Московский государственный технический университет им. Н.Э.Баумана

Г. С. Иванова

ПРОГРАММИРОВАНИЕ НА АССЕМБЛЕРЕ ПЭВМ

**Методические указания к лабораторным работам
по курсу “Системное программирование”**

Часть 1

Машинные команды ассемблера
Приемы программирования

МОСКВА 1999

1. СОДЕРЖАНИЕ

2. АРХИТЕКТУРА МИКРОПРОЦЕССОРА i8086

2.1. Структурная схема микропроцессора i8086

На рис. I представлена структурная схема микропроцессора 8086, в состав которого входят: устройство управления (УУ), арифметико-логическое устройство (АЛУ), блок преобразования адресов и регистры.

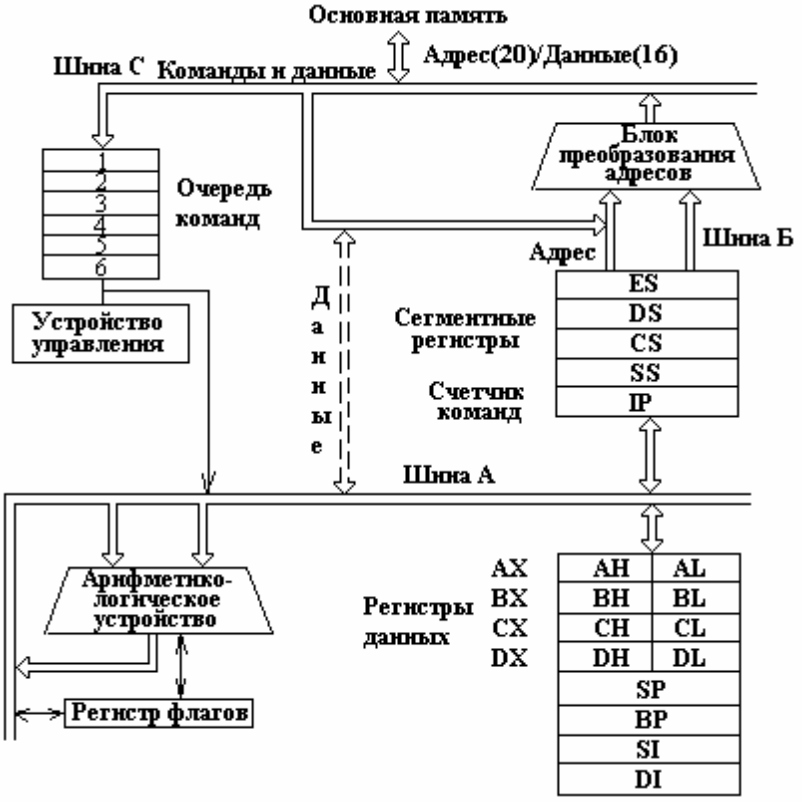


Рис. 1.

Устройство управления дешифрирует коды команд и формирует необходимые управляющие сигналы. Арифметико-логическое устройство осуществляет необходимые арифметические и логические преобразования данных. В блоке преобразования адресов формируются физические адреса данных, расположенных в основной памяти. Наконец, регистры используются для хранения управляющей информации: адресов и данных.

Всего в состав микропроцессора i8086 входит четырнадцать 16-битовых регистров (см. рис. 1):

а) четыре регистра общего назначения (регистры данных):

AX - регистр-аккумулятор,
BX - базовый регистр,
CX - счетчик,
DX - регистр-расширитель аккумулятора;

б) три адресных регистра:

SI - регистр индекса источника,
DI - регистр индекса результата,
BP - регистр-указатель базы;

в) три управляющих регистра:

SP - регистр-указатель стека,
IP - регистр-счетчик команд,
регистр флагов;

г) четыре сегментных регистра:

CS - регистр сегмента кодов,
DS - регистр сегмента данных"
ES - регистр дополнительного сегмента данных,
SS - регистр сегмента стека.

2.2. Организация основной памяти

Минимальной адресуемой единицей основной памяти ПЭВМ является *байт*, состоящий из 8 бит. Доступ к байтам основной памяти осуществляется по номерам (номер байта является ого физическим адресом в устройстве памяти).

Для адресации основной памяти в микропроцессоре i8086 предусматриваются 20-битовые адреса, что позволяет работать с основной памятью до 1 Мбайта.

Физический адрес формируется из 16-битового смещения и содержимого 16-битового сегментного регистра, сдвинутого влево на 4 бита (см. рис. 2).

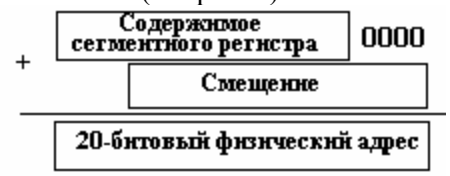


Рис. 2.

Для размещения программ и данных в основной памяти выделяются специальные области - сегменты. Адреса этих областей хранятся в специальных сегментных регистрах.

Каждый из четырех сегментных регистров используется для хранения адреса определенного сегмента (см. рис. 3):

- сегмента кодов, т. е. области программ;
- сегмента данных, т. е. области размещения данных;
- дополнительного сегмента данных, используемого некоторыми командами;
- сегмента стека, т.е. области размещения стека.

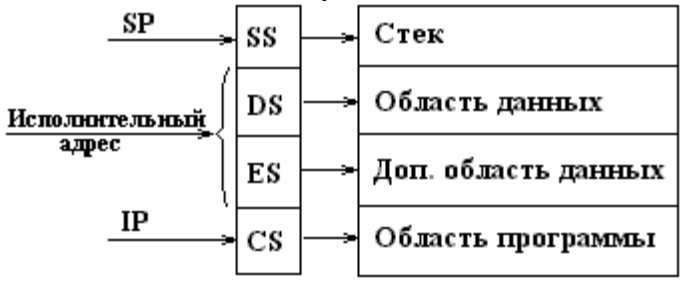


Рис. 3.

Стек представляет собой специальным образом организованную область памяти, допускающую последовательную запись элементов данных длиной 2 байта (слово) и чтение их в порядке, обратном порядку записи. Для хранения адреса последнего слова, занесенного в стек, служит регистр-указатель стека **SP** (см. рис. 4, где а - текущее состояние стека, б - запись X, в - чтение X).

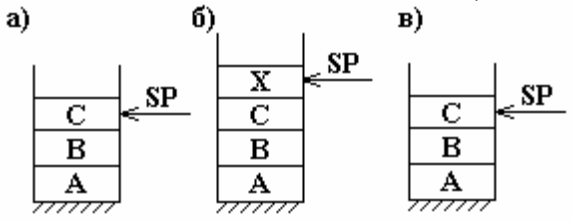


Рис. 4

Стек используется для временного хранения данных и адресов, например при вызове подпрограмм, когда в стек заносится адрес возврата и значения параметров, передаваемых в подпрограмму.

Формат команд микропроцессора 8086 позволяет указывать в команде только один операнд, размещенный в основной памяти, т. е. одной командой нельзя, например, сложить содержимое двух ячеек памяти,

Принципиально допускается 8 способов задания смещения (исполнительного адреса) операндов, размещенных в основной памяти:

SI + <индексное смещение>

DI + <индексное смещение>

BP + <индексное смещение>

BX + <индексное смещение>

BP + **SI** + < индексное смещение>
BP + **DI** +< индексное смещение>
BX + **SI** + <индексное смещение>
BX + **DI** + <индексное смещение>

Во всех случаях исполнительный адрес операнда определяется как сумма содержимого указанных регистров и индексного смещения, представляющего собой некоторое число (одно- или двухбайтовое).

2.3. Выполнение программы

Содержимое регистров **CS** и **IP**, в которых хранится базовый адрес сегмента кодов и смещение очередной команды относительно начала сегмента, определяет физический адрес команды, которая должна быть выполнена на следующем шаге.

По указанному адресу из основной памяти считывается команда и пересылается в микропроцессор. Команда длиной от 1 до 8 байт помещается в очередь команд, откуда поступает в устройство управления, где дешифрируется.

Если при выполнении команды требуются данные, расположенные в основной памяти, то специальные полем кода команды определяется способ адресации и вычисляется исполнительный, и затем и физический адрес данных.

Данные, считанные из основной памяти по указанному адресу, пересылаются в регистр данных или в арифметико-логическое устройство и обрабатываются в соответствии с кодом команды. Результат помещается либо в регистры, либо (в соответствии с командой) в какую-либо область основной памяти.

Если выполненная команда не являлась командой передачи управления, то содержимое регистра **IP** увеличивается на длину выполненной команды, в противном случае в регистр **IP** заносится исполнительный адрес команды, которая должна выполняться следующей.

Затем процесс повторяется.

2.4. Флажковый регистр

На рис. 5 представлен флажковый регистр микропроцессора i8086, в котором в виде однобитовых признаков по принципу ДА - НЕТ (ВКЛЮЧЕНО - ВЫКЛЮЧЕНО) фиксируется информация о результатах выполнения некоторых команд, например арифметических.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
				O	D	I	T	S	Z		A		P	C

Рис. 5

O - признак переполнения;
D - признак направления;
I - признак прерывания;
T - признак трассировки;
S - признак знака: 1 - число < 0, 0 - число > 0
Z - признак нуля: 1 - число = 0
A - признак переноса из тетрады;
P - признак четности;
C - признак переноса.

В последующем эта информация может использоваться, например, командами условной передачи управления.

3. АССЕМБЛЕР ПЭВМ

3.1. Формат операторов ассемблера

Операторы языка ассемблера ПЭВМ имеют следующий формат:

[<метка> :] <код операции > [<список операндов >1 [<комментарии>].

Запись программы выполняется по свободному формату, т. е. специально не оговариваются правила заполнения каких бы то ни было позиций строки. Точка с запятой в начале строки означает, что данная строка является строкой комментария.

Программа может записываться как заглавными, так и строчными буквами. Метку произвольной длины следует записывать с начала строки и отдалять от кода операции двоеточием, за которым может следовать произвольное количество пробелов (вплоть до конца строки).

Код операции должен отделяться от списка операндов хотя бы одним пробелом. Операнды отделяются один от другого запятой.

3.2. Определение полей памяти для размещения данных

Для определения данных в основной памяти и резервирования полей памяти под данные, размещаемые в основной памяти в процессе выполнения программы, используются следующие операторы:

DB - определить однобайтовое поле, **DW** - определить слово (двухбайтовое поле), **DD** - определить двойное слово (четырехбайтовое поле).

Формат команды:

DB

[<имя поля>] **DW** [< количество > **DUP** (]{ <список чисел >}[ )]

DD ?

где <количество >- количество полей памяти указанной длины, которое определяется данной командой (указывается, если определяется не одно поле памяти); ? - используется при резервировании памяти.

Приведем примеры.

1. Записать в байт памяти десятичное число 23 и присвоить этому байту имя a:
   a db 23.
2. Зарезервировать 1 байт памяти: **db ?**
3. Записать в слово памяти шестнадцатеричное число 1234: **dw 1234H.**
4. Определить 31 байт памяти, повторяя последовательность 1, 2, 3, 4, 5, 1, 2, 3, 4,... :
   db 31 dup (1,2,3,4,5)

Примечание. При записи слов в память младший байт записывается в поле с младшим адресом. Например, в примере 3, если запись выполнялась по адресу 100, то по адресу 100 будет записано 34H, а по адресу 101 - 12H.

3.3. Операнды команд ассемблера

Операнды команд ассемблера могут определяться непосредственно в команде, находиться в регистрах или в основной памяти,

Данные, непосредственно записанные в команде, называются литералами. Так, в команде

mov ah, 3 3 - литерал.

Если операнды команд ассемблера находятся в регистрах, то в соответствующих командах указываются имена регистров (если используемые регистры особо не оговариваются для данной команды. Например, в приведенном выше примере **ah** - имя регистра аккумулятора.

Адресация операндов, расположенных в основной памяти, может быть прямой и косвенной.

При использовании прямой адресации в команде указывается символическое имя поля памяти, содержащего необходимые данные, например:

inc OPND

Здесь **OPND** - символическое имя поля памяти, определенного оператором ассемблера

OPND dw ?

При трансляции программы ассемблер заменит символическое имя на исполнительный адрес указанного поля памяти (смещение относительно начала сегмента) и занесет этот адрес на место индексного смеще-

ния. Адресация а этом случае выполняется по схеме: **BP** + <индексное смещение>, но содержимое регистра **BP** при вычислении исполнительного адреса не используется (частный случай).

В отличие от прямого косвенный адрес определяет не местоположение данных в основной памяти, а местоположение компонентов адреса этих данных. В этом случае в команде указываются один или два регистра в соответствии с допустимыми схемами адресации и индексное смещение, которое может задаваться числом или символическим именем. Косвенный адрес заключается в квадратные скобки весь или частично, например:

[OPND +SI]
OPND [SI]
OPND + [SI]
[OPND] +[SI]

Приведенные выше формы записи косвенного адреса интерпретируются одинаково.

При трансляции программы ассемблер определяет используемую схему адресации и соответствующим образом формирует машинную команду, при этом символическое имя заменяется смещением относительно начала сегмента так же, как в случае прямой адресации.

Примечание. При использовании косвенной адресации по схеме BP + <индексное смещение> индексное смещение не может быть опущено, так как частный случай адресации по данной схеме с нулевой длиной индексного смещения используется для организации прямой адресации. Следовательно, при отсутствии индексного смещения в команде следует указывать нулевое индексное смещение, т.е. [**BP** + **0**] .

Приведем два примера: [**a** + **bx**] и [**bp**]+[**si**] +**6**.

В первом случае исполнительный адрес операнда определяется суммой содержимого регистра **bx** и индексного смещения, заданного символическим именем "**a**", а во втором - суммой содержимого регистров **bp**, **si** и индексного смещения, равного 6.

Длина операнда может определяться:

а) кодом команды - в том случае, если используемая команда обрабатывает данные определенной длины, что специально оговаривается;

б) объемом регистров, используемых для хранения операндов (1 или 2 байта);

в) специальными указателями **byte ptr** (1 байт) и **word ptr** (2 байта), которые используются в тех случаях, когда длину операнда нельзя установить другим способом. Например,

mov byte ptr x, 255

т. е. операнд пересылается в поле с именем "**x**" и имеет длину I байт.

3.4. Команды пересылки / преобразования данных

3.4.1. Команда пересылки данных

MOV <адрес приемника> ,< адрес источника>

используется для пересылки данных длиной I или 2 байта из регистра в регистр, из регистра в основную память, из основной памяти в регистр, а также для записи в регистр или основную память данных, непосредственно записанных в команде. Все возможные пересылки представлены на рис. 6.

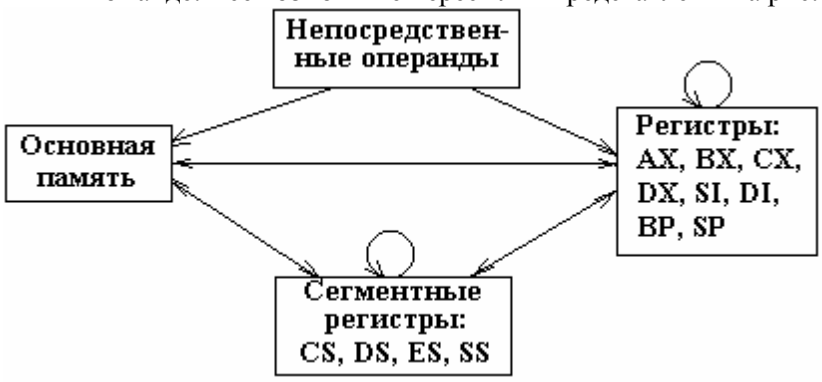


Рис. 6

Приведем примеры:

а) **mov ax, bx**- пересылка содержимого регистра **bx** в регистр **ax**;

б) **mov cx, exword** - пересылка 2 байт, расположенных в поле **exword**, из основной памяти в регистр **cx**;

в) **mov si, 1000** - запись числа 1000 в регистр **si**;

г) **mov word ptr [di+515], 4** - запись числа 4 длиной 2 байта в основную память по адресу **[di+515]**.

Для загрузки "прямого" адреса в сегментный регистр используются две команды пересылки:

```
mov   ax, code
mov   ds, ax
```

3.4.2. Команда обмена данных

XCHG <операнд 1> , <операнд 2>

организует обмен содержимого двух регистров (кроме сегментных) или регистра и поля основной памяти. Например:

xchg bx, cx - обмен содержимого регистров **bx** и **cx**.

3.4.3. Команда загрузки исполнительного адреса

LEA < операнд l > , < операнд 2 >

вычисляет исполнительный адрес второго операнда и помещает его в поле, на которое указывает первый операнд. Приведем примеры:

а) **lea bx, exword** - в регистр **bx** загружается исполнительный адрес **exword**;

б) **lea bx, [di+10]** - в регистр **bx** загружается адрес l0-го байта относительно точки, на которую указывает адрес в регистре **di**.

3.4.4. Команды загрузки указателя

LDS < регистр > ,<операнд 2>

LES < регистр > ,< операнд 2>

Команда **LDS** загружает в регистры **DS** :< регистр> указатель (< адрес сегмента > : < исполнительный адрес >), расположенный по адресу, указанному во втором операнде.

Команда **LES** загружает указатель по адресу, расположенному во втором операнде, в регистры **ES**:< регистр>. Например:

lds si, **exword**

т.е. слово (2 байта) по адресу **exword** загружается в **si**, а по адресу **exword**+ **2** - в **ds**.

3.4.5. Команда записи в стек

PUSH < операнд>

организует запись в стек слова, адрес которого указан в операнде. Например;

push dx - запомнить содержимое регистра **dx** в стеке.

3.4.6. Команда восстановления из стека

POP < операнд>

организует чтение из стека последнего слова и помещает его по адресу, указанному во втором операнде. Например:

pop dx - восстановить содержимое регистра **dx** из стека.

3.4.7. Команды сложения

ADD <операнд 1> , <операнд 2>

ADC <операнд 1> , <операнд 2>

устанавливают флаги четности, знака результата, наличия переноса, наличия переполнения.

Ilo команде **ADD** выполняется сложение двух операндов. Результат записывается по адресу первого операнда. По команде **ADC** также выполнятся сложение двух операндов, но к ним добавляется еще значение, записанное в бите переноса, установленном предыдущей командой сложения.

На рис. 7 показаны возможные способы размещения слагаемых, где а -операнды - слова, б - операнды - байты.

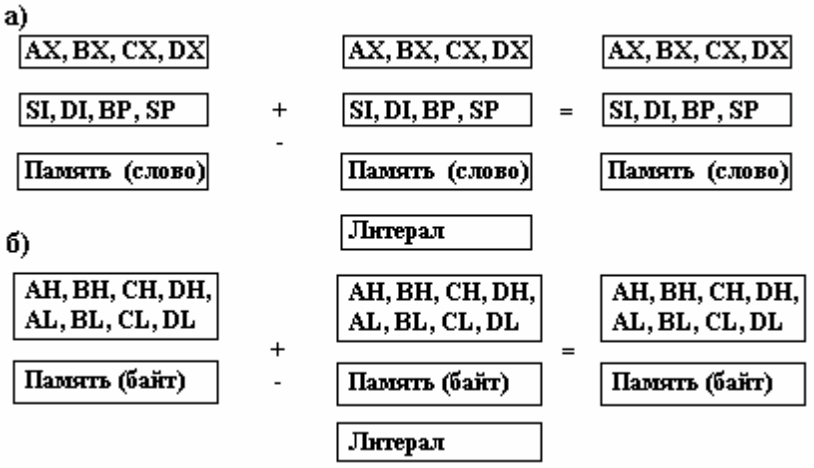


Рис. 7.

Приведем пример сложения двух 32-разрядных чисел:

```
mov   ax,value1
add   value2,ax
mov   ax,value1+2
adc   value2+2,ax
```

Исходные числа находится в основной памяти по адресам **value1** и **value2**, а результат записывается по адресу **value1**.

3.4.8. Команды вычитания

SUB <уменьшаемое-результат> , <вычитаемое>

SBB <уменьшаемое-результат>, <вычитаемое>

устанавливают флаги четности, знака результата, наличия заема, наличия переполнения.

При выполнении операции по команде **SUB** заем не учитывается, а по команде **SBB** - учитывается. Ограничения на местоположение операндов такие же, как и у команды сложения.

3.4.9. Команда изменения знака

NEG <операнд>

знак операнда изменяется на противоположный.

3.4.10.Команда добавления единицы

INC <операнд>

значение операнда увеличивается на единицу.

3.4.11.Команда вычитания единицы

DEC <операнд>

значение операнда уменьшается на единицу.

3.4.12.Команда сравнения

CMP <операнд 1> , < операнд 2>

выполняется операция вычитания без записи результата и устанавливаются признаки во флажковом регистре.

3.4.13.Команды умножения

MUL <операнд>

IMUL <операнд>

устанавливают флаги наличия переноса или переполнения.

По команде **MUL** числа перемножаются без учета, и по команде - **IMUL** с учетом знака (в дополнительном коде).

На рис. 8 (где а - операнды - слова, б - операнды - байты) приведены возможные способы размещения сомножителей и результата (один из сомножителей всегда расположен в регистре-аккумуляторе.

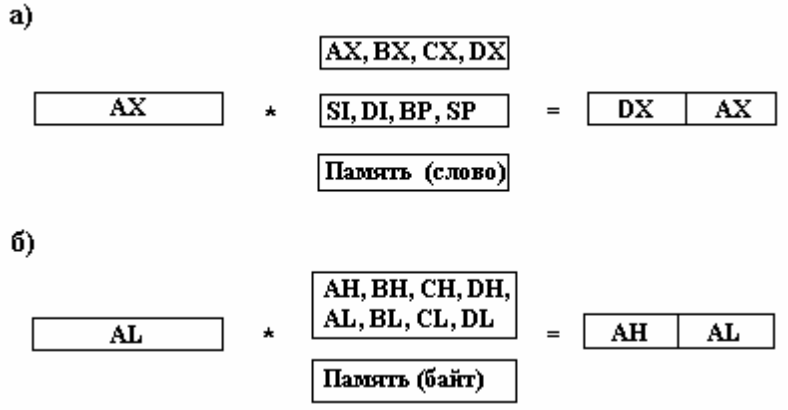


Рис. 8.

Рассмотрим пример:

imul word ptr c

Здесь содержимое основной памяти по адресу "**c**" длиной слово умножается на содержимое регистра **ax**. Младшая часть результата операции записывается в регистр **ax**, а старшая часть - и регистр **dx**.

3.4.14.Команда деления

DIV <операнд-делитель>
IDIV <операнд-делитель>

По команде **DIV** операция деления выполняется без учета, а по команде **IDIV** - с учетом знака (в дополнительном коде).

На рис. 9 приведены возможные способы размещения делимого, делителя и результата (а - операнды - слова, б - операнды - байты).

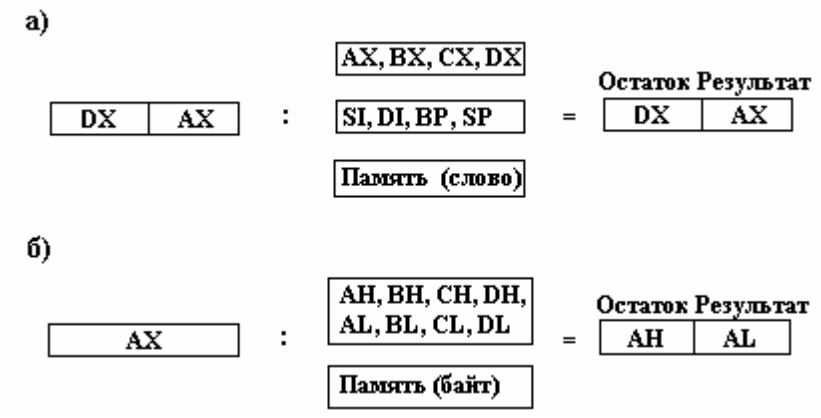


Рис. 9.

3.4.15.Команда преобразования байта в слово, а слова - в двойное слово

CBW
CWD

По команде **CBW** число из **al** переписывается в **ax** (дополнение выполняется знаковыми разрядами). Аналогично по команде **CWD** число из **ax** переписывается в два регистра **dx** и **ax**.

3.5. Команды передачи управления

3.5.1. Команда безусловного перехода

JMP <адрес перехода>

имеет три модификации в зависимости от длины ее адресной части:

short - при переходе по адресу, который находится на расстоянии -128...127 байт относительно адреса данной команды (длина адресной части 1 байт);

near ptr - при переходе по адресу, который находится на расстоянии 32 Кбайта (-32768...32767 байт) относительно адреса данной команды (длина адресной части 2байта);

far ptr - при переходе по адресу, который находится на расстоянии превышающем 32 Кбайта (длина адресной части 4 байта).

При указании перехода к командам, предшествующим команде перехода, ассемблер сам определяет расстояние до метки перехода и строит адрес нужной длины. При указании перехода к последующим частям программы необходимо ставить указатели **short**, **near ptr** и **far ptr**.

В качестве адреса команды перехода используются метки трех видов:

а) < имя > : **nop** (**nop** - команда "нет операции");

б)< имя> **label near** (для внутрисегментных переходов);

в) <имя> **label far** (для внесегментных переходов).

Примеры:

а) **jmp short b** - переход по адресу **b**;

б) **jmp [bx]** - переход по адресу в регистре **bx** (адрес определяется косвенно);

в) **a : nop -** описание метки перехода "**a"**;

г) **b label near** - описание метки перехода "**b**".

3.5.2. Команды условного перехода

<мнемоническая команда> <адрес перехода>

Мнемоника команд условного перехода:

JZ - переход по "ноль";

JE - переход по "равно";

JNZ - переход по "не нуль";

JNE - переход по "не равно";

JL - переход по "меньше";

JNG, JLE - переход по "меньше или равно ";

JG - переход по "больше";

JNL, JGE - переход по "больше или равно ";

JA - переход по "выше" (беззнаковое больше);

JNA, JBE - переход по "не выше"(беззнаковое не больше);

JB - переход по "ниже" (беззнаковое меньше);

JNB, JAE - переход по"не ниже" (беззнаковое не меньше).

Все команды имеют однобайтовое поле адреса, следовательно, смешение не должно превышать -128...127 байт. Если смещение выходит за указанные пределы, то используется специальный прием:

вместо	программируется
jz zero	**jnz continue**
	jmp zero
	continue: ...

3.5.3. Команды организации циклической обработки

В качестве счетчика цикла во всех командах циклической обработки используется содержимое регистра **cx**.

1) Команда организации цикла.

LOOP < адрес перехода >

при каждом выполнении уменьшает содержимое регистра **cx** на единицу и передает управление по указанному адресу, если **cx** не равно 0:

```
            mov    cx,  loop_count          ;  загрузка счетчика
begin_loop:
      ;  ...  тело цикла ...
             loop   begin_loop
```

Примечание. Если перед началом цикла в регистр **cx** загружен 0, то цикл выполняется 35536 раз.

2) Команда перехода по обнуленному счетчику.

JCXZ <адрес перехода>

передает управление по указанному адресу, если содержимое регистра **cx** равно 0. Например:

```
            mov    cx,  loop_count          ;  загрузка счетчика
            jcxz    end_of_loop             ;  проверка счетчика
begin_loop:
   ;  ...  тело цикла ...
             loop    begin_loop
end_of_loop:    ...
```

3) *Команды организации цикла с условием.*

LOOPE <адрес перехода>

LOOPNE <адрес перехода>

уменьшают содержимое на единицу и передают управление по указанному адресу при условии, что содержимое **cx** отлично от нуля, но **LOOPE** дополнительно требует наличия признака "равно", а **LOOPNE** - "не равно", формируемых командами сравнения. Например:

```
              mov    cx,  loop_count          ;  загрузка счетчика
              jcxz    end_of_loop             ;  проверка счетчика
begin_loop:
   ;  ...  тело цикла ...
             cmp     al, 100                 ; проверка содержимого al
             loopne  begin_loop    ; возврат в цикл, если cx≠0 и al≠100
end_of_loop:    ...
```

3.5.4. Команды вызова подпрограмм

1) Команда вызова процедуры.

CALL <адрес процедуры>

осуществляет передачу управления по указанному адресу, предварительно записав в стек адрес возврата.

При указании адреса процедуры так же как и при указании адреса перехода в командах безусловного перехода, возникает необходимость определить удаленности процедуры от места вызова:

а) если процедура удалена не более чем на -128...127 байт, то специальных указаний не требуется;

б) если процедура удалена в пределах 32 кбант, то перед адресом по процедуры необходимо указать **near ptr**,

в) если процедура подпрограмма удалена более, чем на 32 кбайта, то перед адресом процедуры необходимо записать **far ptr**.

Например:

call near ptr p - вызов подпрограммы "**p**".

Текст процедуры должен быть оформлен в виде:

< имя процедуры> **proc** < указатель удаленности>

... тело процедуры ...

<имя процедуры> **end**

Здесь указатель удаленности также служит дли определения длины адресов, используемых при обращении к процедуре: **near** - при использовании двухбайтовых адресов, **far** - при использовании четырехбайтовых адресов.

2) Команда возврата управления.

RET [<число>]

извлекает из стека адрес возврата и передает управление по указанному адресу.

Если в команде указано значение счетчика, то после восстановления адреса возврата указанное число добавляется к содержимому регистра-указателя стека. Последний вариант команды позволяет удалить из стека параметры, передаваемые в процедуру через стек.

3.6. Команды обработки строк

Команды обработки строк используются для организации циклической обработки последовательностей элементов длиной I или 2 байта. Адресация операндов при этом выполняется с помощью пар регистров: **DS:SI** - источник, **ES:DI** - приемник. Команды имеют встроенную корректировку адреса операндов согласно флагу направления **D**: 1 - уменьшение адреса на длину элемента, 0 - увеличение адреса на длину элемента. Корректировка выполняется после выполнения операции.

Установка требуемого значения флага направления выполняется специальными командами:

STD - установка флага направления в единицу,

CLD - сброс флага направления в ноль.

*1) Команда загрузки строки **LODS**.*

LODSB (загрузка байта),

LODSW (загрузка слова).

Команда загружает байт в **AL** или слово в **AX**. Для адресации операнда используются регистры **DS:SI**

*2) Команда записи строки **STOS**.*

STOSB (запись байта),

STOSW (запись слова)

записывает в основную память содержимое **AL** или **AX** соответственно. Для адресации операнда используются регистры **ES:DI**.

*3) Команда пересылки **MOVS**.*

MOVSB (пересылка байта),

MOVSW (пересылки слова)

пересылает элемент строки из области, адресуемой регистрами **DS:SI**, в область, адресуемую регистрами **ES:DI**.

*4) Команда сканирования строки **SCAS**.*

SCASB (поиск байта),

SCASW (поиск слова).

По команде содержимое регистра **AL** или **AX** сравниваются с элементом строки, адресуемым регистрами **DS:SI** и устанавливается значение флажков в соответствии с результатом **[DI] - AL** или **[DI]-AX**.

*5) Команда сравнения строк **CMPS**.*

CMPSB (сравнение байт),

CMPSW (сравнение слов)

элементы строк, адресуемых парами регистров **DS:SI** и **ES:DI**, сравниваются и устанавливаются значения флажков в соответствии с результатом **[DI]-[SI]**.

6) Префиксная команда повторения.

REP <команда>

позволяет организовать повторение указанной команды **CX** раз. Например:

rep stosb

Здесь поле, адресуемое парой регистров **ES:DI** длиной **CX** заполняется содержимым **AL** .

7) Префиксные команды "повторять, пока равно" и "повторять, пока не равно".

REPE < команда >

REPNE < команда ^

Префиксные команды используются совместно с командами **CMPS** и **SCAS**. Префикс **REPE** означает повторять, пока содержимое регистра **CX** не равно нулю и значение флажка нуля равно единице, а **REPNE** - повторять, пока содержимое регистра **CX** не равно нулю и значение флажка нуля равно нулю.

3.7. Команды манипулирования битами

3.7.1. Логические команды

NOT <операнд> - логическое НЕ;
AND <операнд 1>, <операнд 2> - лигическое И;
OR <операнд 1>, <операнд 2> - логическое ИЛИ;
XOR <операнд 1>, <операнд 2> - исключающее ИЛИ;
TEST <операнд 1>, <операнд 2> - И без записи результата.

Операнды байты или слова.

Пример. Выделить из числа в AL первый бит:

```
and  al, 10000000B
```

3.7.2. Команды сдвига

<код операции> <операнд>, <счетчик>

Счетчик записывается в регистр CL. Если счетчик равен 1, то его можно записать в команду.
Коды команд сдвига:

SAL - сдвиг влево арифметический;
SHL - сдвиг влево логический;
SAR - сдвиг вправо арифметический;
SHR - сдвиг вправо логический;
ROL - сдвиг влево циклический;
ROR - сдвиг вправо циклический;
RCL - сдвиг циклический влево с флагом переноса;
RCR - сдвиг циклический вправо с флагом переноса.

Пример. Умножить число в AX на 10:

```
mov  bx, ax
shl  ax, 1
shl  ax, 1
add  ax, bx
shl  ax, 1
```

3.8. Команды ввода - вывода

Обмен данными с внешней средой осуществляемся с помощью следующих команд:
IN <регистр>, <порт> (ввод из порта в регистр),
IN <регистр >, **DX** (ввод из порта, номер которого указан в регистре **DX** в регистр);
OUT <порт>, <регистр> (вывод содержимого регистра в порт),
OUT DX, <регистр> (вывод содержимого регистра в порт, номер которого указан в регистре **DX**).

В качестве регистра можно указать **AL** или **AX** (соответственно будет обрабатываться байт или два байта). Порт отождествляется с некоторым внешним устройством (0...255).

Однако при организации ввода - вывода помимо самой операции необходимо осуществить ряд дополнительных действий, например, проверить готовность устройства. В связи с этим для типовых устройств разработаны стандартные программы организации ввода - вывода, которые вызываются по команде прерывания **int 21h**.

В таблице 1 приведен перечень основные функции, реализуемые подпрограммами ввода - вывода, и их коды. Код функции должен передаваться в подпрограмму в регистре **AH**.

Таблица 1.

Код функции	Функция
01	Ввод с клавиатуры одного символа в регистр AL (с проверкой на Ctrl-Break, с ожиданием, с эхо)
02	Вывод одного символа на экран дисплея из регистра DL (с проверкой на Ctrl-Break)
06	Непосредственный ввод - вывод: ввод в регистр AL (без ожидания, без эхо, без проверки на Ctrl-Break, регистр DL должен содержать 0FFH), вывод из регистра DL (без проверки на Ctrl-Break).
07	Ввод в регистр AL (без проверки на Ctrl-Break, с ожиданием, без эхо)
08	Ввод в регистр AL (с проверкой на Ctrl-Break, с ожиданием, без эхо)
09	Вывод строки на экран (DS:DX - адрес строки, которая должна завершаться символом "$")
10(0Ah)	Ввод строки в буфер (DS:DX - адрес буфера, первый байт которого должен содержать размер буфера, после ввода - второй байт содержит количество введенных символов)
11(0Bh)	Чтение состояния клавиатуры (если буфер пуст, то AL=0, иначе AL=0FFh)

Примеры:

а)
```
mov   ah, 1   ; номер функции
int   21h     ;  ввод символа: символ в AL
```

б)
```
mov   ah, 2   ; номер функции
mov   dl, 'A'
int   21h     ;  вывод символа из DL
```

в)
```
lea   dx, STRING  ; адрес буфера ввода
mov   ah, 0Ah     ; номер функции
int   21h     ; ввод строки:  во втором байте буфера - количество
...           ; введенных символов, далее в буфере символы
STRING  db  50, 50 dup (?)
```

г)
```
lea   dx, MSG  ; адрес выводимой строки
mov   ah, 9    ; номер функции
int   21h      ; вывод строки
...
MSG   db   'Пример вывода', 13, 10, '$'
```

4. ПРОГРАММИРОВАНИЕ НА АССЕМБЛЕРЕ

4.1. Структура программы на ассемблере

Структура программы на языке ассемблера выглядит следующим образом (**.exe**):

```
                    TITLE  <имя программы>
<имя сегмента стека>   SEGMENT  STACK
                    DB   3000  DUB   (?)
<имя сегмента стека>   ENDS
<имя сегмента данных > SEGMENT
                    <данные>
<имя сегмента данных > ENDS
<имя сегмента кодов>   SEGMENT
                    ASSUME CS: <имя сегмента кодов>,  DS:<имя сегмента данных>
                    EXTRN  <имя внешней процедуры >:<тип>
                    PUBLIC <имя внутренней процедуры>
<имя основной процедуры> PROC FAR
                       PUSH  DS
                       MOV   AX, 0
                       PUSH  AX
                       MOV   AX, <имя сегмента данных>
                       MOV   DS, AX
<тело процедуры>       RET
<имя основной процедуры> ENDP
<имя внутренней процедуры> PROC    NEAR
                    <тело внутренней процедуры>
<имя внутренней процедуры> ENDP
<имя сегмента кодов>       ENDS
                       END     <имя основной процедуры >
```

Первая строка программы - заголовок, состоящий из служебного слова **TITLE** и имени программы.

Текст программы состоит из отдельных сегментов, каждый из которых начинается оператором **SEGMENT** и завершается оператором **ENDS**:

```
<имя сегмента > SEGMENT
        ... тело сегмента ...
<имя сегмента > ENDS
```

Сегмент стека содержит специальный описатель **STACK.**

Сегмент кодов, в котором располагается текст программы, начинается псевдокомандой **ASSUME**, которая сообщает ассемблеру, какой сегментный регистр должен использоваться для адресации каждого сегмента.

За псевдокомандой **ASSUME** может следовать описание используемых программой внешних подпрограмм:

```
EXTRN  <имя внешней процедуры >:<тип near или far>
PUBLIC <имя внутренней процедуры>
```

Сегмент кодов всегда адресуется сегментным регистром **CS.** Значение этого регистра операционная система устанавливает автоматически. Значения сегментного регистр **DS** загружается программистом:

```
MOV   AX, <имя сегмента данных>
MOV   DS, AX
```

При необходимости также загружается регистр **ES**:

```
MOV   ES, AX
```

Команды

```
PUSH  DS
MOV   AX, 0
PUSH  AX
```

организуют возможность возврата управления в MS DOS командой **RET.**

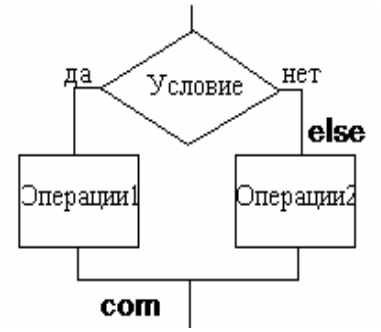
В этом случае в стек в качестве адреса возврата помещается адрес префиксной области программы, первые два байта которой содержат команду **INT 20H** возврата управления операционной системе.

4.2. *Основные приемы программирования на ассемблере*

Ассемблер, являясь языком низкого уровня, не содержит операторов ветвления, циклов, не поддерживает автоматического формирования адресов для структур данных, не обеспечивает автоматического выполнения преобразований при вводе-выводе данных. Все перечисленные операции программируются "вручную" с использованием имеющихся команд ассемблера.

4.2.1. Программирование ветвлений

Ветвления программируются с использованием команд условной и безусловной передачи управления.




```
      cmp    ...
      j<условие>     ELSE
      <операции 1>
      jmp    COM
ELSE: <операции 2>
COM:  <продолжение>
```

Пример.

Написать процедуру вычисления **X=max(A,B)**:

```
max     proc    near
        mov     ax, A
        cmp     ax, B      ; сравнение A и B
        jl      LESS       ; переход по меньше
        mov     X, ax
        jmp     CONTINUE   ;  переход на конец ветвления
LESS:   mov     ax,  B
        mov     X, ax
CONTINUE: ret
max     endp
```

4.2.2. Программирование циклических процессов

Программирование циклических процессов осуществляется с использованием либо команд переходов, либо - в случае счетных циклов - с использованием команд организации циклов.

а) программирование итерационных циклов (цикл-пока):

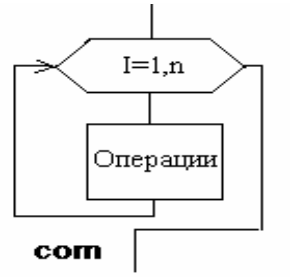

```
CYCL:    cmp  ...
         jne   COM
       <операции>
         jmp   CYCL
COM:   ....
```

Пример.

Написать процедуру суммирования чисел от 1 до 10, используя итерационый цикл.

```
sum      proc     near
         mov      ax, 0     ;  обнуление суммы
         mov      bx, 1     ;  первое слагаемое
CYCL:    cmp      bx, 10    ;  слагаемое больше 10
         jg       CONTINUE  ; выход из цикла
         add      ax, bx    ;  суммирование
         inc      bx        ;  следующее число
         jmp      CYCL      ;  возврат в цикл
CONTINUE:  ret     ; выход,   сумма - в ax
sum      endp
```

а) программирование счетных циклов:




```
         mov   cx, N
CYCL:  <операции>
         loop  CYCL
com:     .....
```

Пример.

Написать процедуру суммирования чисел от 1 до 10, используя счетный цикл.

```
sum      proc     near
         mov      ax, 0     ;   обнуление суммы
         mov      bx, 1     ;   первое слагаемое
         mov      cx, 10    ;   загрузка счетчика
CYCL:    add      ax, bx    ;   суммирование
         inc      bx        ;   следующее число
         loop     CYCL      ;    возврат в цикл
continue:  ret     ; выход,   сумма - в ax
sum      endp
```

4.2.3. Моделирование одномерных массивов

Массив во внутреннем представлении - это последовательность элементов в памяти, например:

```
A       dw    10,13,28,67,0,-1    ; массив из 6 чисел длиной слово.
```

Программирование обработки выполняется с использованием адресного регистра, в котором хранится либо адрес текущего элемента, либо его смещение относительно начала массива. При переходе к следующему элементу адрес (или смещение) увеличивается на длину элемента.

Пример.

Написать процедуру, выполняющую суммирование массива из 10 чисел размером слово.

Вариант 1 (используется адрес):

```
summas    proc
          mov     ax, 0
          lea     bx, MAS
          mov     cx, 10
CYCL:     add     ax, [bx]
          add     bx, 2
          loop    CYCL
          ret
summas    endp
```

Вариант 2 (используется смещение):

```
summas    proc
          mov     ax, 0
          mov     bx, 0
          mov     cx, 10
CYCL:     add     ax, MAS [bx]
          add     bx, 2
          loop    CYCL
          ret
summas    endp
```

Второй вариант позволяет получать более наглядный код и потому является предпочтительным.

В том случае, если элементы просматриваются непоследовательно, адрес элемента может расчитываться по его номеру: $A_{исп} = A_{начала}$+(<номер>-1)*<длина элемента>. Полученный по формуле адрес записывается в один из адресных регистров (**BX, BP, DI, SI**) и используется для доступа к элементу.

Пример.

Написать процедуру, которая извлекает из массива, включающего 10 чисел размером слово, число с номем n (n≤10).

```
n_mas     proc
          mov   bx, N     ; номер числа
          dec   bx        ; вычитаем 1
          sal   bx, 1     ; умножили на длину (сдвинули влево на 1)
          mov   ax, MAS [bx]  ; результат в ax
          ret
n_mas     endp
```

4.2.4. Моделирование матриц

Значения матрицы могут располагаться в памяти по строкам и по столбцам. Для определенности будем считать, что матрица расположена в памяти построчно.

При моделировании обработки матрицы следует различать просмотр по строкам, просмотр по столбцам, просмотр по диагоналям и произвольный доступ.

Просмотр по строкам иногда может выполняться так, как в одномерном массиве (без учета перехода от одной строки к другой).

Пример.

Написать процедуру определения максимального элемента матрицы A(3,5).

```
maxmatr   proc
          mov     bx, 0     ; смещение 0
          mov     cx, 14    ; счетчик цикла
          mov     ax, A     ; заносим первое число
CYCL:     cmp     ax, A[bx+2]  ; сравниваем числа
          jge     NEXT      ; если больше, то перейти к следующему
          mov     ax, A[bx+2]  ; если меньше, то запомнить
NEXT:     add     bx, 2     ; переходим к следующему числу
          loop    CYCL
          ret     ; результат в ax
maxmatr   endp
```

Просмотр по строкам при необходимости фиксировать завершение строки и просмотр по стобцам выполняются в двойном цикле: по строкам - во внешнем цикле, по столбцам - во внутреннем или наоборот. В этом случае обычно отдельно формируются смещения строки и столбца.

Пример.

Определить сумму максимальных элементов столбцов матрицы A(3,5).

```
maxmatr     proc
            mov     ax, 0       ; обнуляем сумму
            mov     bx,  0      ; смещение элемента столбца в строке
            mov     cx,  5      ;  количество столбцов
CYCL1:      push    cx          ;  сохраняем счетчик
            mov     cx, 2       ;  счетчик элементов в столбце
            mov     dx, A[bx]   ; заносим первый элемент столбца
            mov     si, 10      ; смещение второго элемента столбца
CYCL2:      cmp     dx,  A[bx]+ [si]  ; сравниваем
            jge     NEXT        ; если больше или равно - к следующему
            mov     dx,  A[bx]+[si]  ;  если меньше, то сохранили
NEXT:       add     si,  10     ;  переходим к следующему элементу
            loop    CYCL2       ; цикл по элементам столбца
            add     ax,  dx     ;  просуммировали макс. элемент
            pop     cx          ;  восстановили счетчик
            add     bx,  2      ;  перешли к следующему столбцу
            loop    CYCL1       ;  цикл по столбцам
            ret                 ;  результат в ax
maxmatr     endp
```

4.2.5. Преобразования ввода-вывода

При программировании операций ввода-вывода на ассемблере приходится вручную осуществлять преобразования чисел из символьного представления во внутренний формат (двоичный с фиксированной точкой, отрицательные числа записаны в дополнительном коде) и обратно.

Для облегчения преобразования во внутренний формат целесообразно оговорить возможные варианты ввода чисел в символьном виде. При этом также используется то, что при добавлении цифры к числу справа число меняется следующим образом: <число>:=<число>*10+<цифра>.

Обратное преобразование из внутреннего формата в символьный для вывода результатов обычно использует стандартное правило перевода числа из двоичной системы счисления в десятичную: деление на 10 с выделением остатков. В этом случае десятичные цифры получаются в обратном порядке.

Если среди вводимых или выводимых чисел могут быть отртцательные, то необходимо предусмотреть специальную проверку и преобразовывать отрицательные числа в дополнительный код при вводе и в прямой при выводе.

Пример.

Написать процедуры ввода массива из **n** чисел размером слово и вывода того же массива. Числа должны вводиться каждое со своей строки, положительные числа должны вводиться без знака с первой позиции, отрицательные - со знаком в первой позиции. Вывод всех чисел должен осуществляться в одну строку через пробелы. Перед отрицательными числами необходимо выводить знак "-".

Составить тестирующую программу.

```
            title inout
code        segment
            assume cs:code, ds:code
N           equ     5     ;  определяем константу для транслятора
A           dw      N dup (?)  ;  резервируем место под массив
main            proc    far      ;  основная процедура
                push    ds       ;  обеспечиваем возврат управления в MS DOS
                mov     ax,  0
                push    ax
                mov     ax, code  ;  загружаем сегментный адрес
                mov     ds, ax    ;        сегмента данных в DS
                call    input     ;  вызываем процедуру ввода
```

```
                call    output     ;  вызываем  процедуру вывода
                ret                ;  возврат управления DOS
main            endp               ;  конец основной процедуры
input           proc   near        ;  процедура ввода
                mov    cx, N       ; загругка размерности массива
                mov    di,  0      ; загрузка смещения массива
CYCL1_IN:  push   cx        ; сохраняем счетчик внешнего цикла
                lea    dx,  MES_IN  ; загружаем адрес запроса на ввод
                mov    ah,  9       ; загружаем номер функции DOS
                int    21h          ; вызываем функцию вывода строки
                lea    dx,  BUF_IN  ; загружаем  адрес строки ввода
                mov    ah,  0ah  ; загружаем номер функции DOS
                int    21h          ; вызываем функцию ввода строки
                mov    byte ptr NEG_IN,  0  ; признак - "число положительно"
                cld          ;      флаг направления - "возрастание адресов"
                mov    cl,   BUF_IN+1   ; загружаем длину введенной строки
                mov    ch,  0    ; счетчик  - в CX
                lea    si,  BUF_IN+2   ; загружаем адрес введенной строки
                cmp    byte ptr [si],  '-' ;   число отрицательно ?
                jne    short UNSIGNED   ; если нет, то переход
                mov    byte ptr NEG_IN, 1  ; признак - "число отрицательно"
                inc    si     ; пропускаем знак
                dec    cx     ; уменьшаем счетчик
UNSIGNED:  mov    bx, 0  ; исходное значение числа
CYCL2_IN:    mov    ax, 10  ; заносим константу 10
                mul    bx      ;  умношаем текущее значение числа на 10
                mov    bx,  ax  ;  текущее значение числа в BX
                lodsb    ;  загрузили очередную цифру
                sub    al,  30h  ;  преобразовали из символа в двоичное число
                cbw              ; преобразовали в слово
                add    bx,  ax   ; добавили к текущему значению числа
                loop   CYCL2_IN   ;  выполняем для всех введенных цифр
                cmp    NEG_IN,  0  ;  число положительно ?
                je     short  DONE    ;  если да, то переход
                neg    bx   ;  преобразуем в отричательное число
DONE:           mov    A[di],  bx   ; записываем результат в массив
                add    di,  2  ; корректируем смещение
                pop    cx   ;  восстанавливаем счетчик внешнего цикла
                loop   CYCL1_IN  ; выполняем для всех чисел
                ret    ;  возвращаем управление основной процедуре
BUF_IN      db      10,10 dup (0)  ;  буфер ввода (до 10 символов)
NEG_IN      db      0   ;  признак "положительно/отрицательно"
MES_IN      db      13,10,'Введите число: $'  ;  запрос на ввод
input       endp
output      proc    near   ; процедура вывода
            lea     dx, MES_OUT ; загружаем адрес заголовка вывода
            mov     ah,  9     ; загружаем номер функции DOS
            int     21h       ; вызываем функцию вывода строки
            mov     cx,  N   ;  загружаем количество чисел
            mov     si,   0   ;  устанавливаем смещение в массиве
            mov     di,  7   ;  устанавливаем смещение для строки вывода
cycl1_out:  mov     byte ptr NEGOUT, 0 ; признак - "число положительно"
            mov     bx,  10 ; загрузили константу 10
            mov     ax, A[si]  ; взяли очередное число из массива
            cmp     ax,  0   ;  сравнили с нулем
            jge     short   AGAIN  ; если положительно, то переход
            mov     byte ptr NEG_OUT, 1 ; признак "число отрицательно"
```

```
            neg    ax  ; преобразовали в положительное
AGAIN:      cwd     ; преобразовали в двойное слово
            div    bx   ;  разделили на 10
            add    dl,  30h   ; преобразовали остаток в символ
            mov    BUF_OUT[di],  dl ; записали в поле вывода
            dec    di   ; уменьшили смещение в поле вывода на 1
            cmp    ax,  0  ; сравнили результат деления с нулем
            jne    AGAIN   ; если не равно, то получаем следующую цифру
            cmp    byte ptr NEG_OUT,  1 ; число отрицательно
            jne    short POSITIVE   ;  если нет, то переход
            mov    BUF_OUT[di],  '-'  ; если да, то вставили знак "-"
POSITIVE:   mov    ax,  N+2  ; считаем смещение следующего числа
            sub    ax,   cx            ;  в поле вывода
            mov    bx,  7              ;
            mul    bx                  ;
            mov    di,  ax             ;   записали смещение в DI
            add    si,   2          ;  перешли к следующему числу массива
            loop   CYCL1_OUT    ;  выполняем для всех чисел массива
            lea    dx,  BUF_OUT    ; загружаем адрес строки вывода
            mov    ah,  9      ; загружаем номер функции DOS
            int    21h       ; вызываем функцию вывода строки
            ret     ; возврат в основную процедуру
NEG_OUT     db  0  ;  признак "положительно/отрицательно"
BUF_OUT     db    13,10, N*7 dup (' '),'$'  ; поле вывода
MES_OUT     db    13,10,'Результат: $'   ; заголовок вывода
output      endp
code        ends
            end    main
```

5. ЛИТЕРАТУРА

1. Лю Ю-Чжен, Гибсон Г. Микропроцессоры семейства 8086/8088. Архитектура, программирование и проектирование микрокомпьютерных систем.: Пер. с англ. - М.: Радио и связь, 1987. - 512 с.
2. Скэнлон Л. Персональные ЭВМ IBM PC и XT. Программирование на языке ассемблера: Пер. с англ. - М.: Радио и связь. 1989. - 336 с.
3. Джордейн Р. Справочник программиста персональных компьютеров ЭВМ IBM PC, XT и AT: Пер. с англ. - М.: Финансы и статистика, 1992. - 544 с.
4. Финогенов К.Г. Самоучитель по системным функциям MS DOS. - М.: Радио и связь, Энтроп, 1995. - 382 с.