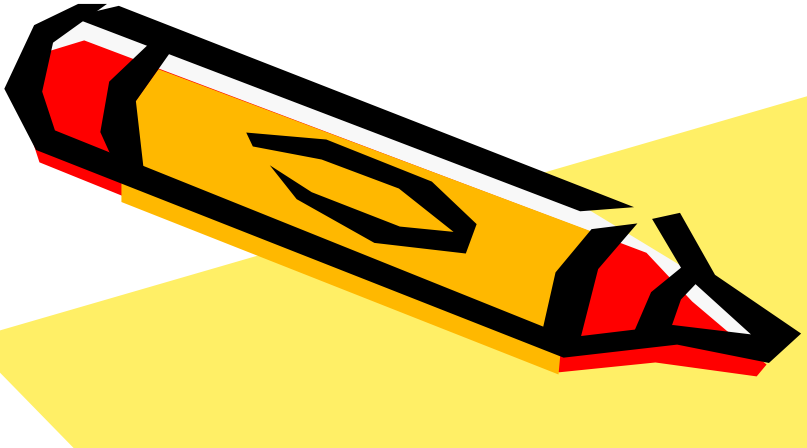
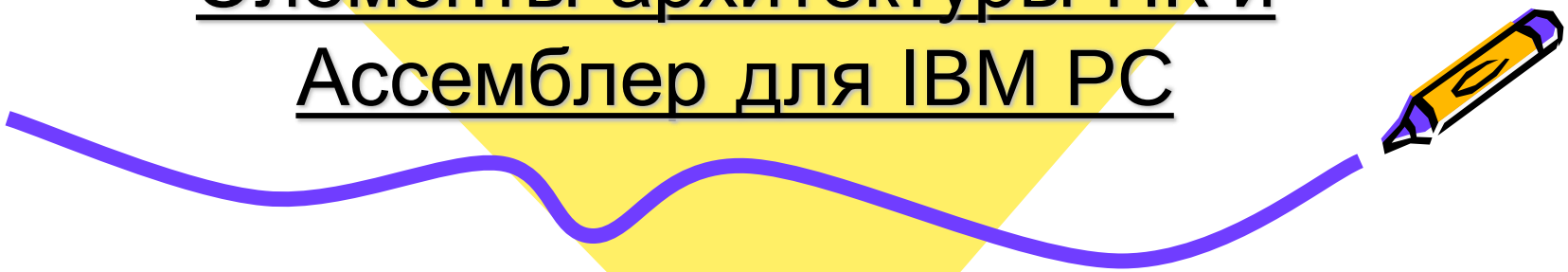


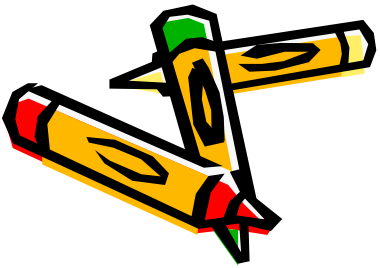
Системное программирование

Элементы архитектуры ПК и Ассемблер для IBM PC



Литература

1. В.Н. Пильщиков «программирование на языке Ассемблера»
2. В.И. Пустоваров «Язык Ассемблера в информационных и управляющих систем программирования»
3. О.В. Бурдаев, М.А. Иванов, И.И. Тетерин «Ассемблер в задачах защиты информации»
4. С.В. Зубков «Ассемблер – язык неограниченных возможностей. Программирование под DOS, Windows, Unix»
5. А. Жуков, А. Авдюхин «Ассемблер. Самоучитель»
6. С.К. Фельдман «Системное программирование»



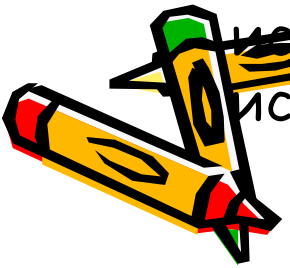
Работа с подпрограммами в Ассемблере

Программа, оформленная как процедура, к которой обращение происходит из ОС, заканчивается командой возврата ret.

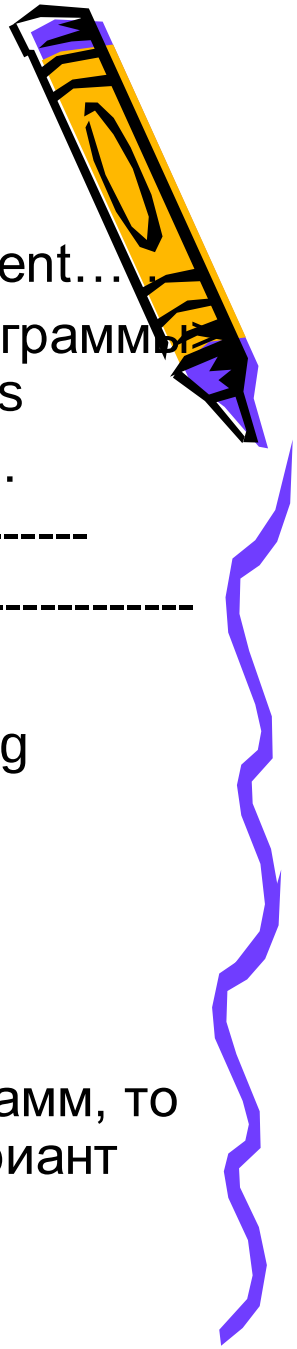
Подпрограмма, как вспомогательный алгоритм, к которому возможно многократное обращение помощью команды call, тоже оформляется как процедура с помощью директив proc и endp. Структуру процедуры можно оформить так:

```
<имя процедуры> proc   <параметр>
                       <тело процедуры>
                ret
<имя процедуры> endp
```

В Ассемблере один тип подпрограмм – процедура. Размещать ее можно в любом месте программы, но так, чтобы управление на нее не попадало случайно, а только по команде call. Поэтому описание ПП принято располагать в конце программного сегмента (после последней исполняемой команды), или вначале его – перед первой исполняемой команды.



Графическое представление



1) cseg segment ……
beg: ----------

---
fin: ----------
< подпрограмма 1>
<подпрограмма 2>

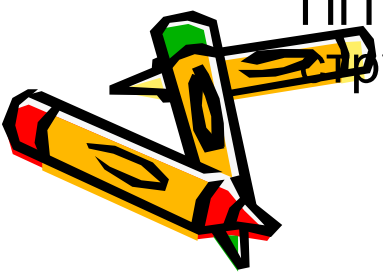
<подпрогрмма N>
cseg ends
end beg

2) cseg segment
< подпрограмма 1>
<подпрограмма 2>

<подпрогрмма N>
beg: ----------
---
cseg ends
end beg

3)cseg_pp segment…
<подпрограммы>
cseg _pp ends
cseg segment…
---
beg: ----------
cseg ends
end beg

Если программа содержит большое количество подпрограмм, то ПП размещают в отдельном кодовом сегменте – вариант структуры 3).



Замечания:

1) После имени в директивах proc и endp двоеточие не ставится, но имя считается меткой, адресом первой исполняемой команды процедуры.

2) Метки, описанные в ПП, не локализуются в ней, поэтому они должны быть уникальными в рамках всей программы.

3) Параметр в директиве начала процедуры один – FAR или NEAR

Основная проблема при работе с ПП в Ассемблере – это передача параметров и возврат результатов в вызывающую программу.

Существуют различные способы передачи параметров: 1) по значению, 2) по ссылке, 3) по возвращаемому значению, 4) по результату, 5) отложенным вычислением.

Параметры можно передавать: 1) через регистры, 2) в глобальных переменных, 3) через стек, 4) в потоке кода, 5) в блоке параметров.

Передача параметров через регистры – наиболее простой способ. Вызывающая программа записывает в некоторые регистры фактические параметры…….

Примерами использования этого метода являются вызовы некоторых прерываний ОС и BIOS.

Когда регистров не хватает, один из способов обойти это ограничение – записать параметр в глобальную переменную, к которой затем обращаться в ПП. Но этот метод считается не эффективным, так как может оказаться невозможной рекурсия, и даже простое повторное обращение к ПП.

Передача параметров через стек. Перед обращением к процедуре фактические параметры (их значения или адреса) записываются в стек, а процедура их из стека извлекает. Именно этот способ используют языки высокого уровня.

Передача параметров в потоке кода заключается в том, что данные, передаваемые в ПП, располагаются сразу за командой обращения к ПП **call**. ПП, чтобы использовать эти данные, должна обратиться к ним по адресу, который записывается в стек автоматически как адрес возврата из ПП. Но ПП в этом случае должна перед командой возврата изменить адрес возврата на адрес байта, следующий за передаваемыми параметрами. Этот метод реализует передачу параметров медленнее, чем через регистры, глобальные переменные или стек, но примерно также, как и метод передачи параметров в блоке параметров.

Блок параметров – это участок памяти, содержащий параметры и располагающийся обычно в сегменте данных.

Процедура получает адрес начала этого блока при помощи любого из рассмотренных методов: в регистре, в переменной, в стеке, в коде или даже в другом блоке параметров. Примеры использования этого способа – многие функции ОС и BIOS, например, поиск файла, использующий блок параметров DTA, или загрузка и исполнение программы, использующая блок параметров EPB.

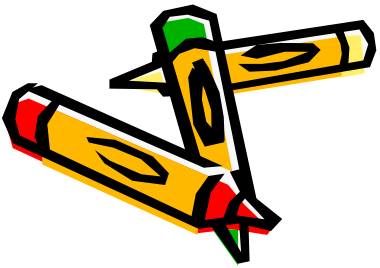
Передача параметров по значению и по ссылке.

При передаче параметров по значению процедуре передается значение фактического параметра, оно копируется в ПП, и ПП использует копию, поэтому изменение, модификация параметра оказывается невозможным. Этот механизм используется для передачи параметров небольшого размера.

Например, нужно вычислить **c = max (a, b) + max (7, a-1)**. Здесь все числа знаковые, размером в слово. Используем передачу параметров через регистры. Процедура получает параметры через регистры AX и BX, результат возвращает в регистре AX.

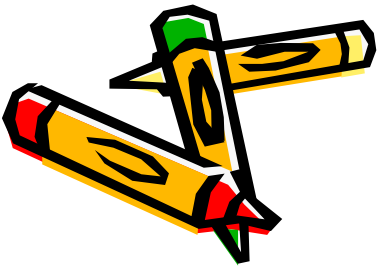
Процедура: AX = max (AX, BX)

```
max     proc
        cmp AX, BX
        jge  met1
        mov AX, BX
met1:   ret
max     endp
```



Фрагмент вызывающей программы:

```
        ---------------------------------
; c = max (a,b) + max (7, a-1)
        mov AX, a
        mov BX, b
        call max      ; AX = max (a,b)
        mov c, AX  ; c = max (a,b)
        mov AX, 7
        mov BX, a
        Dec BX
        call max      ; AX = max (7, a-1)
        add c, AX
        -------------------------------
```

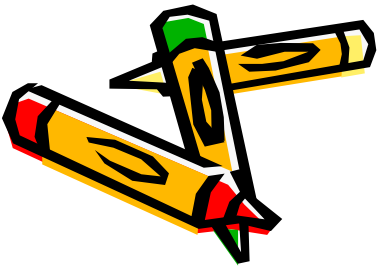


Передача параметров по ссылке.

Оформим как процедуру вычисление **x = x div 16**

Процедура имеет один параметр-переменную x, которой в теле процедуры присваивается новое значение. Т.е. результат записывается в некоторую ячейку памяти. И чтобы обратиться к процедуре с различными параметрами, например, a и b, ей нужно передавать адреса памяти, где хранятся значения переменных a и b. Передавать адреса можно любым способом, в том числе и через регистры. Можно использовать различные регистры, но чаще используются BX, BP, SI, DI. Пусть адрес параметра передается через регистр BX, тогда фрагмент программы:

```
; основная программа
--------------------------
lea BX, a
call  Proc_dv
lea BX,  b
call  Proc_dv
------------------------------
```



Процедура:

```
Proc_dv    proc
           push CX
           mov  CL, 4
           shr  word ptr [BX],  CL ; x = x div 16
           pop CX
           ret
Proc_dv    endp
```

Сдвиг на 4 разряда вправо эквивалентен делению нацело на 16 и выполняется быстрее.

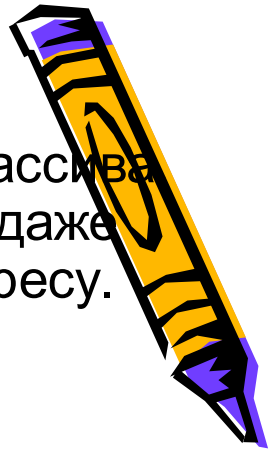
Здесь первая команда в процедуре сохраняет в стеке значение регистра CX, так как затем использует CL в команде сдвига и возможно этот регистр используется в основной программе.

Т.к. регистров немного а и ПП и основная программа могут использовать одни и те же регистры, то при входе в ПП нужно сохранять в стеке значения регистров, которые будут использоваться в ПП, а перед выходом из нее восстанавливать значения этих регистров. Для поддержки этого, начиная с ix186, в систему команд введены команды сохранения в стеке и извлечения из него сразу всех регистров общего назначения

pusha и **popa**, а, начиная с ix386, **pushad** **popad**.

Не нужно сохранять в стеке значение регистра, в который записывается результат работы ПП.

Передача параметров по ссылке в блоке параметров



Если параметров много, например, массив, адрес начала массива как блока параметров, можно передать через регистр, даже если результат ПП не будет записываться по этому адресу.

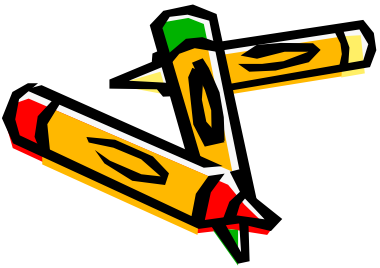
Даны два массива целых положительных чисел без знака

X DB 100 dup (?)

Y DB 50 dup (?)

Вычислить DL = max (X[i]) + max (Y[i]), использовав процедуру max (A[i]), пересылая адрес массива через регистр BX, а результат сохраняя в AL. ----------; фрагмент программы

```
lea BX,  X
mov CX,  100
call  max          ;  AL = max (X[i])
mov DL,  AL        ;  DL = max (X[i])
lea  BX,  Y
mov CX,  50
call max           ; AL = max (Y[i])
ADD DL,  AL
------------------------------
```



Процедура max: AL = max (A[0..n-1]), BX – начальный адрес A, CX = n

```
max     proc
        push CX
        push BX
        mov AL, 0 ; начальное значение max
met1:   cmp [BX], AL
        jle met2
        mov AL, [BX]
met2:   inc BX
        loop met1
        pop BX
        pop CX
        ret
max     endp
```

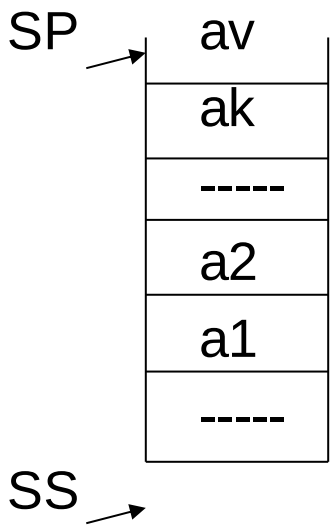
Передача параметров через стек.

Этот способ передачи параметров называют универсальным, его можно использовать при любом количестве параметров, хотя он сложнее, чем передача параметров через регистры. Но для передачи результатов чаще используют регистры.

Если ПП имеет k параметров PP(a_1, a_2, ….a_k) размером в слово и параметры сохраняются в стеке в последовательности слева направо, то команды, реализующие обращение к ПП, должны быть следующими:

```
; обращение к процедуре PP
        push a1
        push a2
        ------------
        push ak
        call  PP
av:     ------------
```

содержимое стека при входе в PP



Обращение к параметрам в процедуре можно осуществить с помощью регистра BP, присвоив ему значение SP.

Но при этом мы испортим старое значение BP, которое может быть используется в основной программе. Поэтому следует вначале сохранить старое значение BP в стеке, а затем использовать его для доступа к параметрам, т.е тело процедуры должно начинаться следующими командами:

```
PP      proc near
        push BP
        mov BP, SP
        ----------
```

в стеке после вып-ия этих команд

SP, ~~BP~~ →	BPст
+ 2	av
+ 4	ak
	-------
	a1

Для доступа к последнему параметру можно использовать выражение [BP + 4], например, mov AX, [BP + 4] ; ak → AX

После «входных действий» в ПП идут команды, реализующие вспомогательный алгоритм, а за ними д.б. команды, реализующие «выходные действия»:

```
        pop BP   ; восстановить старое значение BP
        ret 2*k  ; очистка стека от k параметров
PP      endp     ; возврат в вызывающую программу
```

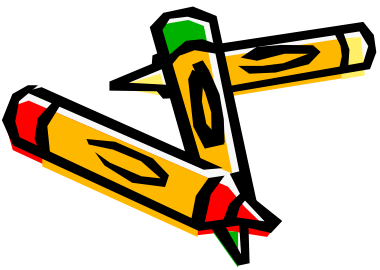
n в команде **ret n** – это количество освобождаемых байтов в стеке,

поэтому количество параметров д. б. умножено на длину параметра…..

Команда ret вначале считывает значение av, а затем удаляет из стека параметры. Очистку стека можно выполнять не в ПП, а после выхода из нее, в основной программе, сразу после команды **call PP**, например, командой **add SP, 2*k**

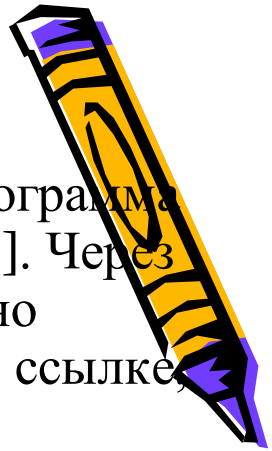
Каждый способ имеет свои достоинства и недостатки, если в ПП, то исполняемый код будет короче, если в основной программе, то можно вызвать ПП несколько раз с одними и теме же параметрами последовательными командами **call….**

Для удобства использования параметров, переданных через стек, внутри ПП можно использовать директиву equ, чтобы при каждом обращении к параметрам не писать точное смещение относительно BP.



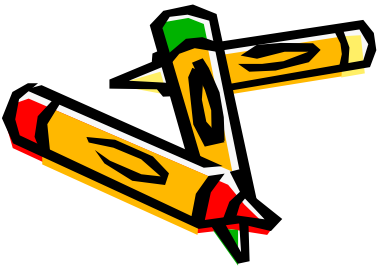
```
    push x
    push y
    push z
    call PP
    ----------


PP   proc near              ; процедура
       push BP
       mov BP,  SP
pp_x equ  [BP + 8]
pp_y equ  [BP + 6]
pp_z equ  [BP + 4]
--------------------
    mov AX, pp_x        ; использование параметра x
----------------------
                pop  BP
                ret  6
       pp       endp
```

Пример передачи параметров через стек.



Пусть процедура заполняет нулями массив A[0..n-1] , основная программа обращается к ней для обнуления массивов X[0..99] и Y[0..49]. Через стек в ПП передается имя массива и его размер, размер можно передавать по значению, а имя массива нужно передавать по ссылке, т.к. этот параметр является и входным и выходным.

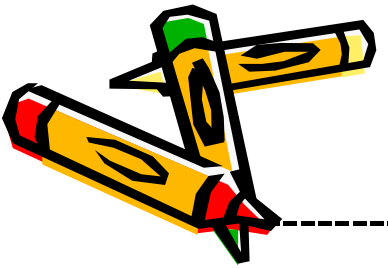
```
; процедура zero_1
zero_1      proc
            push BP             ; входные
            mov BP,  SP         ; действия
            push BX; сохранение значений
            push CX; регистров
            mov CX,  [BP + 4]           ; CX = n   считывание из стека
            mov BX,  [BP + 6]           ; BX = A           параметров
m1:         mov byte ptr [BX],  0       ; цикл обнуления
            inc BX                      ; массива
            loop m1                     ; A[0..n-1]
            ;
```



```
; восстановление регистров и выходные действия
        pop CX
        pop BX
        pop BP
        Ret 4
zero_1  endp
```

Фрагмент основной программы:

```
    X DB 100 dup (?)
    Y DB 50  dup (?)
    ----------------------
            lea AX, X       ; загрузка параметров:
            push AX         ; адреса массива X
            mov AX,  100    ; и его размера
            push AX         ; в стек
            call zero_1     ; обращение к ПП
            lea AX,  Y      ; загрузка параметров для массива Y
            puah AX         ;
            mov AX,  50     ;
            push AX;
            call  zero_1    ; обращение к ПП
    ----------------------
```



О передаче параметров в ПП

1. Передача по значению:

   mov AX, word ptr value

   call PP

2. Передача по ссылке:

   mov AX, offset value

   call PP

3. Передача параметров по возвращаемому значению объединяет передачу по значению и по ссылке: процедуре передается адрес переменной, она делает локальную копию этого параметра, работает с этой копией, а в конце процедуры записывает эту копию по переданному адресу. Этот механизм оказывается эффективным, если процедуре приходится много раз обращаться к параметру в глобальной переменной.

4. Передача параметров по результату заключается в том, что ПП передается адрес только для записи по этому адресу результата работы ПП.

5. Передача параметров по имени макроопределения. Пример:

```
name  macro parametr
        mov AX,  parametr
name    endm
```

Обращение к ПП может быть таким:

```
name value        ; обращение к макро
call  PP          ; обращение к ПП
```

6. Передача параметров отложенным вычислением. Как и в случае передачи параметров по имени, процедура получает адрес ПП, вычисляющей значение параметра. Этот механизм чаще используется в системах искусственного интеллекта и в ОС.

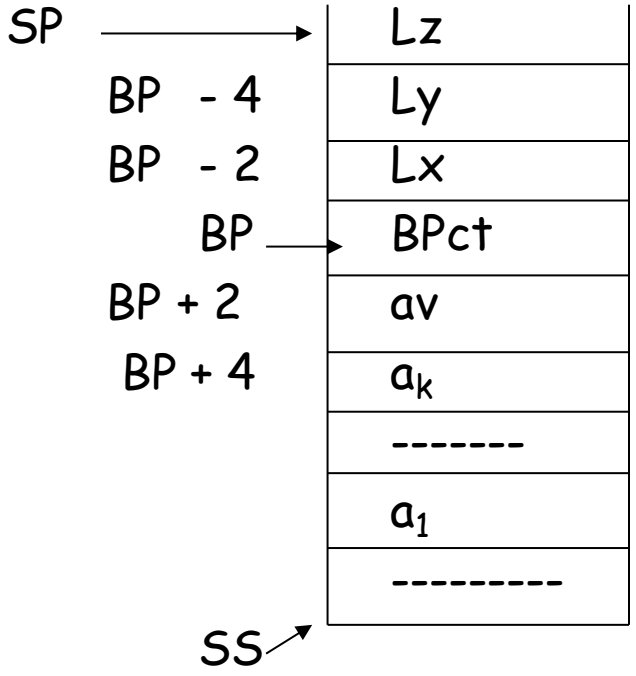
Использование локальных параметров.

Если локальных параметров немного, то их размещают в регистрах, но если их много, то возможны различные варианты: им можно отвести место в сегменте данных, но тогда большую часть времени эта область памяти не будет использоваться.

Использование локальных параметров.

Лучший способ – разместить локальные параметры в стеке на время работы ПП, а перед выходом из ПП их удалить. Для этого после входных действий в процедуре нужно уменьшить значение указателя на вершину стека SP на количество байтов, необходимых для хранения локальных величин и затем записывать их в стек и извлекать их можно с помощью выражений вида: [BP – n], где n определяет смещение локального параметра относительно значения BP.

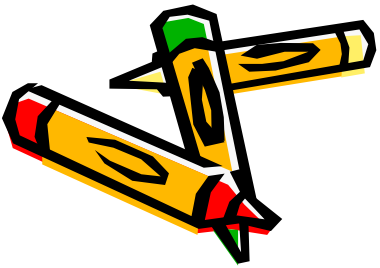
Например, если предполагается, что ПП будет использовать 3 локальные параметра размером в слово, то стек графически можно представить так:



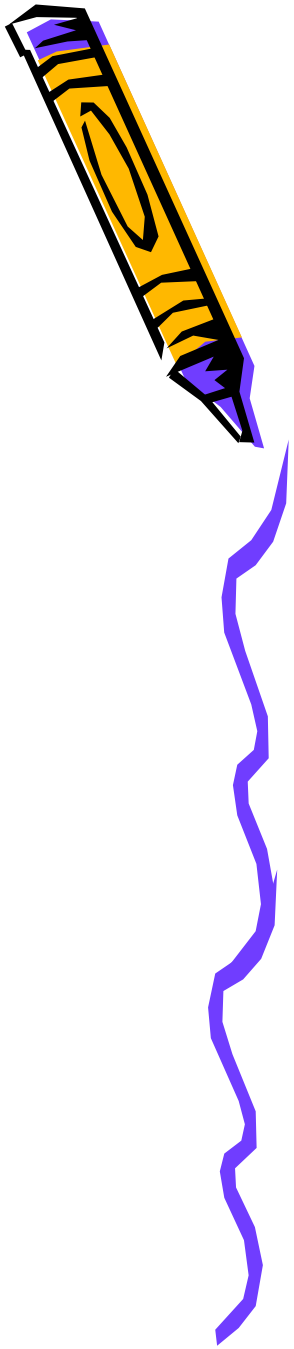
При выходе из процедуры перед выполнением завершающих действий нужно возвратить регистру SP его значение.

Если в стеке хранятся и фактические и локальные параметры, то начало процедуры и ее завершение должно выглядеть следующим образом:

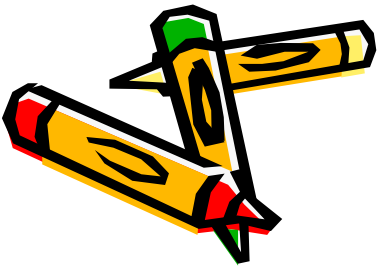
```
PP          proc
            push BP          ; сохранить старое значение BP
            mov BP,  SP      ; (SP) ——→ в BP
            sub SP,  k1       ; отвести в стеке k1 байтов под лок пар - ы
            push AX          ; сохранить в стеке регистры,
            ----------       ; используемые в ПП
            <тело процедуры>
             ----------      ; восстановить регистры
             pop AX          ;
            mov SP,  BP      ; восстановить SP, т.е. освободить место
                             ;     в стеке от локальных параметров
            pop BP           ; восстановить BP, равным до обращения к ПП
            ret k2           ; очистка стека от фактических
                    ; параметров и возврат в вызывающую программу
PP          endp     ; конец ПП
```



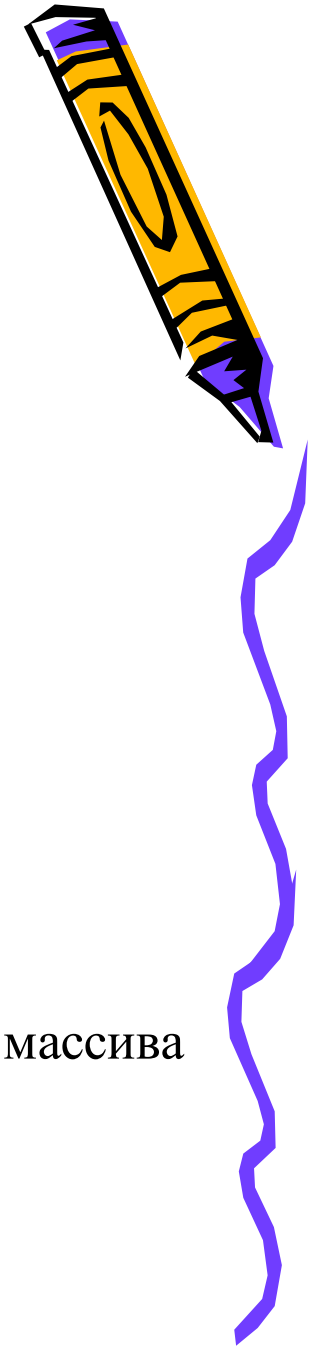
Подсчет количества различных символов в заданной строке.



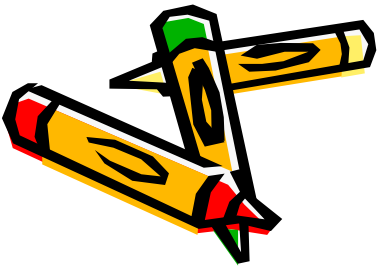
Строка задана как массив символов. Начальный адрес ее передадим в ПП через регистр BX, длину строки через CX, а результат - через AX. Создадим процедуру, в которой выделяется 256 – ый локальный массив L по количеству возможных символов. K-ому элементу этого массива будем присваивать единицу, если символ, цифровой код которого равен K, в заданной строке существует. Затем подсчитаем количество единиц в этом массиве. Вначале весь массив обнуляется. К первому элементу этого массива можно обратиться так: $\mathbf{L_1 = [BP - 256]}$ к K – ому $\mathbf{L_k = [BP - 256 + k]}$ Работая со строками, эту задачу можно решить проще.



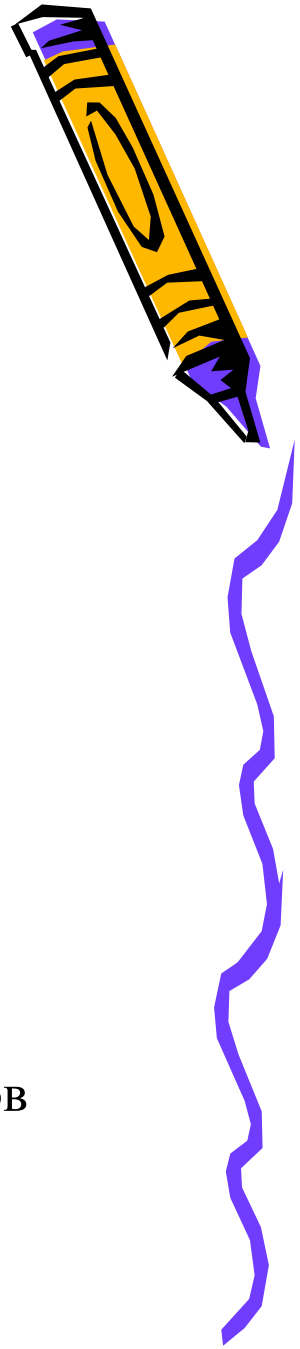
```
Count_s      proc ;
   ; входные действия
              push BP
              mov BP,  SP
              sub SP,  256
              push BX
              push CX
              push SI
```



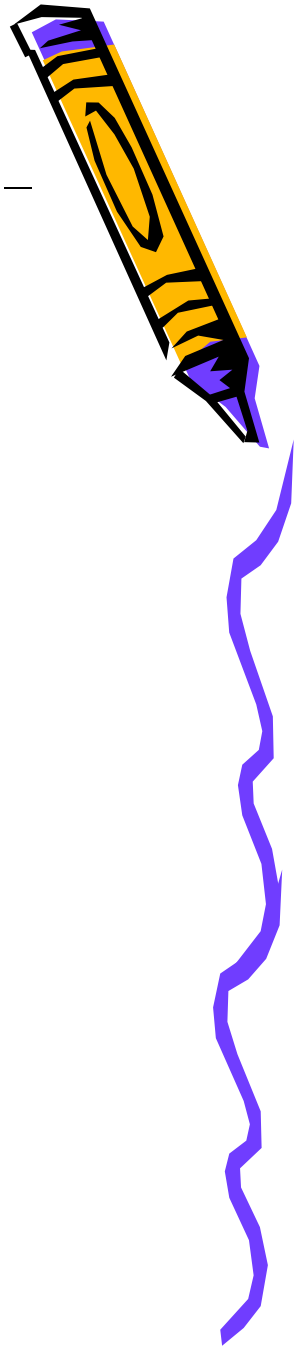
```
; Обнуление локального массива

        mov AX,  CX  ; сохранение длины исходной строки
        mov  CX,  256 ; возможное количество символов
        mov SI,  0   ; индекс элемента массива
m1:     mov byte ptr [BP – 256 + SI],  0 ;
        inc SI
        loop m1
; просмотр заданной строки и запись 1 в локальный массив
        mov CX,  AX  ; длину строки в CX
        mov AX,  0
m2:     mov AL,  [BX]  ; код очередного символа в AL
        mov SI,  AX      ;  пересылаем его в SI
        mov byte ptr [BP – 256 + SI],  1  ; пересылаем 1 в к –й элемент массива
        inc BX
                loop m2  ;
```



```
; подсчет количества 1 в локальном массиве
        mov AX,  0  ; результат будет в AX
        mov CX,  256 ; количество повторений цикла
        mov SI,  0  ;  индекс массива в SI
m3:     cmp  byte ptr [BP – 256 + SI], 1
        jne  m4
        inc AX
m4:     inc SI
        loop m3
; выходные действия
            pop SI    ;  восстановление
            pop CX  ;  регистров
            pop BX   ;
            mov SP,  BP ; освобождение стека от локальных параметров
            pop BP                ;  восстановить старое BP
            ret
            const_s    endp
```



Рекурсия в Ассемблере



Основные трудности, возникающие при реализации рекурсии – это опасность «зацикливания » рекурсии и использование параметров. Зацикливания не произойдет, если в процедуре есть рекурсивная и не рекурсивная ветви и при выполнении некоторого условия вычислительный процесс пойдет по не рекурсивной ветви. Рекурсивное обращение ПП можно представить, если предположить, что при каждом обращении создается копия ПП и адреса возврата сохраняются в стеке. А структура стека позволяет извлекать их в последовательности, обратной поступлению.

Также решается и проблема с параметрами. В рекурсивную процедуру нельзя передавать параметры через ячейки памяти в сегменте данных, а если такая необходимость возникает, то при входе в ПП их необходимо сохранять в стеке, а при выходе из нее восстанавливать. Это значит, что лучше сразу параметры передавать через стек.

Пример рекурсивной функции

F(n) = 1, если n = 0 или n = 1 и
F(n) = F(n – 1) + F(n – 2), если n >1

Вычисление n-го ряда Фибоначчи

```
Fib proc      ; BX = F(n),  AL = n
      cmp AL,  1
      ja   m1     ; если n > 1,      m1
; не рекурсивная ветвь
      mov BX,  1  ; если n < 1 или n = 1      BX = F(n) = 1
      ret
; рекурсивная ветвь
m1:  push AX  ;
      dec AL            ;  AL = n - 1
      call Fib          ;  BX = F(n-1)
      push BX           ; сохранить в стеке F(n-1)
      dec  AL           ; AL = n - 1
      call   Fib        ;  BX = F(n – 2)
      pop AX            ;  AX = F(n – 1)
      add BX,  AX       ;  BX = F(n – 2) + F(n – 1)
      pop AX            ;  восстановить  AX
            ret
            Fib       endp
```

Работа со строками

Строка – это последовательность байтов, слов или двойных слов.

Все команды для работы со строками считают, что строка-источник находится по адресу **DS:SI (DS:ESI)**, а строка приемник – по адресу **ES:DI (ES:EDI)**.

Все команды работают с одним элементом строки: одним байтом, одним словом или одним двойным словом в зависимости от команды и /или от типа операндов.

Чтобы выполнить действие над всей строкой, слева от команды записывается специальный префикс.

Префикс – это команда повторения операции. Префикс действует только на команды работы со строками, поставленный рядом с любой другой командой он никак не влияет на ее выполнение.

Существуют следующие префиксы:

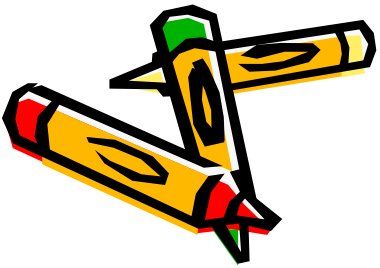
rep – повторять

repe – повторять пока равно

repz – повторять пока ноль

repne – повторять пока не равно

repnz – повторять пока не ноль



1) Префикс повторять **rep <строковая команда>** заставляет повторяться указанную команду n раз, где n - содержимое регистра **CX (ECX)**. Если **(CX) = 0**, то команда не выполнится ни разу.

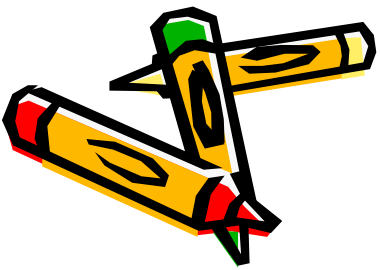
2) **repe <стр. команда> = repz <стр. команда>**

Указанная строковая команда будет повторяться до тех пор пока флаг ZF = 1, но не более n раз, где n = (CX) или (ECX). Работу команды с этими префиксами можно на псевдокоде описать так:

```
m:  if (CX) = 0 then goto m1;
        (CX) = (CX) – 1;
        < стр команда>;
        if ZF = 1 then goto m
m1: ----------------------
```

3) repne <стр команда> = repnz <стр команда>

Указанная строковая команда повторяется до тех пор, пока флаг ZF = 0, но не более n раз, где n – содержимое счетчика CX (ECX).



семантика этих префиксов на псевдокоде:

m: if (CX) = 0 then goto m1;
(CX) = (CX) – 1;
<стр команда>;
if ZF = 0 then goto m;
m1: ----------

Префикс **rep** используется обычно с командами:

movs, lods, stos, ins и outs

Префиксы **repe, repz, repne, repnz** с командами **cmps и scas**.

Команды копирования для строк.

1) **movs op1, op2** ; источник op2 = DS:SI (DS:ESI), приемник op1 = ES:DI (ES:EDI)
2) **movsb** ; байт данных из (DS:SI) пересылается в ES:DI
3) **movsw** ; слово данных из (DS:SI) пересылается в ES:DI
4) **movsd** ; дв. слово данных из (DS:SI) пересылается в ES:DI

При использовании команды 1) - movs Ассемблер сам определяет по типу указанных в команде операндов сколько байтов данных нужно переслать – 1, 2 или 4.

В этой команде можно изменить DS на другой регистр: ES, GS, FS, CS, SS, но регистр операнда приемника ES изменять нельзя.

Чаще команды для строк используются без операндов.

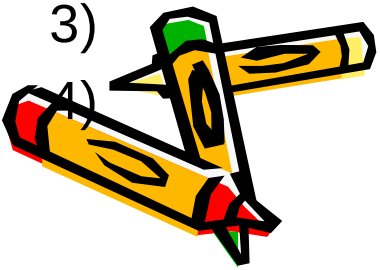
После выполнения любой команды со строками содержимое регистров SI и DI автоматически изменяется в зависимости от значения флажка DF.

Если DF = 0 , то (SI/ESI) и (DI/EDI) увеличивается на 1 или 2 или 4,

Если DF = 1 , то (SI/ESI) и (DI/EDI) уменьшается на 1 или 2 или 4 в зависимости от операндов или кода команды.

Команды сравнения строк.

1) cmps op1, op2 ;
2) cmpsb ; сравнение байтов
3) cmpsw ; сравнение слов
4) cmpsd ; для i386 и > сравнение двойных слов



По команде 1) в зависимости от типа операндов сравнивается содержимое байтов, слов или дв. слов, расположенных по адресам источника и приемника.

В остальном команды сравнения работают также, как и команды пересылки. Эти команды используются с префиксами

1) **repe / repz** и 2) **repne / repnz**

При использовании префиксов 1) сравнение идет до первого не совпадения, 2) – до первого совпадения.

Команды: 1) **scas op1** ; op1 - приемник

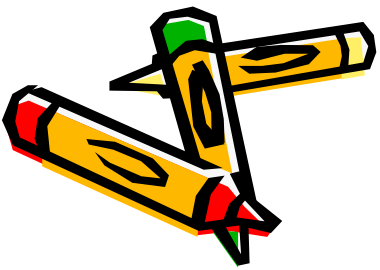
2) **scasb** ; сравнивает (AL) с байтом из ES:DI / ES:EDI

3) **scasw** ; сравнивает (AX) со словом из ES:DI / ES:EDI

4) **scasd** ; для i386 и выше,

; сравнивает (EAX) с двойным словом из ES:DI / ES:EDI

При работе команды 1) количество сравниваемых байтов зависит от разрядности операнда.



Команды **cmps и scas** устанавливают флаги аналогично команде **cmp**.

Команды считывания строки из памяти и загрузки в регистр AL, AX или EAX

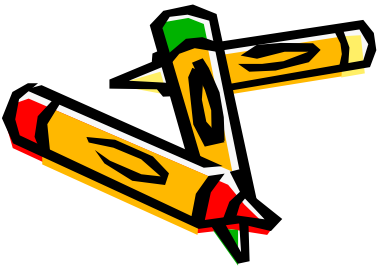
1) lods op2 ; op2 – источник DS:SI или DS:EDI
2) lodsb ; 1 байт из DS:SI или DS:EDI ⟶ AL
3) lodsw ; 2 байта из DS:SI или DS:EDI ⟶ AX
4) lodsd ; 4 байта из DS:SI или DS:EDI ⟶ EAX

lods op2 работает как lodsb или lodsw или lodsd в зависимости от типа операнда и здесь DS можно заменить на ES, FS, GS, CS SS.

Команды записи строки из регистра AL, AX или EAX в память по адресу ES:DI или ES:EDI.

1) stos op1
2) stosb
3) stosw
4) stosd ; для i386 и выше

При использовании этих команд с префиксом rep строка длиной в (CX) (ECX) ячеек заполнится числом, хранящимся в аккумуляторе AL, AX или EAX



Считывание из порта ввода/вывода.

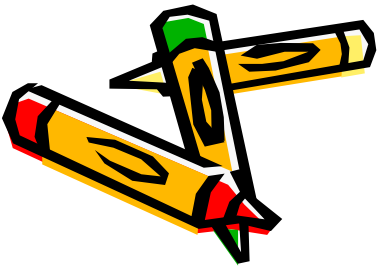
1) ins op1, DX
2) insb
3) insw
4) insd ; i386 и >

Эти команды считывают из порта ввода/вывода, номер которого содержится в регистре DX, байт (insb), слово (insw) или двойное слово (insd) и пересылает их в память по адресу ES:DI или ES:EDI. Команда 1) принимает одну из форм 2), 3), 4) в зависимости от типа операнда.

При использовании с префиксом rep она считывает из порта ввода/вывода блок данных (байтов, слов или двойных слов) длинной, определяемой регистром CX (ECX) и пересылает в память по адресу приемника.

Запись в порт в/в содержимого ячейки памяти, размером в байт, слово или дв. слово, находящейся по адресу DS:SI или DS:EDI

1) outs DX, op2
2) outsb
3) outsw
4) outsd ; i386 и >



Номер порта в командах работы с портами в/в должен находиться в регистре DX

В команде outs можно заменить DS на ES, FS, GS, CS SS. Используя префикс rep можно переслать в порт блок данных размером в (CX) или (ECX) байтов, слов или двойных слов.

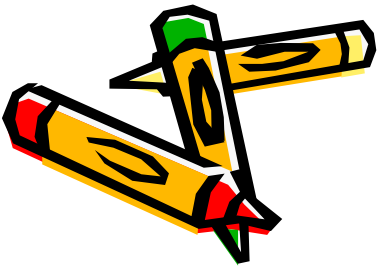
Команды управления флагами.

После выполнения команд со строками изменяется содержимое регистров – индексов в зависимости от значения флажка направления DF. Автоматически его значение не изменяется, его должен изменить программист с помощью команд:

```
cld     ; CLear Df,     DF = 0
std     ; SeT Df,       DF = 1
```

Программист может установить следующие флажки:

```
stc     ; CF = 1
clc     ; CF = 0
cmc     ; инвертировать флаг переноса
lahf    ; копирует младший байт регистра FLAGS a AH
sahf    ; из AH загружает флажки SF, ZF, AF, PF, CF
```



```
cli      ; IF = 0
sti      ; IF = 1
salc     ; установить регистр AL в соответствии с CF
```

Загрузка сегментных регистров

```
lds op1,  op2
les op1,  op2
lfs op1,  op2
lgs op1,  op2
lss op1,  op2
```

Для всех команд op2 – переменная в ОП размером в 32 или 48 бит в зависимости от разрядности операндов. Первые 16 бит этой переменной загружаются в соответствующий сегмент DS, ES и т.д., а следующие 16 или 32 - в регистр общего назначения, указанный в качестве первого операнда. В защищенном режиме значение, загружаемое в сегментный регистр, всегда должно быть правильным селектором сегмента, в реальном режиме любое число может использоваться как селектор.

Загрузка сегментных регистров

```
S1      DB      “ABC$”
ADR     DD      S1
        les     DI, ADR
```

В переменную ADR записывается полный адрес, определяемый именем S1 (Seg:Ofs). В ES записывается значение сегментной части адреса S1, а в DI ее смещение.

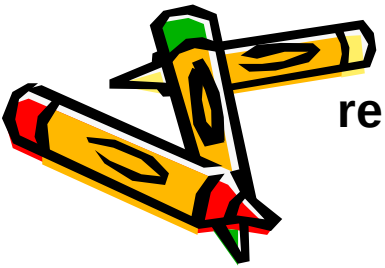
Пример 1 использования команд работы со строками:

```
X  DW  100 dup (?)
Y  DW  100 dup (?)
```

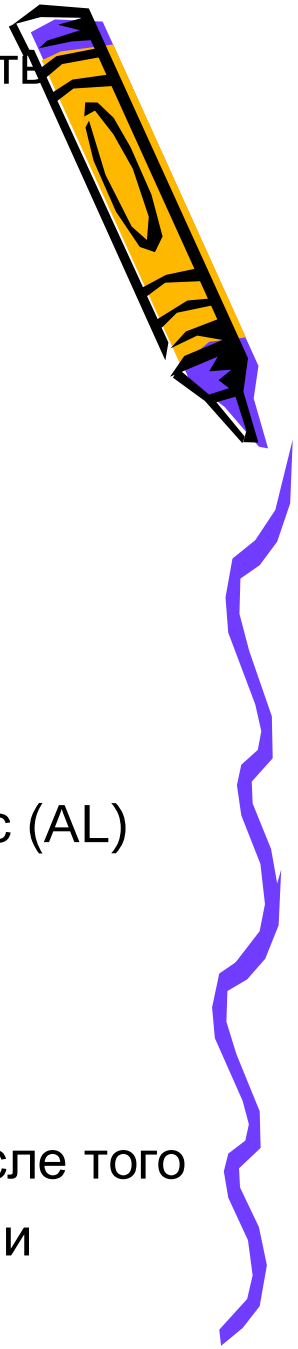
Выполнить пересылку содержимого одной области памяти в другую

X = Y:

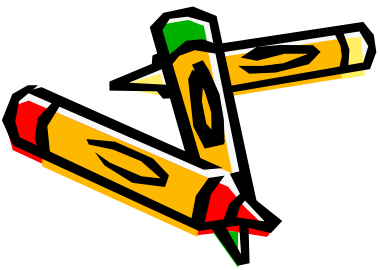
```
        ----------------------------------
        CLD       ; DF = 0
        lea SI,  Y  ;  DS:SI – начало Y
        push DS
        pop ES    ; (ES) = (DS)
        lea  DI,  X
        mov CX,  100
rep     movsw
        ---------------------------------
```



Пример 2. В строке S, состоящей из 500 символов заменить первое вхождение звездочки на точку.



```
        ------------------------
    CLD      ; просмотр строки слева направо
    push DS
    pop ES
    lea DI,  S
    mov CX,  500
    mov AL,  ‘ * ‘
    repne scasb      ; сканирование строки S  и сравнение с (AL)
    jne finish       ; ‘ * ‘ в строке нет
    mov byte ptr ES:[DI – 1],  ‘ . ‘
finish: ------------------------
```



Здесь используется выражение [DI – 1] т.к. после того как звездочка найдена DI увеличивается на 1 и указывает на следующий символ.

Строки переменной длины

Строка в языке Ассемблера может быть реализована по аналогии с тем, как это сделано в языке C/C++ и как в языке Паскаль. В C/C++ за последним символом строки располагают специальный символ, являющийся признаком конца строки. Изменение длины строки сопровождается переносом этого символа. Недостатком такого представления строк переменной длины является то, что, например, для сравнения строк S1 и S2, длиной 500 и 1000 символов необходимо выполнить может быть 500 сравнений, хотя зная, что длина их различна, их можно было совсем не сравнивать. В Паскале строка представляется так:

S

n	s1	s2	……..	Sn	…

Где n – текущая длина. Сколько места необходимо отводить под значение длины строки n – зависит от максимально возможной длины. Если она может состоять не более, чем из 255 символов, то под n достаточно одного байта. Тогда текущая длина строки содержится по адресу S, а ее i-ый символ по адресу S + i. Строку из 200 символов можно описать так:

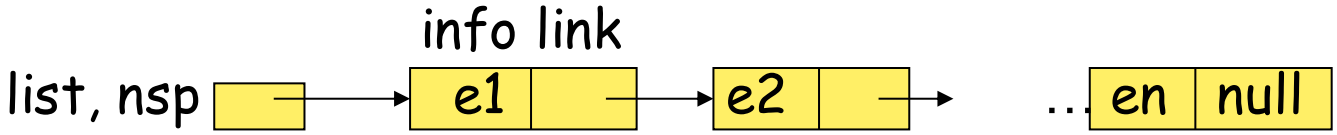
```
S   DB  201 dup (?)
```

Пример 3. Удалить из строки S первое вхождение символа звездочка.

```
---------------------------------------------
        ; поиск ‘ * ‘
        push DS   ;
        pop ES    ;  (ES) = (DS)
        lea DI,  S + 1;  ES:DI = адресу S[1]
        CLD     ; просмотр вперед
        mov CL,  S  ; текущая длина строки
        mov CH,  0  ; в CX
        mov AL,  ‘ * ‘
repne  scasb  ; поиск ‘ * ‘  в S
        jne finish  ; ‘ * ‘ в S нет ⟶ на метку finish
; удаление ‘ * ‘ из S, сдвинуть S на 1 символ Si = Si+1
        mov SI,  DI     ;  DS:SI = адресу, откуда начинать пересылку
        dec DI          ;  ES:DI = куда пересылать
rep movsb               ; сдвиг «хвоста» S на 1 позицию влево
        dec S           ; уменьшить на 1 текущую длину
finish:----------------------
```

Представление и работа со списками в Ассемблере

Односвязный линейный связный представляют в виде:



Стандартных процедур для работы со списками в языке Ассемблера нет, их нужно реализовывать самим. Динамические переменные, располагаются в специальной области ОП, называемой кучей (heap). Размер кучи зависит от количества использованных в программе динамических переменных, будем считать от количества и длины списков.

Предположим, что для кучи достаточно 64 Кб, тогда пусть начало кучи определяет сегментный регистр ES. Если внутри кучи элемент списка имеет адрес (смещение) A, то абсолютный физический адрес этого элемента определяется адресной парой ES:A. Так как в этих адресных парах для всех элементов общим является начало кучи ES, то будем считать адресом элемента списка 16-разрядное смещение A.

Под каждый элемент списка отводится фиксированное количество байтов пусть информационное поле определяется именем elem и занимает 2 байта, тогда элемент списка можно описать как структуру:

```
node struc          ; тип элемента списка
    elem  DW  ?     ; информационное поле
    next  DW  ?     ; ссылочное поле
node  ends
```

Если A описана с помощью директивы node: **A node < >**, то доступ к полям элемента списка с адресом A осуществляется так:

```
ES: A.elem ;
ES: A.next  ;
```

Пустая ссылка – это адрес 0, определив константу **NULL EQU 0**, пользуемся для обозначения пустой ссылки константой NULL, как и в C++.

Ссылки на первые элементы списков описывают обычно в сегменте данных DS как переменные, размером в слово:

```
nsp DW  ? ;
list  DW ?  ;
```

При работе со списками просматриваются элементы один за другим, так что необходимо знать адрес текущего элемента. Используем для хранения этого адреса регистр **BX**, причем в **BX** будет храниться только смещение текущего элемента, адрес отсчитанный от начала кучи, поэтому, чтобы обратиться к элементу списка, необходимо использовать выражение ES: [BX], если укажем просто [BX] , то по умолчанию этот адрес будет выбираться из сегмента DS.

Обращение к полям текущего элемента это:

ES:[BX].elem и ES:[BX].next

Основные операции.

1) **Анализ информационного поля:**

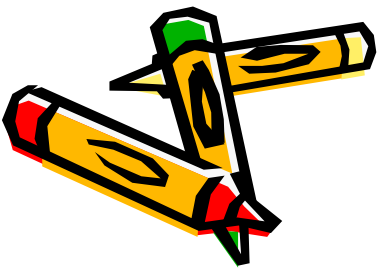
```
--------------------
mov AX, ES:[BX].elem  ; сравнение информационного поля
cmp AX, X             ; со значением X
je  jes               ; если совпали, то переход на jes
--------------------
```

2) **Переход к следующему элементу:**

```
mov BX, ES:[BX].next
```

3) **Проверка на конец списка:**

```
cmp BX, null
je  list_end      ;
```



4) Поиск элемента с заданным значением информационного поля

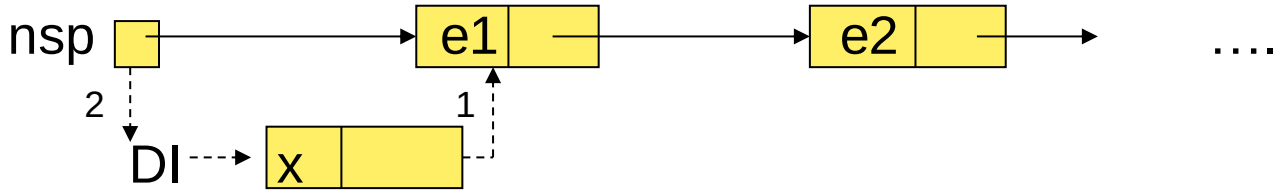
nsp – начало списка, x - искомая величина, в AL – результат = 1, если такой элемент в списке есть, или 0, если такого элемента нет.

```
        ---------------------------------
        mov AL,  0
        mov CX,  x
        mov BX,  nsp  ; BX = null, или адрес первого элемента
L:      cmp BX,  null
        je       no       ;  если BX = null, то на метку no
        cmp ES:[BX].elem, CX
        je       jes      ; [BX].elem = x, то на метку jes
        mov BX,  ES:[BX].next  ; BX = BX.next
        jmp      L ; на повторение цикла пока не конец списка
jes:    mov AL,  1
no:     -------------------------
```

5) Вставка нового элемента в начало списка

В Ассемблере нужно самим написать процедуру new, которая выделяет место в куче для размещения нового элемента. Пусть такая процедура с именем new есть, она без параметров и результатом ее является адрес байта в куче, начиная с которого можно разместить новый элемент списка. Этот адрес передается вызывающей процедуре через регистр DI. Тогда вставить элемент в начало списка

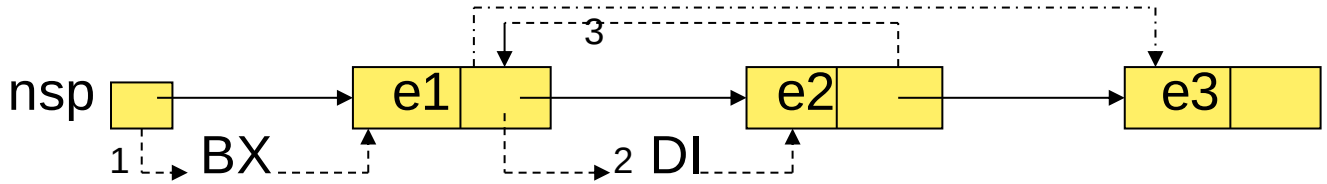
```
-------------------------------
1.      call new  ;
2.      mov  AX,  x
3.      mov  ES:[DI].elem,  AX
4.      mov  AX,  nsp
5.      mov  ES:[DI].next,  AX     ;
6.      mov  nsp,  DI
--------------------------------
```



6) Удаление элемента из списка.

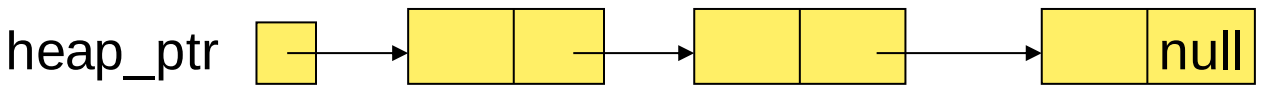
Пусть для адреса 1-го элемента используем BX, адреса 2-го элемента - DI и есть уже процедура dispose (DI), удаляющая элемент из списка, т.е. освобождающая место в куче для дальнейшего использования, тогда удаление второго элемента можно реализовать так:

```
1)  mov  BX,  nsp  ; адрес первого элемента в BX
    cmp  BX,  null  ; if nsp = null, список пуст
    je   finish
2)  mov  DI,  ES:[BX].next  ; DI = null, или адресу 2-го элемента
    cmp  DI,  null          ; если DI = null, 2-го элемента нет
    je   finish
3)  mov  AX,  ES:[DI].next
3)  mov  ES:[BX].next,  AX
    call  dispose           ; освобождение памяти
finish:  ------------------------
```



Организация «кучи» и процедур создания и удаления динамических переменных

При выполнении программы, занятые и свободные ячейки в куче, располагаются не последовательно, а произвольно, так как различные элементы списка могут удаляться и также произвольно создаваться. Чтобы определить какие же ячейки кучи свободны, удобнее всего все свободные ячейки кучи объединить в один список. Его называют списком свободной памяти (ССП). Адрес начала списка хранится в фиксированной ячейке с именем heap_ptr. Если программе необходимо место под очередной элемент в некотором списке, это место выделяется из ССП, если удаляется некоторый элемент, то он добавляется к ССП. Т.е. ССП можно представить как обычный список, для простоты с элементами такой же структуры.

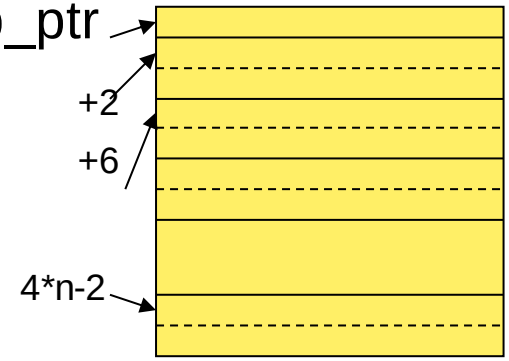


Остается определить где располагается переменная heap_ptr – указатель на ССП. Лучше всего отвести ей место в самом начале кучи, в ячейке с относительным адресом 0.

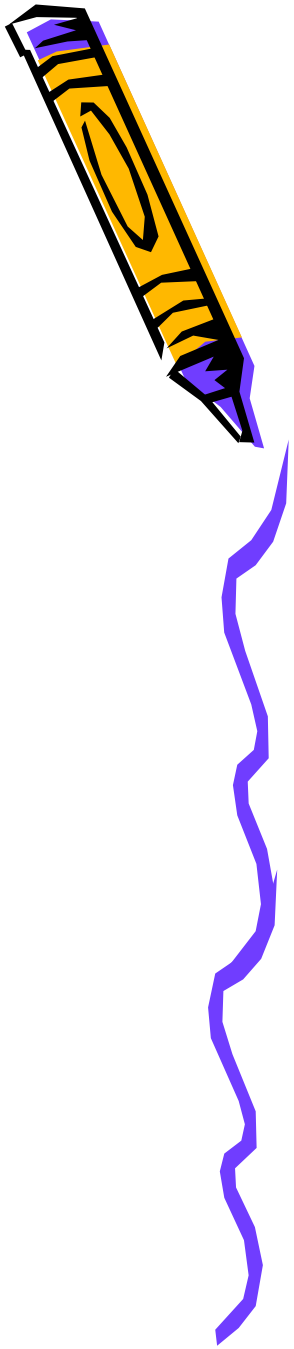
Описание сегмента кучи, в котором может разместиться n элементов размером в двойное слово, может быть таким.

```
--------------------
Heap_size  EQU n   ; n – количество элементов в списке
heap       segment
Heap_ptr   DW  ?    ; ячейка с начальным адресом ССП
           DD heap_size dup (?)  ; n слов в куче
Heap       ends
```

Так описали сегмент кучи, адрес начала кучи должен храниться в регистре ES и программист сам должен загрузить его в этот сегмент. Кроме того, байты этого сегмента нужно объединить в список ССП, например, так чтобы первая ячейка была первым элементом ССП и т.д. Heap_ptr имеет нулевой относительный адрес.



Инициализация кучи и загрузка ее начала в ES

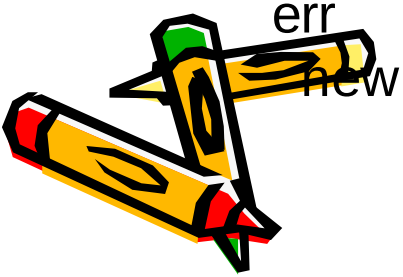


```
init_heap       proc far
                push SI
                push BX
                push CX
; установка ES  на начало кучи
                mov CX,  heap
                mov ES,  CX
; объединение всех двойных слов в ССП
                mov CX,  heap_size
                mov BX,  null
                mov SI,  4*heap_size – 2
          in1:  mov ES:[SI].next,  BX
                mov BX,  SI
                sub SI,  4
                loop in1
                mov ES:heap_ptr,  BX
                pop CX
                pop BX
                pop SI
                ret
init_heap       endp
```

К процедуре init_heap необходимо обращаться до обращения к процедурам new и dispose.

Процедура создания динамической переменной:

```
        ;на выходе процедуры в DI будет адрес нового элемента
        new         proc far
                    mov DI,  ES:heap_ptr ; DI = null или адресу 1-го элемента
                    cmp DI,  null
                    je empty_heap ; если ССП пуст, ——→ empty_heap
                    push ES:[DI].next
                    pop ES:heap_ptr ; указатель на второй элемент кучи
                    ret
  empty_heap: lds DX, CS:aerr  ; реакция на пустую кучу
        ; DS:DX – адрес сообщения об ошибке
                    outstr              ; макрос вывода этого сообщения
                    finish              ; макрос останова программы
           aerr     DD err              ; абсолютный адрес сообщения
            err     DB ‘ошибка в new: исчерпание кучи’,’$’
            new     endp
```

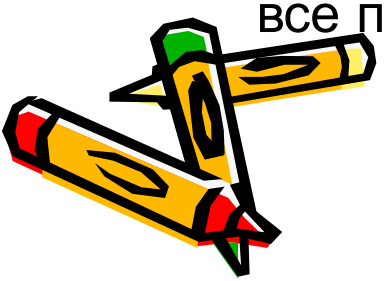


Процедура освобождения динамической памяти dispose

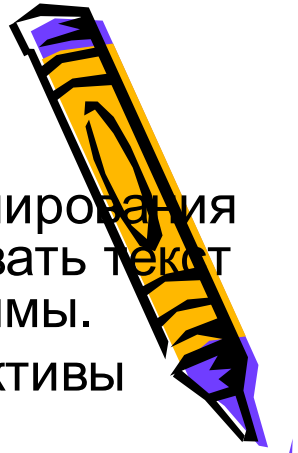
Процедуре **dispose** адрес удаляемого элемента передается в регистре DI, освобождаемая память присоединяется к ССП в его начало, как к односвязному списку:

```
dispose                 proc far
        ; на входе адрес удаляемого элемента в регистре DI
                push ES:heap_ptr
                pop  ES:[DI].next  ; DI.next = heap_ptr
                mov  ES:heap_ptr, DI ; heap_ptr = DI
                ret
dispose                 endp
```

Все процедуры рассчитаны на случай, когда все элементы всех списков имеют один и тот же размер. Если в программе используются списки с элементами различного размера, то все процедуры будут сложнее, но принцип тот же.



Макросредства языка Ассемблер



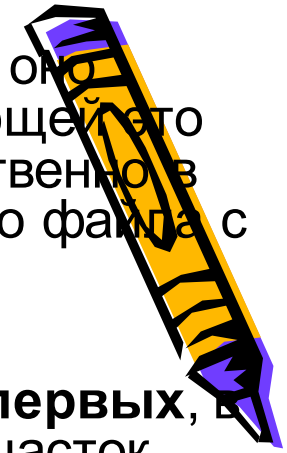
Макросредства называют самым мощным средством прогрммирования в Ассемблере. Они позволяют генерировать, модифицировать текст программы на Ассемблере в процессе трансляции программы.

К макросредствам относят: блоки повторений, макросы, директивы условной генерации.

Программы, написанные на макроязыке, транслируются в два этапа. Сначала она переводится на «чистый» язык Ассемблера, т.е. преобразуется к виду, в котором нет никаких макросредств, этот этап называют **макрогенерацией**. Затем выполняется ассемблирование - перевод в машинные коды. Макрогенерацию называют ещё **препроцессорной обработкой**.

Блоки повторения в процессе макрогенерации заменяются указанной последовательностью команд столько раз, сколько задано в заголовке блока, причем набор команд может повторяться в неизменном или модифицированном виде, в зависимости от вида заголовка блока. Набор команд повторяется *n* раз в том месте программы, где указан блок повторения.

Макросы более похожие на ПП. Аналогично ПП существует описание макроса и обращение к нему. Описание макроса называют **макроопределением**, а обращение - **макрокомандой**. Процесс замены макрокоманды на макрос - **макроподстановкой**, а результат этой подстановки - **макрорасширением**.

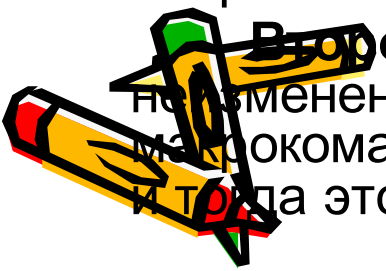


Макроопределение не порождает никаких машинных команд, оно должно предшествовать первой макрокоманде, использующей это макроопределение, и может располагаться как непосредственно в тексте программы, так и может быть подключено из другого файла с помощью директивы

INCLUDE <имя файла>.

Основное отличие макроса от процедуры заключается, **во-первых**, в том, что при обращении к ПП управление передаётся на участок памяти, в котором содержится описание ПП, а при обращении к макросу его тело (макроопределение) вставляется на место макрокоманды, т.е. сколько раз мы обратимся к макросу, сколько макрокоманд будет в программе, столько раз повторится макроопределение, вернее, макрорасширение. Макрос «размножается», увеличивая размер программы. Таким образом, применение процедур дает выигрыш по памяти, но использование макросов дает выигрыш по времени, т.к. нет необходимости передавать управление в ПП и обратно (CALL и RET), а также организовывать передачу параметров.

Рекомендация: если повторяются большие фрагменты программ, лучше использовать процедуры, если относительно небольшие, то макросы.



Второе отличие заключается в том, что текст процедуры неизменен, а содержание макрорасширения зависит от параметров макрокоманды, если используются директивы условной генерации, и тогда это существенно.

Блоки повторений

Общий вид блока повторений:

<заголовок>

<тело>

endm

<тело> - любое количество любых операторов, предложений, в том числе и блоков повторений.

endm определяет конец тела блока. Количество повторений тела и способ модификаций тела блока зависит от заголовка.

Возможны следующие заголовки:

1) REPT n ; n - константное выражение

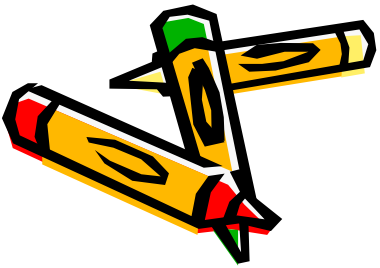
Оно может быть вычислено на этапе макрогенерации, в результате которого n копий тела блока записывается в данном месте программы на Ассемблере. Например:

<u>В исходном тексте</u>

```
N   EQU  8
        REPT  N-6
        DB  0,1
        DW  ?
        endm
```

<u>После макрогенерации на этом месте</u>

```
N  EQU  8
        DB  0,1
        DW  ?
        DB  0,1
        DW  ?
```



Для создания массива с начальными значениями от 0 до OFFH достаточно написать блок повторений:

```
n = 1
mas    DB 0          ;   имя массива mas
Rept   255   ;   начало блока
       DB  n
       n = n + 1
endm
```

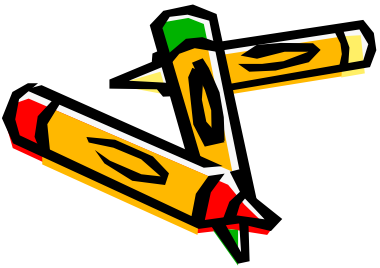
2) Второй вид заголовка:

IRP P , <V1,V2,…Vk> ; < и > обязательные символы

<тело> ; тело повторяется k раз так, что в i-той копии

endm

формальный параметр P замещается фактическим параметром Vi. Формальный параметр P - это локальное имя, не имеющее смысла вне блока. Если оно совпадает с именем другого какого-либо объекта программы, то в теле блока это просто имя, а не этот объект. Например:



1) IRP reg, <AX, BX, CX, SI>

```
    push  reg
endm
```

После макрогенерации

```
push  AX
push  BX
push  CX
push  SI
```

2) **IRP BX , <5,7,9>**
add AX, BX
endm

→

add AX , 5
add AX , 7
add AX , 9

Здесь BX - символическое имя, но не имя регистра BX.

Причём, замена формального параметра на фактический - это просто текстовые замены, один участок программы P заменяется на другой – Vi , т.е. P может обозначать любую часть предложения или все предложение, лишь бы после замены P на Vi получилось правильное предложение языка Ассемблер.

3) **IRP R , <dec word ptr, L: inc word ptr>**
R W
jmp M
endm

→

dec word ptr W
jmp M
L: inc word ptr W
jmp M

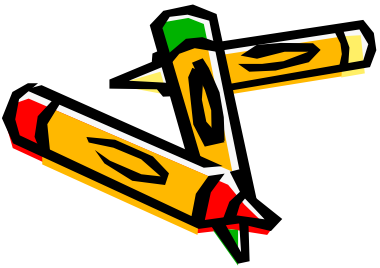
Здесь параметром является имя команды и тип операнда.

3) Вид заголовка: **IRPC P , S1S2….SK**

```
IRPC  P ,  S1S2….SK
        <тело>
 endm
```

P - формальный параметр, Si –символы, любые, кроме пробелов и точки с запятой, если необходимо использовать здесь пробел или точку с запятой, то надо всю последовательность символов записать в угловых скобках. Встречая такой блок, макрогенератор заменяет его на k копий тела так, что в i-той копии параметр P заменен на символ Si. Например:

```
IRPC A,  175P                          add  AX , 1
        add  AX,  A        →           add  AX , 7
endm                                   add  AX , 5
                                       add  AX , P
```



Макрооператоры

В макроопределениях и в блоках повторения могут использоваться специальные операторы Ассемблера, называемые макрооператорами, для записи формальных и фактических параметров.

1) & - амперсанд – используется для того, чтобы указать границы формального параметра, выделить его из окружающего текста, при этом в текст программы он не записывается. Например:

а)
```
IRP  W,  <1,5,7>                 var1  DW ?
 VAR&W  DW?     →                 var5  DW ?
endm                              var7  DW ?
```

б)
```
IRPC  A, " <                      DB  'A, ", "B'
DB     'A, &A, &A&B'     →        DB  'A, <, <B'
endm
```

Здесь параметры W и A заменяются на фактические параметры только в том месте, где они выделены макрооператором &.

Если знаков & рядом несколько, то макрогенератор удаляет за один проход только один из них, и это используется для организации вложенных блоков повторений и макросов. Например:

……..

```
IRPC  P1, AB                IRPC  P2, HL                inc  AH
 IRPC  P2, HL                 inc  A&P2                  inc  AL
  inc  P1&&P2       →        endm              →        inc  BH
 endm                        IRPC  P2, HL                inc  BL
endm                          inc  B&P2
                             endm
```

2) Макрооператор < > - угловые скобки действует так, что весь текст, заключенный в эти скобки, рассматривается как одна текстовая строка, и в неё могут входить пробелы, запятые и другие разделители. Этот макрооператор часто используется для передачи текстовых строк в качестве параметров для макросов и для передачи списка параметров вложенному макроопределению или блоку повторений.

```
a)  IPR  V , <<1,2>,3>          DB  1,2
     DB V                  →      DB 3
    endm
```

б) **IRPC S, <A; B>**
DB ´S´
endm

→

DB ´A´
DB ´ ; ´
DB ´B´

Если в примере б) скобок < > не будет, то символ В будет восприниматься как комментарий после ;

3) Макрооператор ! - восклицательный знак используется аналогично угловым скобкам, но действует только на один следующий символ, так что, если этим символом является один из символов ограничения - запятая, угловая скобка и т.д., то он будет передаваться как параметр или часть параметра.

4) Макрооператор % - процент указывает на то, что следующий за ним текст является выражением, которое должно быть вычислено, и результат передается как параметр. Например:

K EQU 4
……
IRP A, <k+1, % k+1, W% k+1>
DW A
endm

→

DW k+1
DW 5
DW W5

5) Макрооператор ;; - две точки с запятой определяют начало макрокомментария. Текст макрокомментария не включается в макрорасширения и в листинг программы.

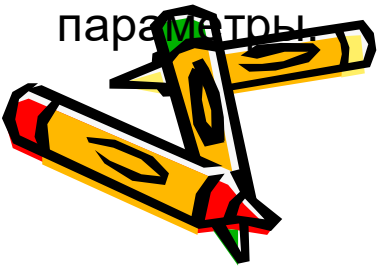
Макросы.

Описание макроса, **макроопределение**, имеет вид:

<имя макроса> Macro <формальные параметры>
LOCAL <список имен>
<тело>
endm

Первая строка - это заголовок макроса, имя макроса будет использоваться для обращения к этому Макроопределению. Формальные параметры записываются через запятую, это локальные имена, никак не связанные с объектами программы. Количество Формальных параметров не ограничено, но они должны умещаться в одной строке. Поскольку на место каждой Макрокоманды записывается Макрорасширение, кроме того, одни и те же метки могут использоваться и в самой программе, чтобы не возникало ошибки «метка уже определена», директива LOCAL <список имен> перечисляет через запятую имена меток, которые будут использоваться в теле макроса.

<тело> - это копируемый фрагмент программы, любое количество любых предложений Я.А., в которых используются формальные параметры.



Макрокоманда – обращение к макросу:
<имя макроса> <фактические параметры>

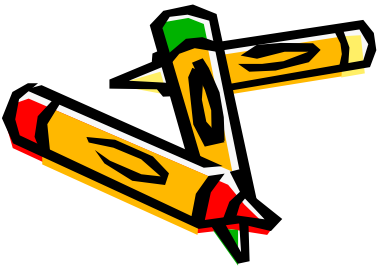
Фактические параметры указываются через запятую или/и пробел. В качестве фактического параметра может быть использован любой текст, в том числе и пустой, но он должен быть сбалансирован по кавычкам и угловым скобкам, и в нем не должно быть запятых, пробелов и точек с запятой вне кавычек и скобок, т.к. запятая и пробел могут отделять один параметр от другого, а точкой с запятой начинается комментарий.

С помощью директивы **EXITM** можно осуществить досрочный выход из макроса, если использовать команды условной генерации **IF x … endif**.

С помощью директивы

PURGE <имя макроса>

можно отменить определенный ранее макрос. Эта директива часто используется сразу после директивы INCLUDE, включившей в текст программы файл с большим количеством готовых макроопределений.



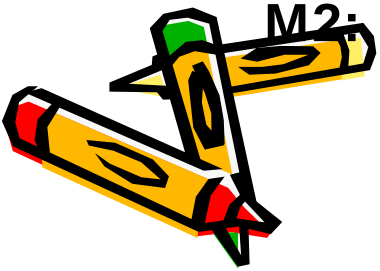
Примеры макросов

1) Использование макросов позволяет составлять программу в терминах более крупных операций. Опишем в виде макроса оператор **IF x< y then GOTO L.**

```
IF_L   MACRO  x, y, L
          mov   AX, x
          cmp   AX, Y
          jl     L
       endm
```

Используя этот макрос, поиск минимального из 3-х чисел запишется так:

```
          ; DX = min (A,B,C)
          mov DX,  A
          IF_L  A, B, M1
          mov DX,  B
M1:       IF_L  DX,  C,  M2
          mov DX,  C
M2:       ---------------------
```



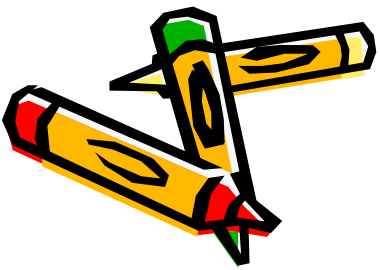
После макрогенерации в программе будет текст:

```
-----------------------------
        mov DX,  A
        mov AX,  A
        cmp AX,  B
        jl,  M1
        mov DX,  B
M1:     mov AX,  DX
        cmp AX,  C
        jl  M2
        mov DX,  C
M2:             ----------------------------
```

2) Обращение к процедурам будет нагляднее, если передачу параметров оформить как макрос.

Например: Вычислить CX = NOD(A,B) + NOD(C,D), если есть процедура вычисления NOD(x,y), и результат ее находится в AX.

```
CALL_NOD MACRO x, y
        mov AX, x
        mov BX, y
        call NOD      ;  (AX) = NOD(x, y)
endm
```



CALL_NOD A, B ; наглядное обращение к ПП с параметрами.
mov CX, AX ; (CX) = NOD(A,B)
CALL_NOD C, D ; (AX) = NOD(C,D)
add CX, AX ; (CX) = NOD(A,B) + NOD(C,D)

Использование меток в макросах.

Пример 1. Если в макроопределении используются метки, но нет директивы Local , то после макрогенерации будет сообщение об ошибке – дублирование меток:

макроопределение	макрокоманда	макрорасширение
		…………
M Macro	M	→ L: …………
…………		…………
L: …………		---- --------------------
endm		…………
	M	→ L: …………
		…………

Вычислить остаток от деления одного числа на другое с помощью вычитания (числа натуральные, r1 и r2 - регистры).

```
MD  Macro  r1, r2        ;   r1 = r1 mod r2
        Local    M,  M1
M:   cmp   r1,  r2       ;   while  r1 >= r2  DO  r1 = r1 – r2
       jb       M1
       sub     r1, r2
       jmp     M
M1:
      endm
```

Обращения к макросу MD: **(r1 = AX, r2 = BX)**

```
1)     MD  AX, BX          ??0000 :        cmp  AX, BX
                                           jb     ?? 0001
                                           sub    AX,  BX
                                           jmp    ?? 0000
                                ??0001: ---------------
```

2) MD CX, DX ; (r1 = CX, r2 = DX)

```
                    ----------
??0002 :    cmp  CX, DX
            jb     ??0003
            sub    CX,  DX
            jmp    ??0002
??0003:     ----------
```

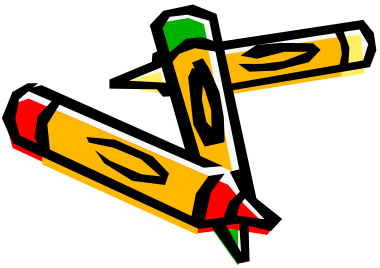
Макрогенератор, встретив директиву Local M, M1, будет заменять метки M и M1 на специальные имена вида ??xxxx,

где xxxx – четырехзначное 16-ричное число от 0000÷FFFF.

Директивы условного ассемблирования (ДУА).

ДУА управляют процессом ассемблирования путем подключения или отключения фрагментов исходного текста программы. Общий вид:

1)
```
if  <выражение>
        if- часть
[ else
        else-часть ]
endif
```



2) if <выражение >

lf-часть

elseif <выражение 1>

elseif-часть 1

elseif <выражение 2>

elseif-часть 2

[else

else-часть]

endif

ДУА в форме 2) используется аналогично операторам выбора в языках высокого уровня. ДУА много, рассмотрим некоторые из них:

1) if <константное выражение> if-часть

if-часть ассемблируется, включается в исходный текст программы, если значение выражения - истина, т.е. не равно нулю, в противном случае работает else-часть, если она есть, если else-части нет, выполняется следующий за директивой if оператор.

2) а) ife константное выражение

в) elseife константное выражение

if-часть работает, если выражение ложно, равно нулю.

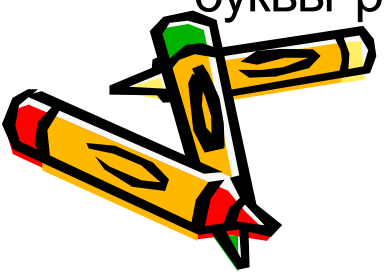
3) a) **ifdef метка** ; if-часть работает, если
b) **elseifdef метка** ; указанная метка определена

4) a) **ifndef метка** ; if-часть работает, если
b) **elseifndef метка** ; указанная метка не определена

5) a) **ifb <аргумент>** ; if-часть работает, если
b) **elseifb <аргумент>** ; значением аргумента является пробел

6) a) **ifnb <аргумент>** ; if-часть работает, если значением
b) **ifnb <аргумент>** ; аргумента является не пробел

7) a) **ifdif <арг1>, <арг2>** ;
b) **elseifdif <арг1>, <арг2>** ;
if-часть работает, если аргументы различны, прописные и строчные буквы различаются



8) a) **ifdifi <арг1>, <арг2>**

b) **elseifdifi <арг1>, <арг2>**

if-часть работает, если аргументы различны, прописные и строчные буквы не различаются

9) a) **ifidn <арг1>, <арг2>**

b) **elseifidn <арг1>, <арг2>**

if-часть работает, если аргументы одинаковы, прописные и строчные буквы различаются

10) a) **ifidni <арг1>, <арг2>**

b) **elseifidni <арг1>, <арг2>**

if-часть работает, если аргументы одинаковы, прописные и строчные буквы не различаются

Здесь угловые скобки обязательны.

Примеры использования ДУА

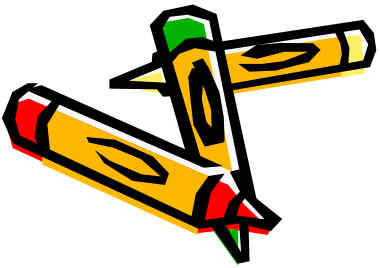
1) Использование ДУА непосредственно в Ассемблере. При отладке большой программы обычно используются отладочные печати для проверки правильности работы отдельных ее участков. В отлаженной программе эти печати удаляются. Но если отладка выполняется в несколько этапов, то добавлять печати в программу и исключать их – это тоже громоздкая работа, во время которой можно внести новые ошибки.

Используя ДУА, задачу включения в исходный текст программы и исключения из нее отладочных печатей перекладываем на макрогенератор, а он не ошибается. Для этого программисту нужно определить константу, установив таким образом режим отладки или счета. Например,

debug EQU 1 - отладка, debug EQU 0 - счет

а в программе должно быть:

```
mov x, AX
if debug
OutInt x
endif
mov BX, 0
```



После макрогенерации в программе появится текст

если debug < > 0:

```
mov x,  AX
OutInt  x
mov BX,  0
```

если debug = 0:

```
mov x,  AX
mov BX,  0
```

Здесь OutInt – макрокоманда обращения к процедуре вывода.

Решить задачу, чтобы отладочная печать работала в режиме отладки и не работала при счете, можно и с помощью команды сравнения и условного перехода:

```
      mov x,  AX
      mov CL,  debug
      cmp CL,  1
      jne   L
      OutInt  x
L:    mov BX,  0
      ---------------------------
```

Однако, этот текст всегда будет занимать место в памяти и, если таких участков будет много, то исходный текст будет громоздким и время выполнения программы будет существенно больше. При использовании условной генерации время тратится только на этапе ассемблирования. При счете этих операций в исходном тексте не будет.

Использование ДУА в макросах

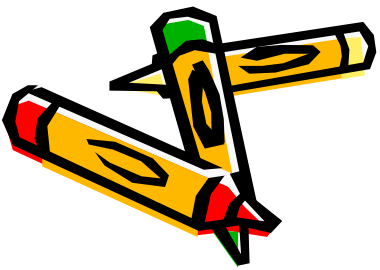
Пример 1. Опишем в виде макроса операцию сдвига значения переменной x на n разрядов вправо. n – явно заданное положительное число. Макрорасширение должно содержать минимально возможное число команд. Это можно сделать так:

```
shift   macro  x,  n
        ife  n – 1          ; n – 1 = 0 ?
          shr x,  1
        else                ; n > 1
          mov CL,  n
          shr  x,  CL
        endif
        endm
```

Обращения: `shift  A,  5` ; x = A, n = 5

После макрогенерации:

```
        mov CL,  5
        shr A,  CL
```



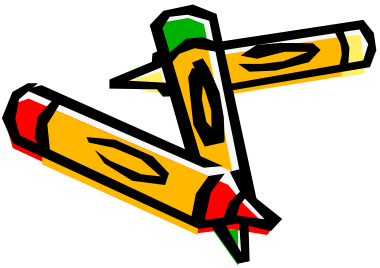
Пример 2

Константное выражение в if и ife может быть любым, но так как оно вычисляется на этапе макрогенерации, в нем не должно быть ссылок на величины, которые станут известными только при выполнении программы. Например, в константных выражениях не должно быть ссылок на регистры и ячейки памяти, не должно быть ссылок вперед. Константное выражение должно быть вычислено макрогенератором при первом проходе.

Константное выражение часто бывает логическим, в нем могут использоваться операторы отношения: EQ, NE, LT, LE, GT, GE и логические операторы NOT, OR, XOR.

Запишем в виде макроса **SET_0 x** операцию x = 0, если x – переменная размером в байт, слово или двойное слово.

```
SET_0 macro x
        if type  x EQ dword
          mov dword ptr x,  0
        elseif  type x EQ word
          mov  word ptr x,  0
        else  mov byte ptr x,  0
        endif

endm
```



Пример 3

Напишем еще одно макроопределение для сдвига вправо на n разрядов значения байтовой переменной B. Учтем, что при n = 0 сдвига нет и макрорасширение не должно появиться в тексте программы, а при n > 7 результат сдвига – это 0, поэтому сдвиг можно заменить записью нуля в B.

```
Set_0   macro  B, n
  if  (n GT 0) AND (n LT 8)  ;; 0 < n <8
    mov CL,  n
    shr  b,  CL
  else
        if n GE 8          ;;  n >= 8
          mov  B,  0
        endif
   endif
endm
```

Здесь использовались вложенные if….endif и в макроопределениях комментарий записывается после ;;

Пример 4 использование ifidn и ifdif

Поиск max или min из двух знаковых величин, хранящихся в байтовых регистрах, т.е. вычислить **R1 = T (R1, R2),**

где T – это max или min, причем, должно генерироваться непустое макрорасширение только если R1 и R2 – это разные регистры. Если обращение к макросу будет Max_Min R1, R2, T, то необходимо проверять несовпадение первых двух параметров. И чтобы один макрос вычислял и max и min, нужно проверять и значение третьего параметра. Макрос может быть таким:

```
Max_Min  macro  R1, R2, T
         local L
         ifdif  <R1>, <R2>  ;; R1 и R2 – разные регистры
         cmp  R1,  R2
            ifidn <T>,  <max>  ;; T = max ?
              jge  L
            else
              jle  L
            endif
                mov R1,  R2
       L:   endif
endm
```

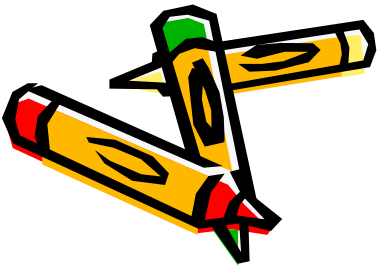
Макрокоманда Max_Min AL, BH, MIN приведет к следующим действиям макрогенератора: (пусть метка L заменилась меткой вида ??0110)

```
-------------------------
ifdif  <AL>,  <BH>
cmp  AL,  BH
     ifidn  <MIN>,  <MAX>
       jge  ??0110
      else
       jle   ??0110
      endif
       mov AL,  BH
??0110:
endif
```

```
cmp  AL,  BH
ifidn <MIN>, <MAX>
   jge   ??0110
else
  jle   ??0110
endif
  mov  AL,  BH
??0110:
--------------------------
```

в исходном тексте программы окажется

```
cmp AL,  BH
 jle   ??0110
 mov  AL,  BH
??0110:
--------------------
```



Пример многомодульной программы

Предположим, что есть модуль, содержащий процедуры ввода и вывода символов и строк, который подключается к основной программе на этапе редактирования:

masm p.asm, p.obj, p.list

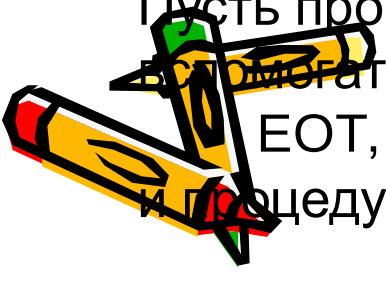
link p.obj + ioproc.obj, p.exe

p.exe

Есть файл **io.asm**, содержащий описания макросов обращения к этим процедурам. Этот файл подключается на этапе ассемблирования с помощью директивы **include io.asm.**

Для иллюстрации организации многомодульной программы решим задачу: ввести текст не более, чем из 100 символов, заканчивающийся точкой и вывести его в обратном порядке, заменив прописные буквы на строчные. Эту задачу можно решить проще, но….

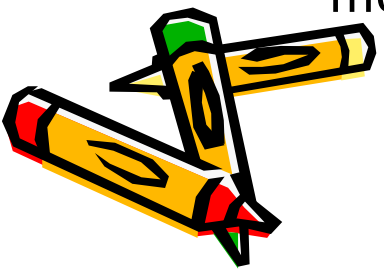
Пусть программа состоит из двух модулей - головного и вспомогательного. Во вспомогательном описывается переменная EOT, значением которой является символ конца ввода текста, и процедура LOWLAT, заменяющую прописную на строчную.



Головной модуль должен вводить текст, записывать его в массив в обратном порядке, обращаясь к процедуре LOWLAT для замены больших букв на малые, а затем выводить этот массив на экран.

```
; вспомогательный модуль
    public EOT, LOWLAT
    D1  segment
        EOT  DB '.'     ; символ конца ввода
    D1  ends
    C1  segment
        assume CS: C1
        LOWLAT proc far
; процедура перевода больших букв в малые,
; на входе (AL) – любой символ, на выходе (AL) – малая буква
                cmp AL,  'A'  ; AL < 'A' или AL > 'Z', то      nolat
                jb  nolat
                cmp AL,  'Z'
                ja  nolat
                add  AL, - 'A' + 'a'
            nolat: ret
        LOWLAT endp
    C1 ends
                end
```

```
; головной модуль
include io.asm
extrn  EOT: byte, LOWLAT: far
s  segment stack
    DB 256 dup (?)
s  ends
d  segment
    txt DB 100 dup (?), '$'
d ends
c segment
    assume SS: s, DS: d,  CS: c
start: mov AX,  d
        mov DS,  AX             ; DS = d для доступа к TXT
        mov AX,  seg EOT
        mov ES,  AX             ; ES = D1 для доступа к EOT
        mov SI, 100
            OutCH '>'           ; приглашение к вводу символа
```



```
inp: InCH  AL          ; ввод символа
     cmp AL,  ES:EOT   ; если достигнут конец ввода
     je   PR           ; то ⟶ PR
     call LOWLAT       ; замена символа
      dec SI
     mov TXT[SI],  AL
     jmp  inp
PR:  lea DX,  TXT[SI]  ; вывод
     OutSTR            ; на экран TXT
     Finish
c ends
     end  start
```

В основной программе OutCH <параметр> и InCH <параметр> - макрокоманды ввода и вывода на экран параметра

OutSTR - макрокоманда вывода строки

```
Finish  macro    ; макрос окончания счета
        mov AH,  4Ch
        int  21h
endm
```