:    Push cs
```

```
    Pop ds
```

```
    Sum a,b,c,l
```

```
    Mov ax,4c00h
```

```
    Int 21h
```

```
.data
```

```
A db 10
```

```
B db 30
```

```
C db ?
```

```
End N
```

### Связь Ассемблер – Ассемблер

При работе на ассемблере можно создавать многомодульные программы. Модуль – это функционально автономный объект. Для связи модулей используются директивы **public** и **extrn**.

Директива **public** предназначена для объявления некоторого имени, определенного в данном модуле и видимого в других модулях.

### **Public имя**

Директива **extrn** предназначена для объявления некоторого имени внешним по отношению к данному модулю. Это имя в другом модуле должно быть объявлено как **public**.

### **Extrn имя:тип**

Имя - это идентификатор, объявленный в другом модуле. Если это

- Имя переменной, определенной одной из директив **db, dw** и т.д., тип может быть **byte, word, dword, qword** и т.д.
- Имя процедуры. тип может быть **near** или **far**.
- имя константы, определенной оператором **=** или **equ**, то тип должен быть **abs**.

Настройка модулей выполняется на этапе компоновки- редактирования связей. Сначала надо получить объектные модули программ, а затем объединить их.

**Tlink** <имя 1.obj> + <имя 2.obj>

При объединении нескольких модулей первым должен идти главный модуль, а остальные – в произвольном порядке.

Пример.

Создать вспомогательный модуль, содержащий процедуру **print** для вывода символа на экран. Вызвать эту процедуру из основного модуля для вывода переменной **b**. Код и данные должны находиться в одном сегменте.

Решение.

В вспомогательном модуле процедура **print** должна быть объявлена как **public**, а переменная **b** как **extrn** с типом **byte**, так как через нее будут передаваться печатаемые символы. В основном модуле **b** должна быть объявлена как **public**, а процедура **print** как **extrn** с типом **near**.

Если оба модуля используют один сегмент данных, то получим модули следующего вида:

Вызывающий модуль	Вызываемый модуль
<code>code segment</code> <code>assume cs:code,ds:data</code> <code>public b</code> <code>extrn print:near</code> <code>n: mov ax,data</code> <code>mov ds,ax</code> <code>call print</code> <code>mov ax,4c00h</code>	<code>public print</code> <code>extrn b:byte</code> <code>codep segment</code> <code>print proc</code> <code>assume cs:codep</code> <code>mov dh,b</code> <code>mov ah,02</code> <code>int 21h</code>

<pre> int 21h code ends data segment b      db      'a' data ends end n </pre>	<pre> ret print endp n1: codep ends end n1 </pre>
--	---

Если у каждого модуля есть собственный сегмент данных, то во вспомогательном модуле необходимо получить адрес сегмента, в котором находятся передаваемые для печати данные

Вызывающий модуль	Вызываемый модуль
<pre> code segment     assume cs:code,ds:data     public b     extrn print:near n:    mov ax,data         mov ds,ax         call print         mov ax,4c00h         int 21h code ends data segment b      db      'a' data ends end n </pre>	<pre> public print extrn b:byte codep segment print proc     assume cs:codep,ds:datap     mov ax,seg b     mov es,ax     mov ah,02     mov dl,es:b     int 21h     ret print endp n1:    mov ax,datap         mov ds,ax         mov ax,4c00h         int 21h codep ends datap segment a      db      100 datap ends end n1 </pre>

Программы на языках высокого уровня можно комбинировать с программами на языке ассемблера. Существует несколько способов связи: встроенный ассемблер и использование внешних процедур.

**Встроенный ассемблер.** При этом команды ассемблера включаются в текст программы на языке C. Ассемблерная вставка начинается с ключевого слова `__asm` и может использоваться везде, где может стоять оператор языка C. Например:

```
int x=10;
```

```
__asm mov eax,x.
```

Команды на языке ассемблера заключаются в фигурные скобки. Например:

```
__asm{  
    mov ax,10  
    add ax,bx  
}
```

Допускается указывать ключевое слово `__asm` перед каждой инструкцией на ассемблере. В этом случае можно указать несколько операторов в одной строке. Например:

```
__asm add ax,5 __asm mov x,eax
```

В ассемблерных вставках

- недопустимо объявление переменных при помощи директив **db**, **dw**, **dd**.
- Недопустимо использование структур и записей.
- Недопустимо объявление макросов.
- Допускается использование любого выражения, результатом которого является число или адрес.
- При обращении к сегментам следует указывать сегментные регистры, а не имена сегментов.
- Блоки могут содержать комментарии в стиле языка C.
- Псевдокоманда `_emit` определяет один байт в текущей позиции в текущем сегменте. При помощи этой директивы можно вставить в текст программы машинные коды. Например:

```
__asm _emit 0xB0 //mov al,5
```

Чтобы получить машинные коды, надо выполнить ассемблирование файла, содержащего программу с нужными командами. А затем из листинга можно выделить коды команд, которые и будут вставлены в программу на языке C.

### Задание

1. Создать файл, содержащий текст процедуры, которая выполняет нахождение обнуление отрицательных элементов массива, состоящего из значений типа word. Тип процедуры – ближний.

Вставить этот файл в программу при помощи директивы *include*. Вызвать процедуру для вычисления суммы элементов заданного массива.

2. Создать файл *2.asm*, содержащий процедуру, которая печатает строку символов. Вызвать эту процедуру из программы, находящейся в модуле *1.asm*.

3. В программе на языке *C* выполнить суммирование элементов массива, состоящего из целых чисел при помощи ассемблерной вставки..

4. В программу на языке *C* вставить машинные коды команд, которые увеличивают на 1 значения регистров *ax* и *dl*.