

МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

имени М.В.ЛОМОНОСОВА

Факультет вычислительной математики и кибернетики

И.В.Горячая

Строковые команды в Masm 6.14:

Учебно-методические материалы

(издание 2-е дополненное и исправленное)

Настоящие материалы ориентированы на студентов первого курса факультета ВМК МГУ имени М.В.Ломоносова и призваны облегчить им освоение строковых команд в рамках дисциплины «Архитектура ЭВМ и язык ассемблера» (архитектура IA-32, язык ассемблера MASM 6.14).

В представленных материалах изложены правила и особенности выполнения строковых команд, разобраны типовые примеры их использования.

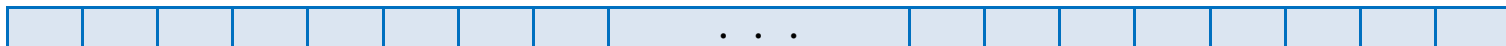
Отдельное внимание уделено вопросам представления строк переменной длины и специфике их обработки. Материалы сопровождаются наглядными иллюстрациями, способствующими более глубокому пониманию рассматриваемых вопросов.

Во второе издание были внесены некоторые дополнения и изменения в соответствии с текущим содержанием лекций.

Москва - 2026

Под **строкой** будем понимать одно из трёх:

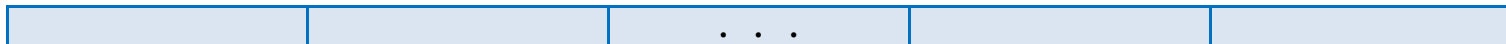
- последовательность *соседних байтов*:



- последовательность *соседних слов*:



- последовательность *соседних двойных слов*:

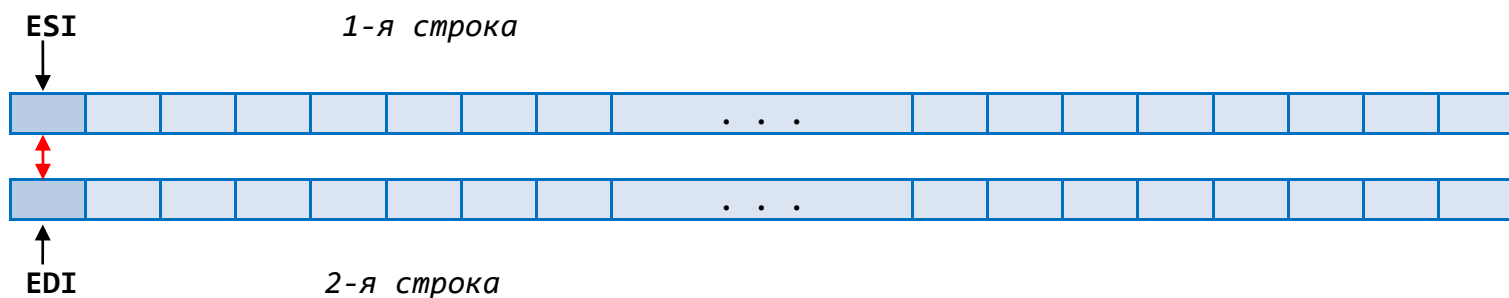


Соответственно, каждая строковая команда представляется в трёх вариантах:

- для работы со строками из *байтов*, записывается в виде **<мнемокод>b**
- для работы со строками из *слов*, записывается в виде **<мнемокод>w**
- для работы со строками из *двойных слов*, записывается в виде **<мнемокод>d**

ПУНКТ 1 Сравнение строк из *байтов* (**cmpsb**), *слов* (**cmpsw**) и *двойных слов* (**cmpsd**)

Операнды в команде явно не указываются, так как их местоположение заранее известно.



Команда сравнивает *текущие элементы* 1-й и 2-й строк.

При этом строго требуется, чтобы **адрес** (смещение) текущего элемента **1-й строки** задавался регистром **ESI**, а **адрес** (смещение) текущего элемента **2-й строки** задавался регистром **EDI**. О подготовке содержимого этих регистров программист должен позаботиться заранее.

Другая важная установка, которую следует выполнить перед началом сравнения строк: задать **направление обработки строк** (от начала к концу или от конца к началу). Заметим, что *текущее* направление обработки строк зависит от состояния специального **флага направления DF** (**Direction Flag**):

$$DF = \begin{cases} 0 & \text{(строка просматривается } \textit{вперёд}, \text{ в направлении } \textit{возрастания адресов}) \\ 1 & \text{(строка просматривается } \textit{назад}, \text{ в направлении } \textit{убывания адресов}) \end{cases}$$

Внимание! В начальный момент работы программы **DF** имеет *неопределённое значение* (т.е. какого-либо значения по умолчанию для **DF** не предусмотрено).

Состоянием флага **DF** можно управлять с помощью следующих команд:

cld ; задать просмотр строки *вперёд* (**Clear Direction Flag**, т.е. **DF := 0**)

std ; задать просмотр строки *назад* (**Set Direction Flag**, т.е. **DF := 1**)

Действие команды **cmpsb/cmpsw/cmpsd**: **сравнение** пары *текущих* элементов (байтов/слов/дв.слов) и *автоматическая настройка* на пару *соседних* элементов.

1) **[ESI]-[EDI], + флаги**

Из *текущего элемента* (с адресом **[ESI]**) 1-й строки вычитается *текущий элемент* (с адресом **[EDI]**) 2-й строки. Разность никуда не записывается, но по результату вычитания формируются флаги (**CF**, **OF**, **ZF**, **SF**).

2) **ESI := ESI ± 1/2/4, EDI := EDI ± 1/2/4**

Регистры **ESI** и **EDI** автоматически настраиваются на соседние элементы (байты/слова/дв.слова) – в зависимости от направления просмотра строк (**“плюс”** при **DF=0** и **“минус”** при **DF=1**).

Замечание: символ **/** обозначает (здесь и далее) один из нескольких вариантов.

Заметим, что команда **cmpsb/cmpsw/cmpsd** сравнивает только *ПАРУ текущих элементов* из обеих строк. Чтобы применить эту команду ко *ВСЕМ ПАРАМ* элементов, можно организовать обычный цикл с использованием команды **loop**. Но для достижения *бóльшего быстрогодействия* лучше выписать перед соответствующей командой подходящий **префикс повторения** (машинный эквивалент которого является однобайтовой командой): **<префикс повторения> cmpsb/cmpsw/cmpsd ; всё в одной строке !!!**

Важно: следует предварительно настроить **ECX** на *максимально возможное число шагов* цикла (обычно оно соответствует количеству элементов в строке, т.е. *длине строки*).

Различают 2 вида префиксов повторения:

repE/repZ

обе указанных выше мнемоники *эквиваленты* (т.е. транслируются в одну и ту же машинную команду)

repNE/repNZ

обе указанных выше мнемоники *эквиваленты* (т.е. транслируются в одну и ту же машинную команду)

Рассмотрим более подробно, каково назначение и действие этих префиксов.

repE/repZ

Используются для поиска **первой неравной пары** элементов.

Действие (цикл с предусловием):

```
L:  if ECX=0 then goto skip
    ECX := ECX-1
    cmpsb/cmpsw/cmpeb ; ZF - ?
    if ZF=1 then goto L
skip:
```

Исходы:

все пары совпали	нашли несовпадающую пару элементов
ZF=1, ECX=0	ZF=0, ECX - любое

repNE/repNZ

Используются для поиска **первой равной пары** элементов.

Действие (цикл с предусловием):

```
L:  if ECX=0 then goto skip
    ECX := ECX-1
    cmpsb/cmpsw/cmpeb ; ZF - ?
    if ZF=0 then goto L
skip:
```

Исходы:

все пары не совпали	нашли совпадающую пару элементов
ZF=0, ECX=0	ZF=1, ECX - любое

Вывод: для выяснения, **с каким исходом вышли из цикла**, нужно ориентироваться по значению флага нуля **ZF**.

Предостережение. Если до начала цикла (с использованием префикса повторения) значение регистра **ECX** было нулевым, то флаг **ZF** после цикла будет таким же, как и до начала цикла (т.к. цикл не выполнится ни разу). В этом случае анализ флага **ZF** нам ничего не даст. Следовательно, если сравниваемые строки могут быть ПУСТЫМИ, то до входа в цикл нужно проверить значение **ECX** (с помощью команды **jECXz**) и, в случае нуля, выполнить обход команд, анализирующих **ZF**.

Пример 1

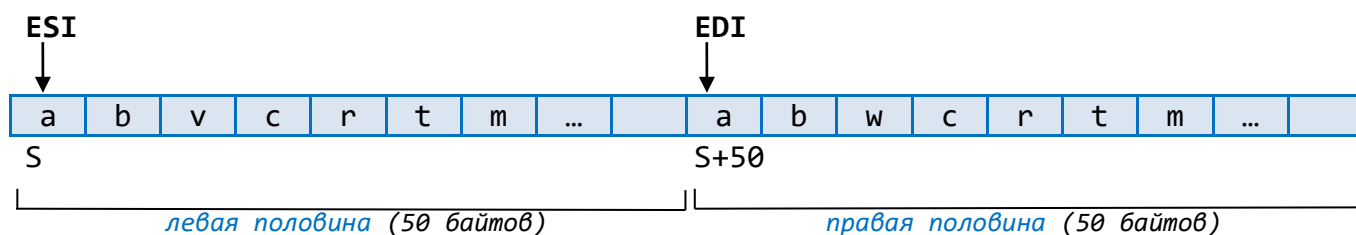
S db 100 dup(?) ; S[0..99] of char

S – символьная строка в секции данных. Определить, **равны ли** левая и правая половины строки **S**, и записать ответ **1** (равны) или **0** (не равны) в регистр **AL**.

Решение.

Делим исходную строку на две подстроки и применяем к ним команду сравнения строк.

Состояние **после** настроек **ESI**, **EDI** должно быть таким:



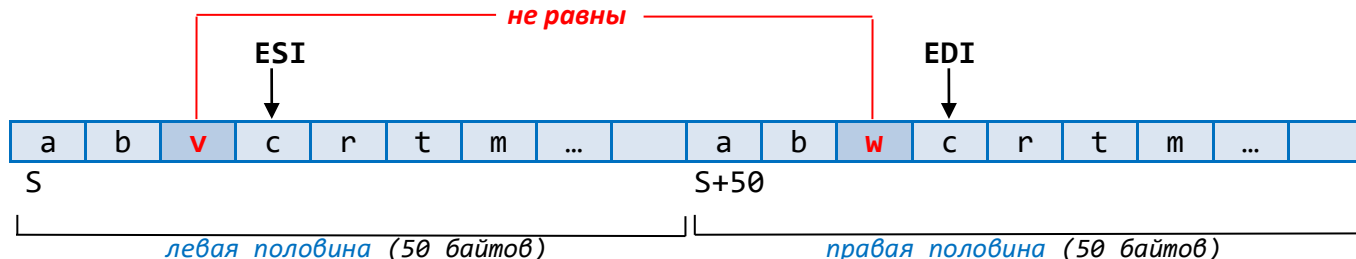
```
lea ESI,S      ; ESI - на начало левой половины
lea EDI,S+50   ; EDI - на начало правой половины
cld           ; задаём просмотр строки вперёд
mov ECX,50     ; максимальное число сравнений
mov AL,0       ; по умолчанию считаем, что не равны
```

```

repE cmpsb      ; “крутись” в цикле, пока сравниваемые элементы равны
; возможные исходы:
; ZF=1 (все пары были равны, вышли по концу цикла при ECX=0)
; ZF=0 (попалась первая неравная пара, ECX м.б. любым)
; по флагу нуля (ZF) определяем окончательный ответ:
jNZ Fin        ; if ZF=0 then goto Fin (т.е. не равны)
mov AL,1       ; половинки оказались равны
Fin:

```

Состояние **после** обнаружения **первой неравной пары** (и досрочного выхода из цикла):



Наблюдения (важные!) по этой задаче:

1) Пусть досрочно вышли из цикла (т.к. попалась пара неравных элементов). Тогда регистры **ESI** и **EDI** будут указывать не на элементы, из-за которых вышли из цикла, а на **следующие за ними элементы** (см. рисунок выше).

Объяснение: команда сравнения строк устроена так, что **независимо от результата сравнения** идёт **автоматическая настройка на соседние элементы**.

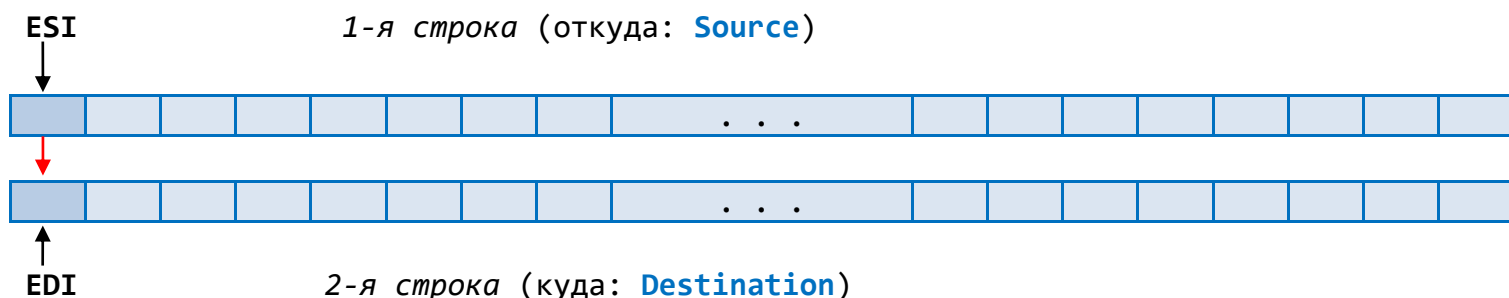
2) Как узнать, сколько пар осталось нерассмотренными?

Ответ: **ECX** пар.

Объяснение: сколько пар сравнили, столько и “выбросили” из **ECX**. Всё, что в **ECX** осталось – мы не рассмотрели.

Эти два наблюдения будем активно использовать при решении дальнейших задач.

ПУНКТ 2 Пересылка строки из байтов (**movsb**), слов (**movsw**) и двойных слов (**movsd**)



ESI задаёт строку-источник (**Source**), а **EDI** задаёт строку-назначение (**Destination**).

Действие команды **movsb/movsw/movsd**: **пересылка** текущего элемента (байта/слова/дв.слова) из 1-й строки в текущую позицию 2-й строки, и автоматическая настройка на соседние элементы. Команда **не меняет флагов**!

1) [ESI] => [EDI]

Текущий элемент (с адресом [ESI]) 1-й строки пересылается в текущую позицию (с адресом [EDI]) 2-й строки. Флаги (CF, OF, ZF, SF) не меняются.

2) ESI := ESI ± 1/2/4, EDI := EDI ± 1/2/4

Регистры **ESI** и **EDI** автоматически настраиваются на соседние элементы (байты/слова/дв.слова) – в зависимости от направления просмотра строк (“плюс” при **DF=0** и “минус” при **DF=1**).

Замечание: символ / обозначает (здесь и далее) один из нескольких вариантов.

Чтобы **переслать всю строку**, выписываем перед командой **префикс повторения rep**. В паре с командой пересылки **movsb/movsw/movsd** префикс **rep** заставляет эту команду повториться ровно **ECX** раз. Следовательно, такой цикл обязательно проработает **до конца** (пока содержимое одной строки полностью не переписывается в другую строку).

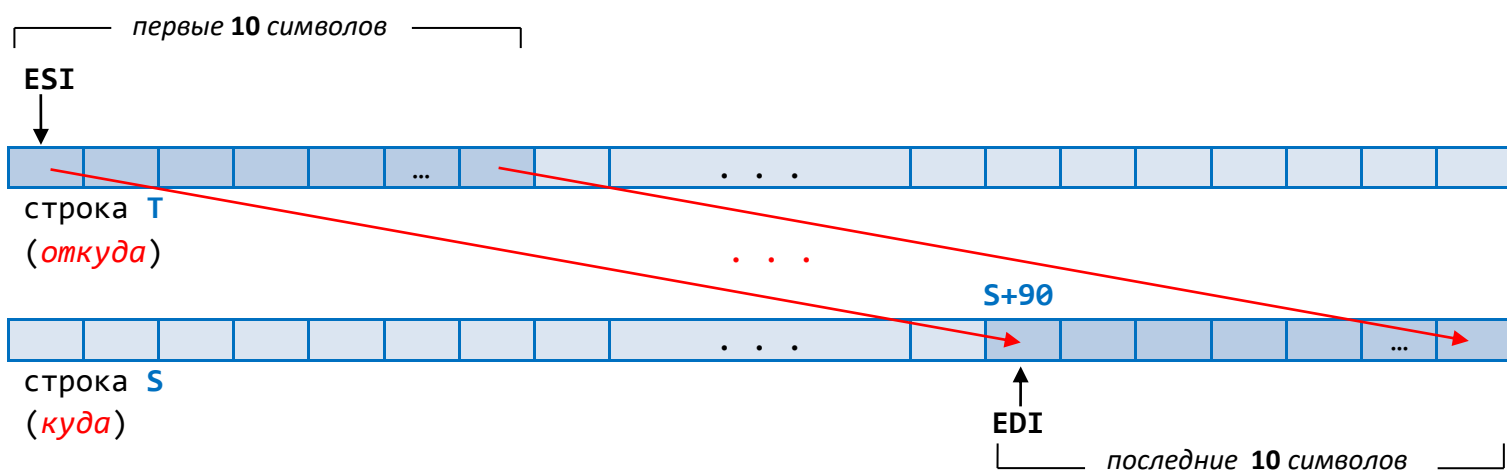
Итак:
rep movsb/movsw/movsd; копирование строки (с адресом в **ESI**) на новое место,
 ; начиная с позиции, адрес которой задаётся регистром **EDI**
 ; (предварительно нужна настройка **ECX** и флага **DF**)

Пример 2

```
T db 100 dup(?)      ; T[0..99] of char
S db 100 dup(?)      ; S[0..99] of char
```

S и **T** – символьные строки в секции данных. Заменить *последние 10 символов* строки **S** на *10 первых символов* строки **T**.

Решение.



```
lea ESI,T      ; откуда
lea EDI,S+90   ; куда
cld           ; просмотр вперёд
mov ECX,10     ; количество копируемых символов
rep movsb     ; зациклили команду movsb 10 раз
```

ПУНКТ 3 Загрузка строки из *байтов* (**lodsb**), *слов* (**lodsw**) и *двойных слов* (**lodsd**)

ПУНКТ 4 Сохранение строки из *байтов* (**stosb**), *слов* (**stosw**) и *двойных слов* (**stosd**)

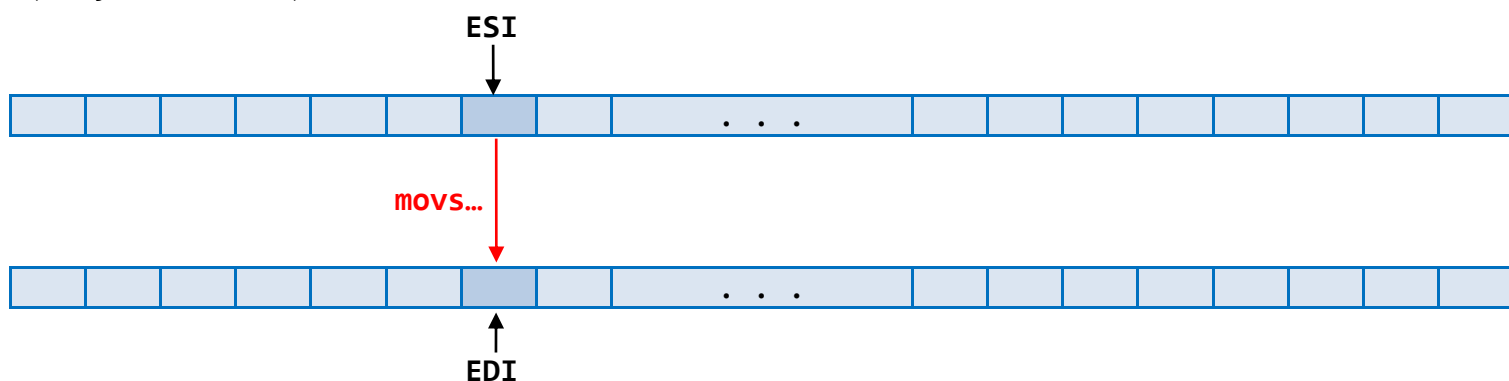
Пункты 3 и 4 рассмотрим **одновременно**.

Вспомним, как работает команда пересылки **movsb/movsw/movsd**.

Эта команда пересылает *текущий элемент напрямую* (безо всяких “промежуточных” пунктов) из строки-источника в строку-назначение:

1-я строка

(откуда: **Source**)



2-я строка

(куда: **Destination**)

ESI задаёт строку-источник (**Source**), а **EDI** задаёт строку-назначение (**Destination**). Здесь вместо троеточия подразумевается одна из трёх букв: **b**, **w** или **d**.

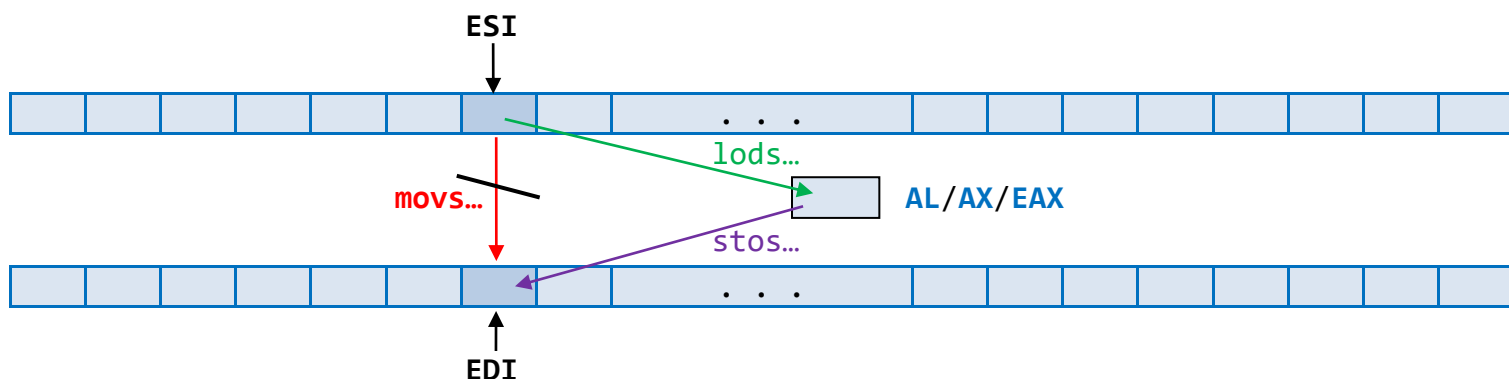
Однако, пересылку элемента можно выполнить **в 2 этапа** в тех случаях, когда необходимо предварительно “поработать” с пересылаемым элементом (прежде чем поместить его на окончательное место). В таких случаях связь ↓ разрывают, помещая элемент предварительно на регистр **AL/AX/EAX** (в зависимости от *размера* элемента и буквы **b**, **w** или **d** в конце *названия команды*). “Поработав” с элементом на регистре, далее его пересылают в строку-назначение.

Эти действия обеспечиваются парой следующих команд:

lodsb/lodsw/lodsd и **stosb/stosw/stosd**

Рассмотрим сказанное более подробно.

1-я строка
(откуда: **Source**)



2-я строка
(куда: **Destination**)

ESI задаёт строку-источник (**Source**), а **EDI** задаёт строку-назначение (**Destination**).
Здесь вместо троеточия одна из букв: **b**, **w** или **d**

Действие команды **lods**/**lodsw**/**lodsd**: **пересылка** текущего элемента (байта/слова/дв.слова) из строки-источника в регистр **AL/AX/EAX**, и автоматическая настройка на соседний элемент (в строке-источнике). Команда **не меняет флагов**!

1) **[ESI] => AL/AX/EAX**, флаги (**CF**, **OF**, **ZF**, **SF**) не меняются.

2) **ESI := ESI ± 1/2/4** (байты/слова/дв.слова) – в зависимости от направления просмотра строки (“плюс” при **DF=0** и “минус” при **DF=1**).

Действие команды **stos**/**stosw**/**stosd**: **сохранение** элемента (байта/слова/дв.слова) из регистра **AL/AX/EAX** в текущую позицию строки-назначения, и автоматическая настройка на соседний элемент (в строке-назначении). Команда **не меняет флагов**!

1) **AL/AX/EAX => [EDI]**, флаги (**CF**, **OF**, **ZF**, **SF**) не меняются.

2) **EDI := EDI ± 1/2/4** (байты/слова/дв.слова) – в зависимости от направления просмотра строки (“плюс” при **DF=0** и “минус” при **DF=1**).

Вопрос_1

Есть ли смысл зацикливать команду **lods...** ?

Ответ: в этом нет никакого смысла.

Вопрос_2

Есть ли смысл зацикливать команду **stos...** ?

Ответ: да, есть смысл, для быстрого заполнения некоторой области (**ECX** раз) нужным значением: **rep stos...**

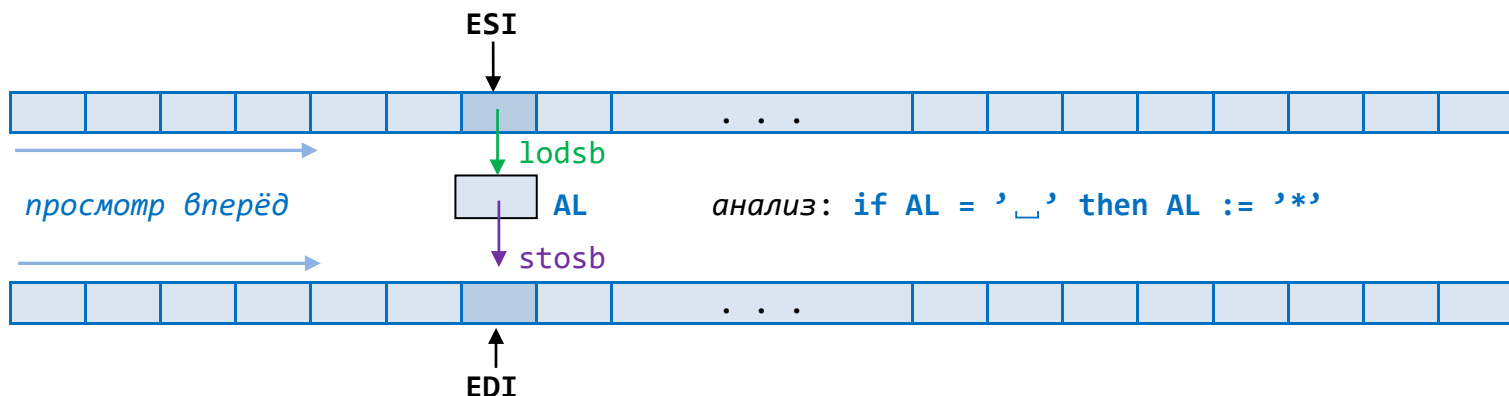
Пример 3

```
T db 100 dup(?)      ; T[0..99] of char
S db 100 dup(?)      ; S[0..99] of char
```

S и **T** – символьные строки в секции данных. Переписать содержимое строки **T** в строку **S** с заменой всех ПРОБЕЛОВ на символ **'*'**.

Решение.

1-я строка (откуда: **Source**)



2-я строка (куда: **Destination**)

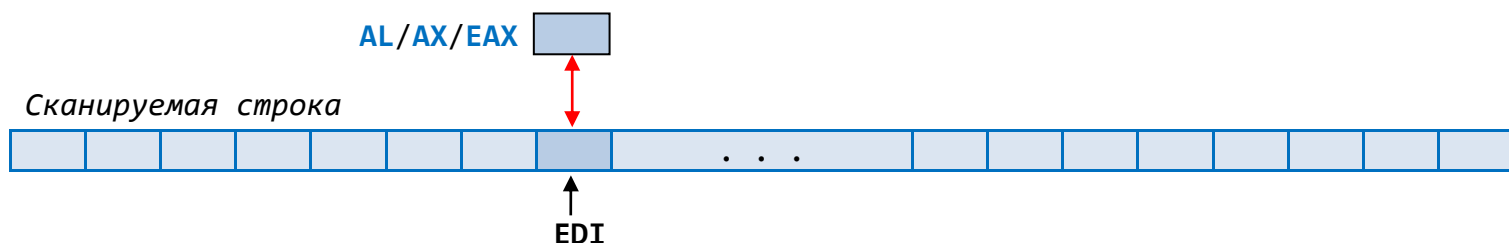
```
    lea ESI,T          ; откуда
    lea EDI,S          ; куда
    cld                ; просмотр вперед
    mov ECX,length T   ; длина (=100) пересылаемой строки
L:   lodsb              ; AL := [ESI], ESI := ESI + 1
     cmp AL,'_'
     jNE Skip
     mov AL,'*'
Skip: stosb              ; AL => [EDI], EDI := EDI + 1
     loop L
```

ПУНКТ 5 Сканирование строки из *байтов* (**scasb**), *слов* (**scasw**) и *двойных слов* (**scasd**)

Используется для организации поиска в строке.

Строка, в которой производится поиск (сканирование), рассматривается как *строка-назначение* (т.е. на неё следует настраивать регистр **EDI**).

До обращения к команде **scas...** следует загрузить в регистр **AL/AX/EAX** так называемый **“образец поиска”**, т.е. значение, с которым надо сравнивать элементы проверяемой строки.



Действие команды **scasb/scasw/scasd**: **сравнение** заданного в регистре **AL/AX/EAX** “**образца поиска**” с текущим элементом (байтом/словом/дв.словом) сканируемой строки, и автоматическая настройка **EDI** на соседний элемент (в сканируемой строке).

1) **AL/AX/EAX - [EDI]**, из содержимого регистра **AL/AX/EAX** вычитается текущий элемент сканируемой строки. Разность никуда не записывается, но по результату вычитания формируются флаги (**CF**, **OF**, **ZF**, **SF**).

2) **EDI := EDI ± 1/2/4** (байты/слова/дв.слова) – в зависимости от направления просмотра строки (“**плюс**” при **DF=0** и “**минус**” при **DF=1**).

Целесообразно использование префиксов повторений в паре с **scasb/scasw/scasd**:

repE/repZ scasb/scasw/scasd

обе мнемоники (**repE** и **repZ**) эквивалентны (т.е. транслируются в одну и ту же машинную команду)

Используются для поиска **первого элемента, отличного от заданного “образца”**.

Смысл: “крутись” в цикле (и повторяй сравнения), **пока** элементы строки **совпадают** со значением **AL/AX/EAX**

Исходы:

не нашли отличного от AL/AX/EAX элемента	нашли первый элемент, отличный от AL/AX/EAX
ZF=1, ECX=0	ZF=0, ECX - любое

repNE/repNZ scasb/scasw/scasd

обе мнемоники (**repNE** и **repNZ**) эквивалентны (т.е. транслируются в одну и ту же машинную команду)

Используются для поиска **первого элемента, совпадающего с заданным “образцом”**.

Смысл: “крутись” в цикле (и повторяй сравнения), **пока** элементы строки **отличны** от значения **AL/AX/EAX**

Исходы:

нашли первый элемент, совпадающий с AL/AX/EAX	не нашли ни одного элемента, совпадающего с AL/AX/EAX
ZF=1, ECX - любое	ZF=0, ECX=0

Вывод: для выяснения **причины выхода из цикла** следует интересоваться флагом нуля **ZF**

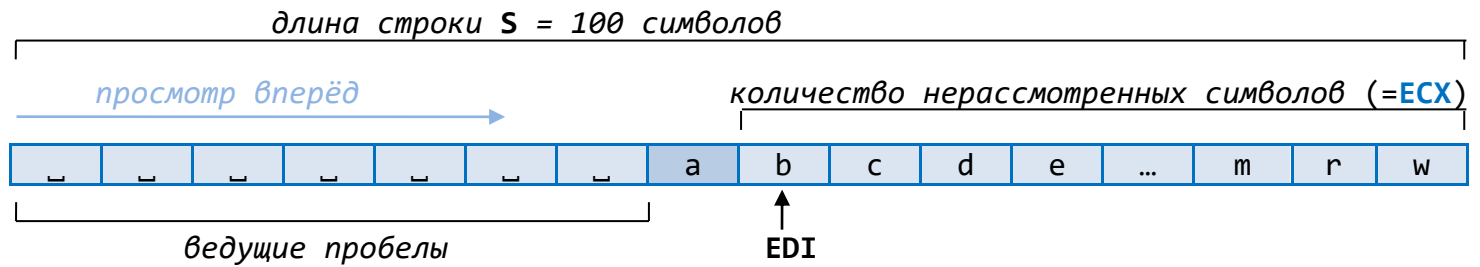
Пример 4

S db 100 dup(?) ; S[0..99] of char

S – символьная строка в секции данных. Определить, **со скольких пробелов начинается строка S**, ответ записать в регистр **CL**.

Решение.

Картина в строке **S** после обнаружения первого не пробела:



```

lea EDI,S      ; строка-назначение (сканируемая строка)
cld            ; просмотр вперёд
mov AL,'_'     ; "образец поиска"
mov ECX,100    ; длина проверяемой строки
repE scasb     ; продвигаемся вперед по строке, пока идут пробелы
; возможные исходы:
; ZF=1 (во всех позициях были пробелы, вышли по концу цикла, ECX=0)
; ZF=0 (попался первый не пробел, ECX м.б. любым)
jZ CL100       ; строка S полностью заполнена пробелами (100 пробелов!)
; сейчас ECX (значит и CL) хранит количество нерассмотренных символов
; => В начале строки было пробелов (см. рисунок выше) столько:
; 100 - (ECX+1) = 99 - ECX
neg CL
add CL,99      ; CL := 99-CL (искмое количество ведущих пробелов)
jmp Fin        ; обход установки CL для случая 100 пробелов в строке S
CL100:mov CL,100 ; строка S полностью заполнена пробелами
Fin:

```

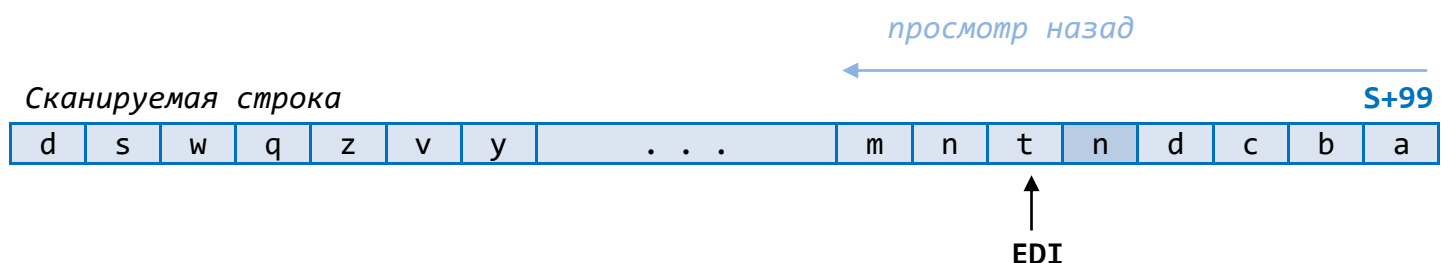
Пример 5

S db 100 dup(?) ; S[0..99] of char

S – символьная строка в секции данных. Заменить в строке **S** последнее вхождение буквы 'n' на букву 'N'.

Решение.

Картина после обнаружения первого (с конца) вхождения 'n':



Идея: запускаем просмотр от конца к началу.

```

lea EDI,S+99      ; настраиваемся на последний символ строки
std               ; просмотр назад (DF := 1)
mov AL,'n'        ; “образец поиска”
mov ECX,length S  ; максимальное число (=100) шагов цикла
repNE scasb       ; “беги” по строке, пока не встретишь 'n'
; возможные исходы:
; ZF=1 (нашли вхождение символа 'n', ECX – любое)
; ZF=0 (не было вхождений символа 'n', ECX=0)
jNZ Fin
; нашли, но EDI при этом автоматически сдвинулся к элементу левее найденного
; меняем найденное последнее вхождение символа 'n' на символ 'N':
mov byte ptr [EDI+1],'N'
Fin:

```

Пример 6

S db 100 dup(?) ; S[0..99] of char

S – символьная строка в секции данных. Подсчитать в строке **S** количество вхождений символа **'*'** и записать ответ в регистр **BH**.

Решение.

Не имеет принципиального значения, в каком направлении просматривать строку. Для определённости выберем просмотр строки *вперёд*.

Строка **S** *просмотр вперёд*

...	*	...	*	...	*	...	*	*	*	...	*	...	*
-----	---	-----	---	-----	---	-----	---	---	---	-----	---	-----	---

Идея. Сканируем строку до ближайшей **'*'**, учли. Сканируем строку до следующей **'*'**, учли. И так далее, пока не кончится строка.

```

lea EDI,S        ; настраиваемся на начало проверяемой строки S
cld              ; просмотр вперед
mov AL,'*'       ; “образец поиска”
mov ECX, length S ; длина (=100) строки
xor BH,BH        ; счётчик
L: jECXz skip     ; чтобы не было закливания при ECX=0
repNE scasb      ; двигаемся вперёд, пока не '*'
; возможные исходы:
; ZF=1 (нашли очередную '*', ECX – любое)
; ZF=0 (дошли до конца строки и больше не встретили символа '*')
; на последнем отрезке поиска, ECX=0)
jNE skip         ; для ZF=0
inc BH          ; для ZF=1
; осталось аналогично просмотреть ECX символов

```

```

; (для случая ECX=0 при переходе на метку L предусмотрена
; специальная проверка)
jmp L

```

skip:

ПУНКТ 6 Представление строк переменной длины

В рассмотренных выше примерах велась работа со строками **фиксированной длины**. Число символов в них *не менялось*. Т.е. такие строки рассматривались как *обычные массивы из байтов (=символов)*.

В реальных же задачах бóльший интерес представляют строки **переменной длины**: число символов в них жёстко не зафиксировано и *может меняться*. При представлении таких строк в памяти заранее оцениваем максимально возможную длину строки и резервируем соответствующее количество байтов с расчетом на этот максимум.

Пусть известна максимально возможная длина строки.

Возникает вопрос: *как представить строку в памяти, чтобы было понятно, сколько и какие символы в ней имеются?*

Известны **два основных способа** представления строк *переменной длины*:

1 способ (представление с “**длиной**” – тип `string` в Turbo Pascal или в Free Pascal с установленной опцией `{ $mode TP }` компилятору);

2 способ (представление с “**маркером конца**” – в языке Си).

Рассмотрим особенности каждого из этих способов.

1 способ (представление с “**длиной**”):

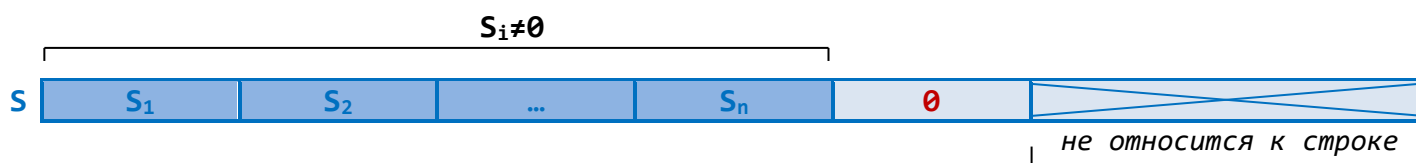


В **начальном байте** хранится **текущая длина** строки. Сами же символы, образующие строку, задаются вслед за длиной строки. Всё, что находится дальше последнего символа – к строке *не относится*.

Если для хранения длины строки отвести *один байт*, то максимальная длина строки составит **255 байтов** (=символов):

`S db 256 dup(?)` ; в начальном байте задаётся *текущая* длина строки

Особенность: при работе с таким представлением нужно обязательно корректировать длину, если изменяется число символов в строке.

2 способ (представление с “маркером конца”):

Выбирается некоторый символ (например, с кодом **0**) в качестве **маркера конца строки** (при этом заранее договариваются, что этот символ не встретится среди символов S_i строки). Всё, что находится за этим спецсимволом – *к строке не относится*.

Особенность: при изменении длины строки *меняется положение спецсимвола* (он сдвигается).

Неудобства: 1) Чтобы *узнать длину строки*, нужно пройти её до конца; 2) *Проверка на равенство двух строк*: если они разной длины, то заведомо не равны. Но чтобы это установить, придётся выполнить много действий.

ПУНКТ 6.1 Примеры задач на обработку строк, использующих представление с “длиной”

Пример 7

S db 256 dup(?) ; длина $S \leq 255$

S – символьная строка переменной длины (с текущей длиной в начальном байте), размещённая в секции данных. *Оставить* в строке **S** *только первые 10 символов* (если они есть); если символов меньше десяти – строку не менять.

Решение.

```
cmp S,10      ; проверяем текущую длину строки (содержимое начального байта)
jBE Fin       ; если  $n \leq 10$ , то строку не трогаем
; (здесь через n обозначена текущая длина строки)
; если  $n > 10$ , то достаточно изменить содержимое начального байта:
mov S,10
```

Fin:

Пример 8

S db 256 dup(?) ; длина $S \leq 255$

S – символьная строка переменной длины (с текущей длиной в начальном байте), размещённая в секции данных. Сделать значением **S** строку из *50 пробелов*.

Решение.

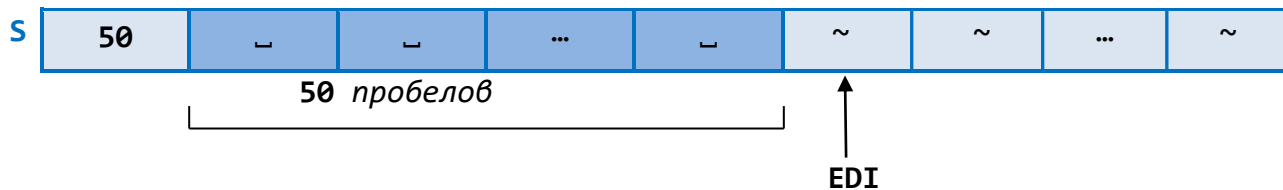
```
mov S,50      ; задаём длину строки
lea EDI,S+1    ; строка-назначение (для команды stosb)
```

```

cld          ; просмотр вперёд
mov AL,' '   ; для заполнения пробелом
mov ECX,50   ; число повторений
rep stosb

```

В результате таких действий получим:

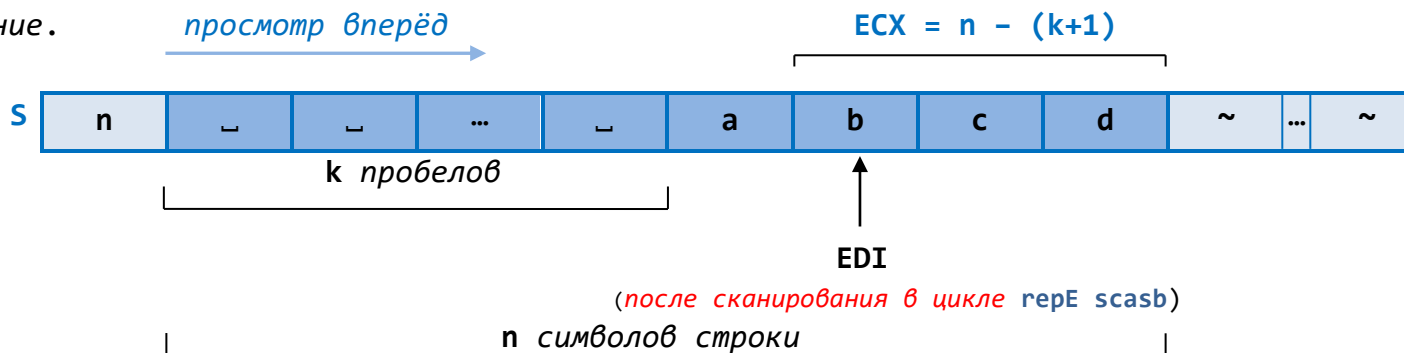


Пример 9

`S db 256 dup(?)` ; длина $S \leq 255$

S – символьная строка переменной длины (с текущей длиной в начальном байте), размещённая в секции данных. Удалить все пробелы **в начале** строки **S**.

Решение.



```

movzx ECX,S      ; ECX := dword(n), где n – обозначает текущую длину строки
jECXz Fin        ; если строка пустая -> на конец фрагмента
cld              ; просмотр вперёд
lea EDI,S+1      ; сканируемая строка (настройка EDI на её начало)
mov AL,' '       ; “образец поиска”
repE scasb       ; “крутись” в цикле, пока идут пробелы
; возможные исходы:
; ZF=1 (строка состоит только из пробелов, ECX=0)
; ZF=0 (нашли первый непобел, ECX - любое)
jZ set_n
; проверка: были ли ведущие пробелы?
cmp EDI,offset S+2
jE fin           ; пробелов в начале S не было => ничего сдвигать не надо
lea ESI,[EDI-1] ; откуда
lea EDI,S+1      ; куда
inc ECX          ; число пересылаемых символов (см. рисунок выше)
set_n:mov S,CL   ; новая длина строки
rep movsb        ; при ECX=0 цикл не выполнится (т.к. по правилам использования
                  ; префиксов повторения этот цикл реализуется с предусловием)
Fin:

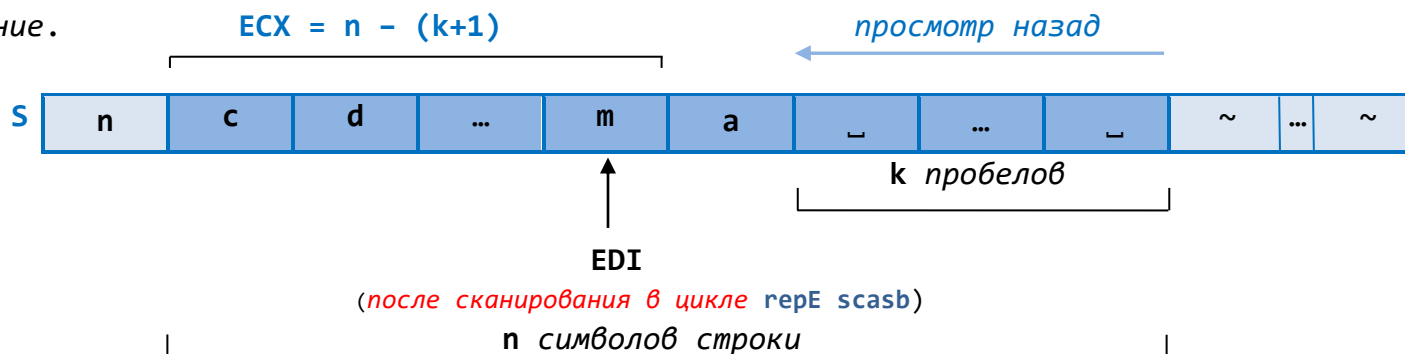
```

Пример 10

S db 256 dup(?) ; длина $S \leq 255$

S – символьная строка переменной длины (с текущей длиной в начальном байте), размещённая в секции данных. Удалить все пробелы **в конце** строки **S**.

Решение.



```

movzx ECX,S      ; ECX := dword(n), где n – обозначает текущую длину строки
jECXz Fin       ; если строка пустая -> на конец фрагмента
std             ; просмотр назад
lea EDI,S[ECX]  ; сканируемая строка (настроились на последний символ строки)
mov AL,' '      ; “образец поиска”
repE scasb      ; “крутись” в цикле, пока идут пробелы
; возможные исходы:
; ZF=1 (строка состоит только из пробелов, ECX=0)
; ZF=0 (нашли первый непобел, ECX – любое)
jZ Set_n
; сейчас в ECX – количество нерассмотренных символов (но следует также учесть
; и найденный непобел) => величина ECX+1 соответствует новой длине строки
inc CL          ; новая длина строки (заметим, что этот ответ получится верным
; и для случая, когда пробелов в конце строки не оказалось)
Set_n:mov S,CL  ; новая длина строки
Fin:

```

Пример 11

S db 256 dup(?) ; длина $S \leq 255$

S – символьная строка переменной длины (с текущей длиной в начальном байте), размещённая в секции данных. Если в строке **S** от **10** до **40** символов, то продублировать **10**-й символ.

Решение.

```

cmp S,10
jB Fin
cmp S,40
jA Fin

```

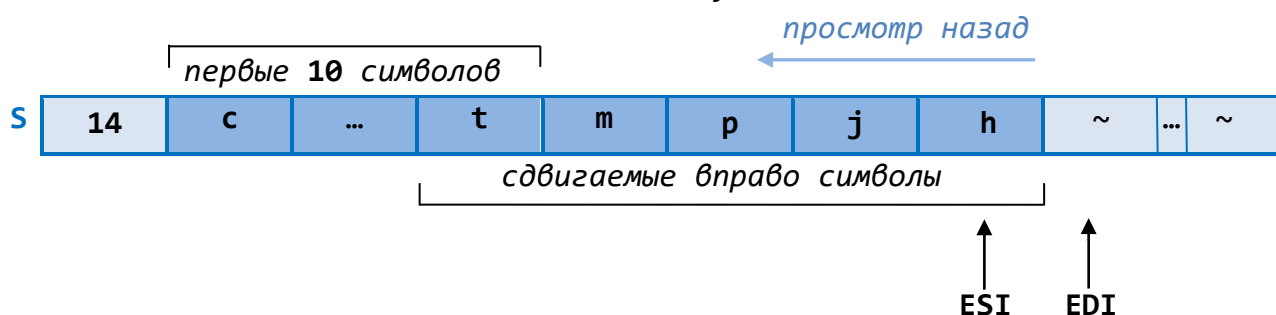
```

; длина строки S лежит в диапазоне [10..40] => дублируем её 10-ый символ
std                ; просмотр назад (важно это осознать!)
movzx ECX,S        ; ECX := dword(n), где n – обозначает текущую длину строки
lea ESI,S[ECX]     ; откуда
lea EDI,S[ECX+1]   ; куда
sub ECX,9          ; количество копируемых символов
rep movsb          ; сдвигаем символы S10 ... Sn на одну позицию вправо,
                  ; начиная с символа Sn и заканчивая символом S10
inc S              ; новая длина строки
Fin:

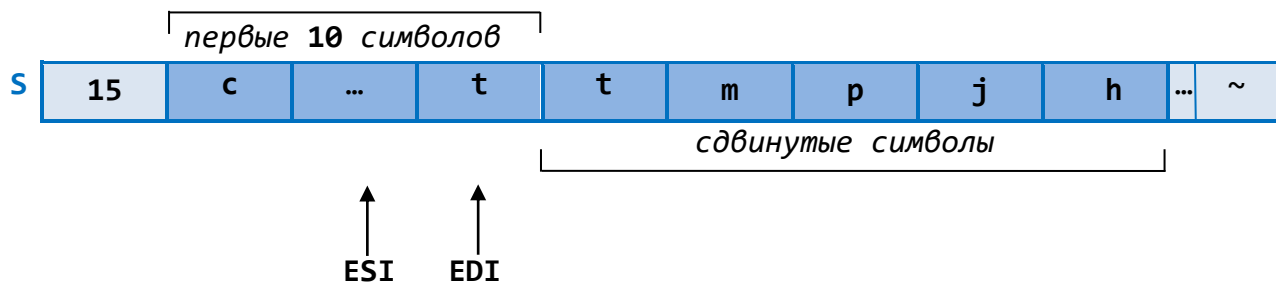
```

Иллюстрация к решению (для случая n=14):

Подготовка к сдвигу:



После сдвига:



Пример 12

S db 256 dup(?) ; длина $S \leq 255$

S – символьная строка переменной длины (с текущей длиной в начальном байте), размещённая в секции данных. Удалить из строки **S** все пробелы.

Решение.

```

movzx ECX,S        ; ECX := dword(n), где n – обозначает текущую длину строки
jECXz Fin          ; если строка пустая -> на конец фрагмента
cld                ; просмотр вперёд
lea ESI,S[1]       ; откуда
mov EDI,ESI        ; куда
L: lodsb           ; AL := [ESI] , ESI := ESI+1
cmp AL, ' '

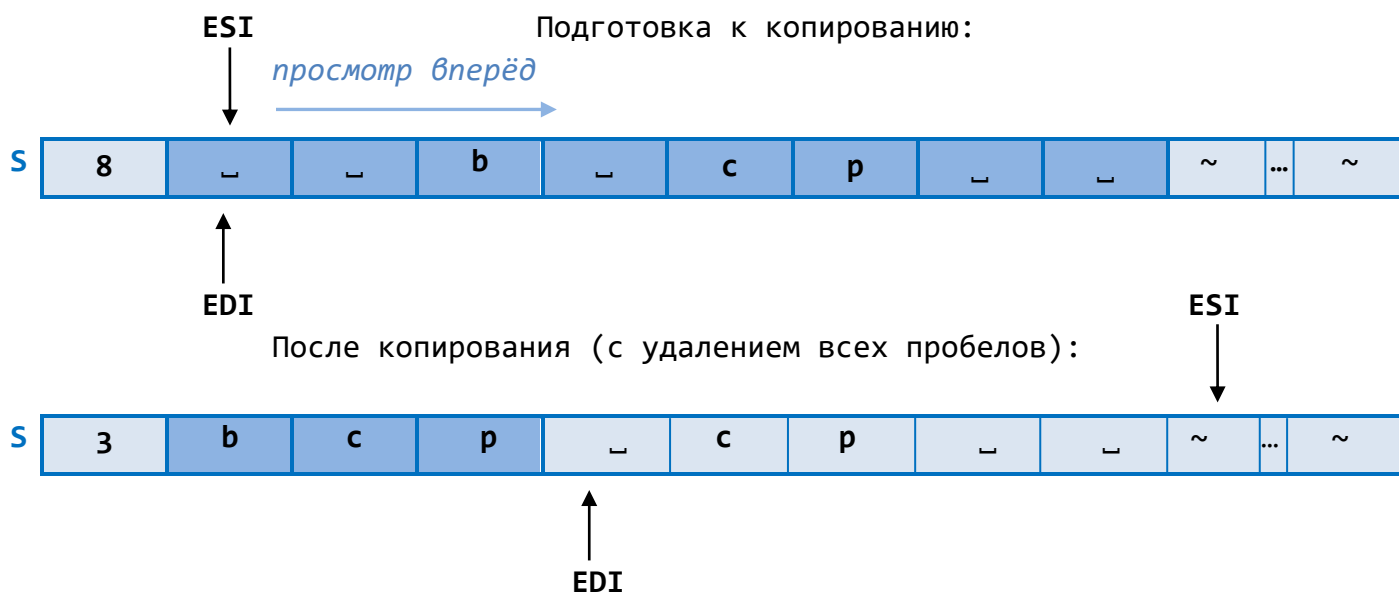
```

```

jE Skip      ; пробел игнорируем, “выбрасываем” его из строки
; не пробел переносим на очередную свободную позицию:
stosb        ; AL => [EDI] , EDI := EDI+1
jmp Next
Skip: dec S   ; if AL='_' then n := n-1
Next: loop L
Fin:         ; если в строке пробелов не было, то согласно приведённому алгоритму
; строка будет скопирована на саму себя (т.е. останется без изменений)

```

Иллюстрация к решению (для случая n=8):



ПУНКТ 6.2 Примеры задач на обработку строк, использующих представление с “маркером конца”

Пример 13

K equ 100000

S db K dup(?) ; память (K байтов) под представление строки

n dd ?

S – символьная строка переменной длины с “маркером конца” (нулевым байтом), размещённая в секции данных. Записать в переменную n длину строки S.

Решение.

```

cld      ; просмотр вперёд
xor AL,AL ; AL:=0 (“образец поиска”)
lea EDI,S ; сканируемая строка
mov ECX,K ; максимальное число итераций
repNE scasb ; “крутись” в цикле (максимум K раз), пока не встретишь 0
neg ECX
add ECX,K-1
mov n,ECX ; n := K-(ECX+1) (см. ниже иллюстрацию к решению)
; замечание: для пустой строки алгоритм даст корректный ответ

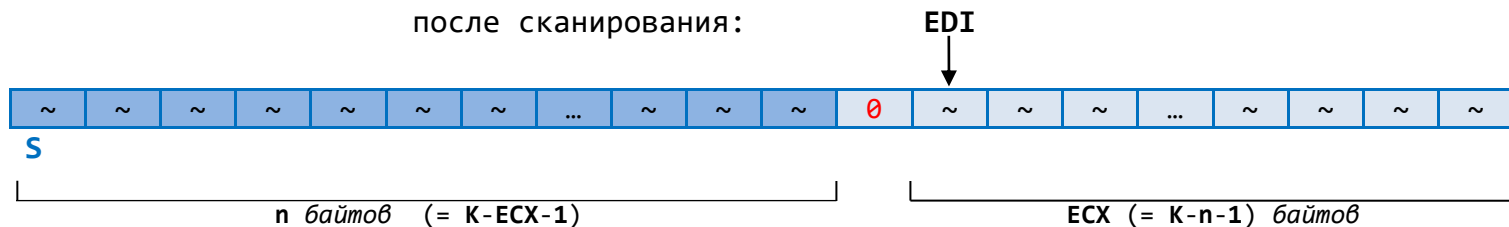
```

Иллюстрация к решению

до начала сканирования:



после сканирования:



Пример 14

K equ 100000

S db K dup(?) ; память (K байтов) под представление строки

S – символьная строка переменной длины с “маркером конца” (нулевым байтом), размещённая в секции данных. Сделать значением S строку из 50 пробелов.

Решение.

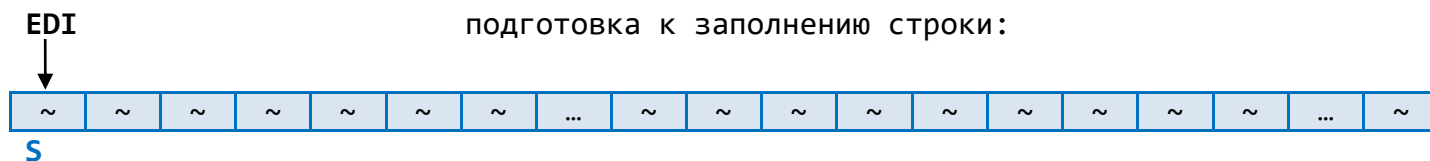
```

cld          ; просмотр вперёд
mov AL,' '   ; подготовка значения для заполнения им байтов строки
mov ECX,50   ; всего – 50 байтов
lea EDI,S    ; строка-назначение
rep stosb    ; 50 раз выгружаем пробел из AL в очередной байт
; завершаем строку “маркером конца” (байтом 0):
mov byte ptr [EDI], 0

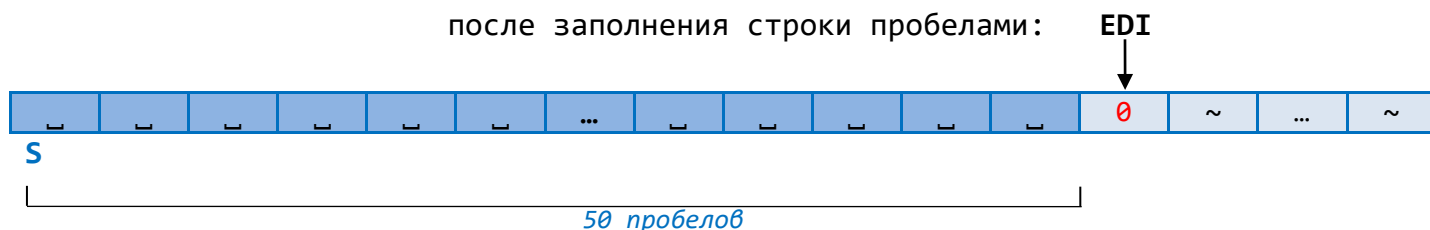
```

Иллюстрация к решению

подготовка к заполнению строки:



после заполнения строки пробелами:



Пример 15

K equ 100000

S db K dup(?) ; память (**K** байтов) под представление строки
S – символьная строка переменной длины с “маркером конца” (нулевым байтом), размещённая в секции данных. Удалить все пробелы **в начале** строки **S**.

Решение.

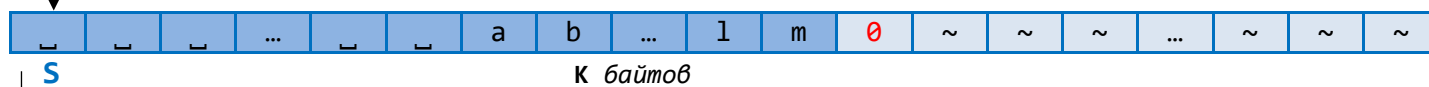
```

    cmp S,0
    jZ Fin          ; если строка пустая (n=0), то на конец фрагмента
    cld             ; просмотр вперёд
    mov ECX,K       ; максимальное число итераций
    mov AL,' '      ; “образец поиска”
    lea EDI, S      ; сканируемая строка
repE scasb          ; “крутись” в цикле, пока идут пробелы
    ; нашли первый не пробел (EDI указывает на байт вслед за ним),
    ; готовимся к копированию “хвоста” строки в начало:
    lea ESI,[EDI-1] ; первый не пробел (откуда)
    lea EDI,S       ; начало строки (куда)
    cmp ESI,EDI
    jE Fin          ; в строке нет пробелов – ничего не делаем
L:  lodsb           ; [ESI] → AL, ESI := ESI + 1
    stosb           ; AL → [EDI], EDI := EDI + 1
    cmp AL,0        ; если выгрузили ноль то выходим из цикла
    jNE L

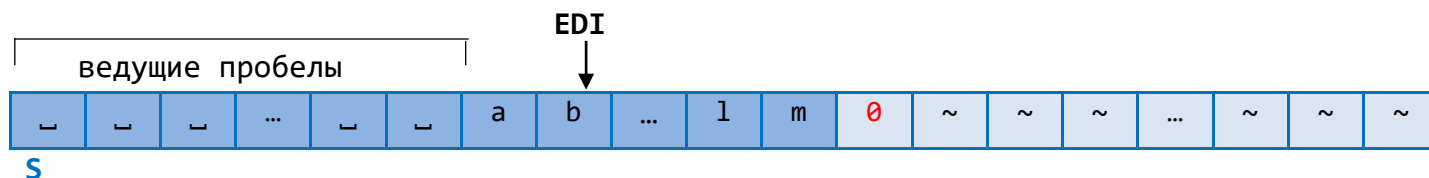
```

Fin: *Иллюстрации к решению*

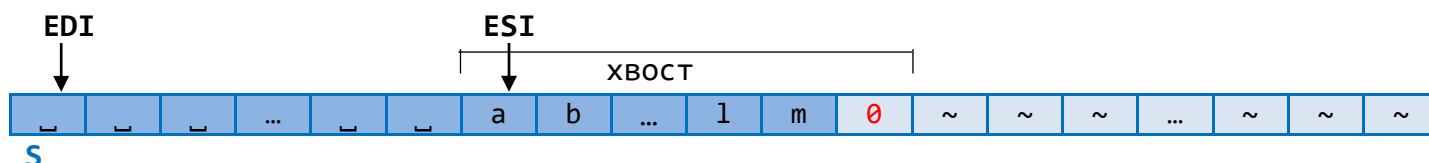
EDI подготовка к сканированию строки:



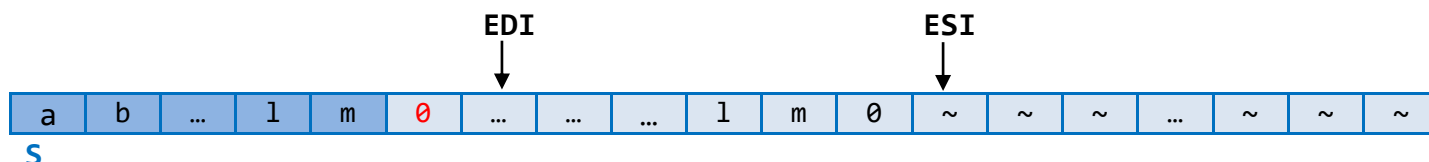
после сканирования (найден первый не пробел):



подготовка к копированию “хвоста” в начало:



после удаления ведущих пробелов:



Пример 16

K equ 100000

S db K dup(?) ; память (**K** байтов) под представление строки

S – символьная строка переменной длины с “маркером конца” (нулевым байтом), размещённая в секции данных. Удалить все пробелы **в конце** строки **S**.

Решение.

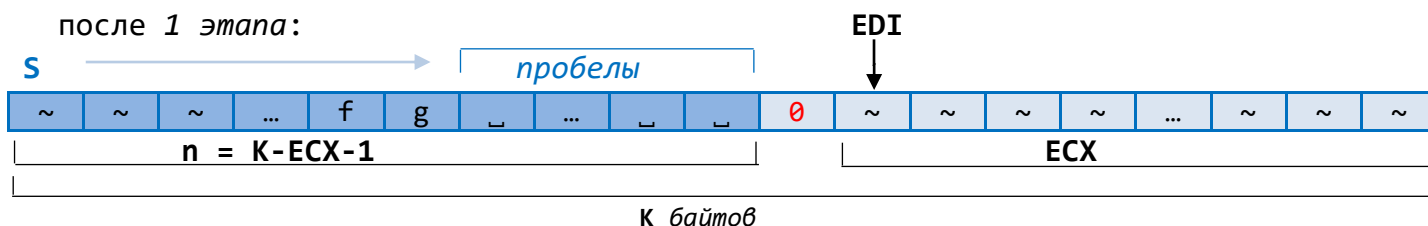
```

cmp S,0
jz Fin          ; если строка пустая (n=0), то на конец фрагмента
; 1 этап (поиск конца строки):
cld            ; просмотр вперёд
mov ECX,K      ; максимальное число итераций
xor AL,AL      ; “образец поиска” – байт 0
lea EDI,S      ; сканируемая строка
repNE scasb    ; “крутись” в цикле, пока не ноль (0 встретится обязательно)
; сейчас в EDI – адрес ячейки за “маркером конца” (за байтом 0)
; в ECX – количество нерассмотренных символов
; 2 этап (движение по строке от конца к началу в поиске первого не пробела):
lea EDI,[EDI-2] ; настройка на последний символ в строке (символ левее 0)
std           ; просмотр назад
; вычисляем максимальное число итераций (= K-ECX-1) при движении назад:
neg ECX
add ECX,K-1
mov AL,'_ '    ; “образец поиска”
repE scasb     ; “крутись” в цикле, пока пробел
; возможные исходы:
; все символы строки – пробелы: ZF=1, ECX=0 (EDI – на байт слева от S)
; нашли первый не пробел: ZF=0, ECX – любое (EDI – на байт левее не пробела)
jz Zero
inc EDI        ; корректировка для случая обнаружения первого не пробела
; (если в конце строки не было пробелов, то, согласно
; алгоритму, ноль будет записан на своё прежнее место)
Zero: ; 3 этап (генерируем “маркер конца”):
mov byte ptr [EDI+1],0
Fin:

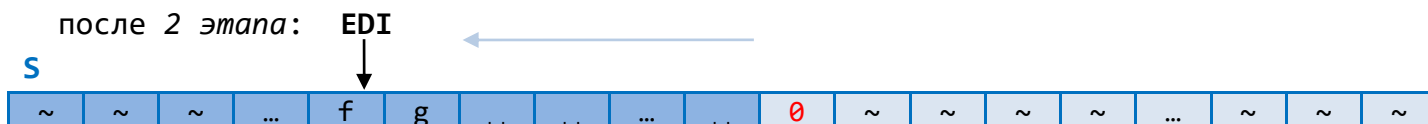
```

Иллюстрации к решению (для случая наличия пробелов в конце строки)

после 1 этапа:



после 2 этапа:



после 3 этапа:

EDI



Пример 17

K equ 100000

S db K dup(?) ; память (K байтов) под представление строки

S – символьная строка переменной длины с “маркером конца” (нулевым байтом), размещённая в секции данных. Если в строке **S** от 10 до 40 символов, то продублировать 10-й символ.

Решение.

```
cld                ; просмотр вперед
lea EDI,S          ; сканируемая строка
mov ECX,K          ; максимальное число итераций
xor AL,AL          ; “образец поиска” (байт 0)
```

repNE scasb

; “крутись” в цикле, пока не ноль (0 обязательно встретится)
 ; сейчас в **EDI** – адрес ячейки за “маркером конца” (за байтом 0)
 ; в **ECX** – кол-во нерассмотренных символов: K-(n+1), где n длина этой строки

```
cmp ECX,K-(10+1)
```

```
jA Fin
```

```
cmp ECX,K-(40+1)
```

```
jB Fin
```

; если строка короче 10 или длиннее 40, то на конец фрагмента

; сдвигаем вправо “хвост” строки (начиная с 10-го символа и кончая байтом 0)

; вычисляем (в **ECX**) длину сдвигаемого вправо “хвоста”

; (см. пояснения по дальнейшему решению на иллюстрации ниже)

```
lea EAX,S+9        ; EAX := адрес 10-го символа
```

```
neg EAX
```

```
lea ECX,[EDI+EAX]
```

```
std                ; просмотр назад (важно !)
```

```
lea ESI,[EDI-1]; откуда (начиная с байта 0)
```

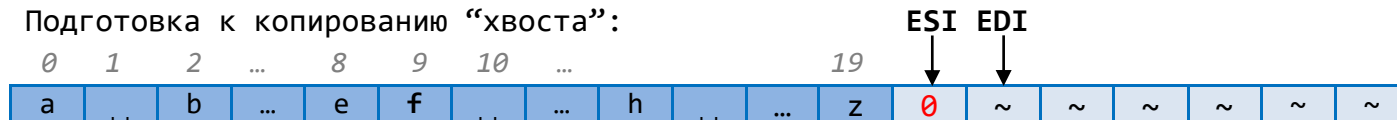
; **EDI** – куда (уже готово!)

```
rep movsb          ; копирование “хвоста” строки на 1 позицию вправо
```

Fin:

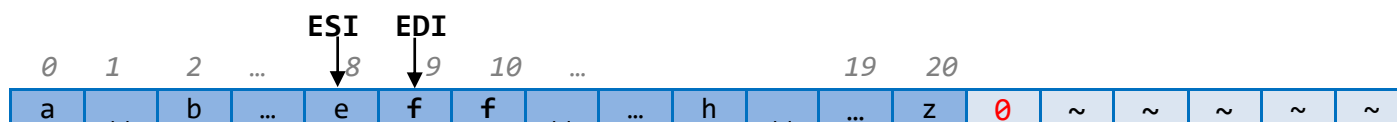
Иллюстрации к решению (для случая строки длины 20)

Подготовка к копированию “хвоста”:



S длина “хвоста”: ECX = EDI - (offset S + 9)

После копирования:



S

Пример 18

K equ 100000

S db K dup(?) ; память (**K** байтов) под представление строки

S – символьная строка переменной длины с “маркером конца” (нулевым байтом), размещённая в секции данных. Удалить из строки **S** все пробелы.

Решение.

```

    cld                ; просмотр вперед
    lea ESI,S          ; откуда
    lea EDI,S          ; куда
L : lodsb              ; [ESI] → AL, ESI := ESI+1
    cmp AL,' '         ; проверка на пробел
    jE L               ; если пробел, то его игнорируем
    stosb              ; если не пробел, то AL → [EDI], EDI := EDI+1
    cmp AL,0           ; проверка на конец строки
    jNE L              ; если не ноль, то продолжаем работать в цикле

```

Иллюстрации к решению

Подготовка к обработке строки:

ESI (откуда)



EDI (куда)

После удаления из строки всех пробелов:

