Московский государственный технический университет
имени Н.Э. Баумана

Факультет «Информатика и вычислительная техника»

Кафедра «Компьютерные системы и сети»

Г.С. Иванова, Т.Н. Ничушкина

Ассемблер IA32:
обработка символьной информации

Методические указания по выполнению домашнего задания № 1
по дисциплине «Машинно-зависимые языки и основы компиляции»

Москва

УДК 004.432

Иванова Г.С., Ничушкина Т.Н.

Ассемблер IA32: обработка символьной информации. Методические указания по выполнению домашнего задания № 1 по дисциплине «Машинно-зависимые языки и основы компиляции». - М.: МГТУ имени Н.Э. Баумана, 2022. 26 с.

Определена цель и объем домашнего задания № 1 по дисциплине «Машинно-зависимые языки и основы компиляции». Представлен теоретический материал, необходимый для разработки программы домашнего задания. Приведены примеры программ, аналогичных разрабатываемым во время выполнения домашнего задания. Описан порядок выполнения задания, приведены варианты заданий, определены требования к отчету и дан примерный список контрольных вопросов для защиты домашнего задания.

Для студентов МГТУ имени Н.Э. Баумана, обучающихся по программам бакалавриата направлений «Информатика и вычислительная техника» и «Прикладная информатика».

Иванова Галина Сергеевна

Ничушкина Татьяна Николаевна

Ассемблер IA32: обработка символьной информации.

Методические указания по выполнению домашнего задания № 1 по дисциплине Машинно-зависимые языки и основы компиляции

Оглавление

Цель работы: изучение команд обработки цепочек и приемов обработки символьной информации

Объем работы: 6 часов

Теоретическая часть

Команды обработки цепочек также называют командами *обработки строк символов*. Отличие в том, что под *строкой символов* понимается последовательность байт, а *цепочка* — это более общее название для случаев, когда элементы последовательности имеют размер больше байта — слово или двойное слово.

Таким образом, цепочечные команды позволяют проводить действия над блоками памяти, представляющими собой последовательности элементов следующих размеров:

- 8 бит — байт;
- 16 бит — слово;
- 32 бита — двойное слово.

Содержимое этих блоков для процессора не имеет никакого значения. Это могут быть символы, числа и все что угодно. Главное, чтобы размерность элементов совпадала с одной из перечисленных и эти элементы находились в соседних ячейках памяти.

Всего в системе команд процессора имеется пять *операций-примитивов* обработки цепочек. Каждая из них реализуется в процессоре тремя командами. Каждая из этих команд работает с соответствующим размером элемента — байтом, словом или двойным словом.

Особенность всех цепочечных команд в том, что они, кроме обработки текущего элемента цепочки, осуществляют еще и *автоматическое продвижение указателя адреса* к следующему элементу цепочки.

Ниже приведены основные операции-примитивы и команды, с помощью которых они реализуются:

- *пересылка цепочки*:

```
movs Адрес_приемника,Адрес_источника
movsb
movsw
movsd
```

- *сравнение цепочек*:

  cmps Адрес_приемника,Адрес_источника

  cmpsb

  cmpsw

  cmpsd

- *сканирование цепочки*:

  scas Адрес_приемника

  scasb

  scasw

  scasd

- *загрузка элемента из цепочки:*

  lods Адрес_источника

  lodsb

  lodsw

  lodsd

- *сохранение элемента в цепочке:*

  stos Адрес_приемника

  stosb

  stows

  stosd

Логически к этим командам нужно отнести и, так называемые, *префиксы повторения*. В соответствии с форматом машинной команды, первые необязательные байты формата - префиксы. Один из возможных типов префиксов — это *префиксы повторения*. Они предназначены для использования цепочечными командами.

Префиксы повторения имеют свои мнемонические обозначения:

rep

repe или **repz**

repne или **repnz**

Эти префиксы повторения указывают перед нужной цепочечной командой. Цепочечная команда без префикса выполняется один раз. Размещение префикса перед цепочечной командой заставляет ее выполняться в цикле.

Отличия приведенных префиксов в том, на каком основании принимается решение о циклическом выполнении цепочечной команды:

- *по состоянию регистра* **ecx/cx**
- *по флагу нуля ZF.*

Префикс повторения **rep** (REPeat). Этот префикс используется с командами, реализующими операции-примитивы пересылки и сохранения элементов цепочек — соответственно, **movs** и **stos**.

Префикс **rep** заставляет данные команды выполняться, пока *содержимое* **ecx/cx** *не станет равным 0*. При этом цепочечная команда, перед которой стоит префикс, *автоматически уменьшает содержимое* **ecx/cx** *на единицу*.

Префиксы повторения **repe** **или** **repz** (REPeat while Equal or Zero). Эти префиксы являются абсолютными синонимами.

Они заставляют цепочечную команду выполняться до тех пор, пока *содержимое* **ecx/cx** *не равно нулю или флаг ZF равен 1*. Как только одно из этих условий нарушается, управление передается следующей команде программы. Благодаря возможности анализа флага *ZF*, наиболее эффективно эти префиксы можно использовать с командами **cmps** и **scas** для поиска отличающихся элементов цепочек.

Префиксы повторения **repne** **или** **repnz** (REPeat while Not Equal or Zero). Эти префиксы также являются абсолютными синонимами. Их действие на цепочечную команду несколько отличается от действий префиксов repe/repz Префиксы **repne/repnz** заставляют цепочечную команду циклически выполняться до тех пор, пока *содержимое ecx/cx не равно нулю или флаг ZF равен нулю*.

При невыполнении одного из этих условий работа команды прекращается. Данные префиксы также можно использовать с командами **cmps** и **scas**, но для поиска совпадающих элементов цепочек.

Следующий важный момент, связанный с цепочечными командами, заключается в *особенностях формирования физического адреса* операндов Адрес_источника и Адрес_приемника.

Цепочка-источник, адресуемая операндом *Адрес_источника*, ***может*** находиться в текущем сегменте данных, определяемом регистром **ds**. **Цепочка-приемник**, адресуемая операндом *Адрес_приемника*, ***должна*** быть в дополнительном сегменте данных, адресуемом сегментным регистром **es**. (Указанные сегменты могут быть наложены один на другой, т.е. физически могут находиться по одному адресу). Важно отметить, что допускается *замена* (с помощью префикса замены сегмента) только регистра **ds**, регистр **es** подменять нельзя.

Вторые части адресов – *смещения цепочек* — также должны находиться в строго определенных местах. Для цепочки-источника это регистр **esi/si** (Source Index register — индексный регистр источника). Для цепочки-получателя это регистр **edi/di** (Destination Index register - индексный регистр приемника).

Таким образом, полные физические адреса для операндов цепочечных команд следующие:

- Адрес_источника — регистры **ds:esi/si**;
- Адрес_приемника — регисты **es:edi/di**.

Нетрудно заметить, что все группы команд, реализующих цепочечные операции-примитивы, имеют похожий по структуре набор команд. В каждом из этих наборов присутствует одна команда с явным указанием операндов и три команды, не имеющие операндов.

На самом деле, набор команд процессора имеет соответствующие машинные команды только для цепочечных команд ассемблера без операндов. Команды с операндами транслятор ассемблера использует только для определения типов операндов и *то не во всех трансляторах***.** После того как выяснен тип элементов цепочек по их описанию в памяти, генерируется одна из трех машинных команд для каждой из цепочечных операций. По этой причине все регистры, содержащие адреса цепочек, должны быть инициализированы заранее, в том числе и для команд, допускающих явное указание операндов.

В силу того, что цепочки адресуются однозначно, нет особого смысла применять команды с операндами. Главное, что следует запомнить, — *правильная загрузка регистров адресами обязательно требуется до выдачи любой цепочечной команды*.

Важный момент, касающийся всех цепочечных команд, — это *направление обработки цепочки*. Есть две возможности:

- от начала цепочки к ее концу, то есть в направлении возрастания адресов;

- от конца цепочки к началу, то есть в направлении убывания адресов.

Следует отметить, что цепочечные команды сами выполняют модификацию регистров, адресующих операнды, обеспечивая тем самым автоматическое продвижение по цепочке. Количество байтов, на которые эта модификация осуществляется, определяется кодом команды. А вот знак этой модификации определяется значением флага направления *DF* (Direction Flag) в регистре **eflags/flags**:

- если *DF* = 0, то значение индексных регистров **esi/si** и **edi/di** будет автоматически увеличиваться (операция инкремента) цепочечными командами, то есть обработка будет осуществляться в направлении возрастания адресов;

- если *DF* = 1, то значение индексных регистров **esi/si** и **edi/di** будет автоматически уменьшаться (операция декремента) цепочечными командами, то есть обработка будет идти в направлении убывания адресов.

Состоянием флага *DF* можно управлять с помощью двух команд, не имеющих операндов:

cld (Clear Direction Flag) — очистить флаг направления. Команда сбрасывает флаг направления *DF* в 0.

std (Set Direction Flag) — установить флаг направления. Команда устанавливает флаг направления *DF* в 1.

Далее более подробно рассмотрены все операции и команды, которые ее реализуют.

1.1. Операция пересылки цепочек

Команды, реализующие эту операцию-примитив, производят копирование элементов из одной области памяти (цепочки) в другую. Размер элемента определяется применяемой командой.

movs Адрес_приемника,Адрес_источника

Команда копирует байт, слово или двойное слово из цепочки, адресуемой операндом *Адрес_источника*, в цепочку, адресуемую операндом *Адрес_приемника.*

Размер пересылаемых элементов ассемблер определяет, исходя из атрибутов идентификаторов, указывающих на области памяти приемника и источника. К примеру, если эти идентификаторы были определены директивой **db**

(**resb**), то пересылаться будут байты, если идентификаторы были определены с помощью директивы **dd** (**resd**), то пересылке подлежат 32-битовые элементы, то есть двойные слова.

Ранее уже было отмечено, что для цепочечных команд с операндами, к которым относится и команда пересылки

movs Адрес_приемника,Адрес_источника, не существует машинного аналога.

При трансляции в зависимости от типа операндов транслятор преобразует ее в одну из трех машинных команд:

movsb, movsw или **movsd**.

Сама по себе команда **movs** пересылает только один элемент, исходя из его типа, и модифицирует значения регистров **esi/si** и **edi/di**. Если перед командой написать префикс **rep**, то одной командой можно переслать до 4 Гбайт данных.

Число пересылаемых элементов должно быть загружено в *счетчик — регистр* **cx**/**ecx**.

Ниже перечислен набор действий, которые нужно выполнить в программе для того, чтобы выполнить пересылку последовательности элементов из одной области памяти в другую с помощью команды **movs**.
В общем случае этот набор действий можно рассматривать как типовой для выполнения любой цепочечной команды:

- Установить значение флага *DF* в зависимости от того, в каком направлении будут обрабатываться элементы цепочки — в направлении возрастания или убывания адресов.
- Загрузить указатели на адреса цепочек в памяти в пары регистров **ds:esi** и **es:edi**.
- Загрузить в регистр **ecx/cx** количество элементов, подлежащих обработке.
- Выдать команду **movs** с префиксом **rep**.

Пример 1. Написать программу пересылки строки символов из строки *Source* в строку *Result*.

```
        .data
Source db   'Primer of move string' ; Исходная строка
        .bss
```

```
Result resb  21                          ;Строка результат
       .text
_start:
       cld          ; сброс флага DF - обработка строки от начала к концу
       lea     esi,[Source]     ; загрузка в si смещения строки-источника
       lea     edi,[Result]     ; загрузка в DS смещения строки-приёмника
       mov     ecx,21   ; для префикса rep — счетчик повторений (длина строки)
     rep   movsb    ;пересылка строки источника в строку приемник
      ...
```

1.2. Операция сравнения цепочек

Команды, реализующие эту операцию, выполняют сравнение элементов цепочки-источника с элементами цепочки-приемника.

Синтаксис команды cmps:

`cmps` Адрес_приемника,Адрес_источника

где:

- *Адрес_источника* определяет **цепочку-источник** в сегменте данных. Адрес цепочки должен быть заранее загружен в пару регистров **`ds:esi/si`**;
- *Адрес_приемника* определяет **цепочку-приемник**. Цепочка должна находиться в дополнительном сегменте, и ее адрес должен быть заранее загружен в пару **`es:edi/di`**.

Алгоритм работы команды cmps заключается в последовательном выполнении вычитания (элемент цепочки-источника – элемент цепочки-приемника) над очередными элементами обеих цепочек.

Принцип выполнения вычитания командой cmps аналогичен команде сравнения **`cmp`**. Она, так же, как и **`cmp`**, производит вычитание элементов, не записывая при этом результата, и устанавливает флаги *ZF*, *SF* и *OF*.

После выполнения вычитания очередных элементов цепочек командой cmps, индексные регистры **`esi/si`** и **`edi/di`** *автоматически изменяются в соответствии со значением флага DF на значение, равное размеру элемента сравниваемых цепочек*.

Чтобы заставить команду **`cmps`** выполняться несколько раз, то есть производить последовательное сравнение элементов цепочек, необходимо перед

командой **cmps** определить префикс повторения. С командой cmps можно использовать префикс повторения **repe/repz** или **repne/repnz**:

- **repe** или **repz** — если необходимо организовать сравнение до тех пор, пока не будет выполнено одно из двух условий:
  - достигнут конец цепочки (содержимое **ecx/cx** равно нулю);
  - в цепочках встретились разные элементы (флаг *ZF* стал равен нулю);
- **repne** или **repnz** — если нужно проводить сравнение до тех пор, пока:
  - не будет достигнут конец цепочки (содержимое **ecx/cx** равно нулю);
  - в цепочках встретились одинаковые элементы (флаг *ZF* стал равен единице).

Таким образом, выбрав подходящий префикс, удобно использовать команду **cmps** для поиска одинаковых или различающихся элементов цепочек. Выбор префикса определяется причиной, которая приводит к выходу из цикла. Таких причин может быть две для каждого из префиксов. Так как в регистре **ecx/cx** содержится счетчик повторений для цепочечной команды, имеющей любой из префиксов повторения, то, анализируя **ecx/cx**, можно определить причину выхода из зацикливания цепочечной команды. Если значение в **ecx/cx** *не равно нулю*, то это означает, что выход произошел по причине совпадения, либо несовпадения очередных элементов цепочек. Если же значение в **ecx/cx** *равно нулю*, то выход произошел по выполнению нужного количества повторений и совпадения, либо несовпадения очередных элементов цепочек не произошло.

Для определения конкретной причины наиболее подходящим является способ, использующий команду условного перехода **jcxz**. Ее работа заключается в анализе содержимого регистра **ecx/cx**, и если оно равно нулю, то управление передается на метку, указанную в качестве операнда **jcxz**. Однако, если при сравнении элементов отличие в последнем проверяемом элементе, то счетчик обнулится, но условие равенства будет не выполнено, поэтому переход по **jcxz** будет неверным. В этом случае нужна дополнительная проверка флагов, сформированных командой.

Существует возможность еще больше конкретизировать информацию о причине, приведшей к окончанию операции сравнения. Сделать это можно с помощью команд условной передачи управления (таблица 1 и 2).

Таблица 1. Сочетание команд условной передачи управления с результатами команды **`cmps`** (для чисел со знаком)

Причина прекращения операции сравнения	**Команда условного перехода, реализующая переход по этой причине**
Операнд_источник > Операнд_приемник	**`jg`**
Операнд_источник = Операнд_приемник	**`je`**
Операнд_источник <> Операнд_приемник	**`jne`**
Операнд_источник < Операнд_приемник	**`jl`**
Операнд_источник <= Операнд_приемник	**`jle`**
Операнд_источник >= Операнд_приемник	**`jge`**

Таблица 2. Сочетание команд условной передачи управления с результатами команды **`cmps`** (для чисел без знака)

Причина прекращения операции сравнения	**Команда условного перехода, реализующая переход по этой причине**
Операнд_источник > Операнд_приемник	**`ja`**
Операнд_источник = Операнд_приемник	**`je`**
Операнд_источник <> Операнд_приемник	**`jne`**
Операнд_источник < Операнд_приемник	**`jb`**
Операнд_источник <= Операнд_приемник	**`jbe`**
Операнд_источник >= Операнд_приемник	**`jae`**

Для определения местоположение очередных совпавших или не совпавших элементов в цепочках следует знать, что после выхода из цикла в соответствующих индексных регистрах находятся адреса элементов, расположенных в цепочке **после (!) элементов**, которые послужили причиной выхода из цикла. *Для получения истинного адреса этих элементов необходимо скорректировать содержимое индексных регистров, увеличив или уменьшив значение в них на длину элемента цепочки.*

Пример 2. Написать программу проверки вхождения подстроки в строку символов. Строка и подстрока задаются константами в программе.

```
; Template for console application
     .data
IsxStroka db  'Primer of string mov'   ; исходная строка
PodStroka db   'string'                ; подстрока
Yes     db   'Yes'    ; подстрока входит в строку
No      db   'Not'    ; подстрока не входит в строку
Result  times 22 db  ' '   ;строка для вывода результата
      .text
_start:
        cld   ; сброс флага DF - обработка строки от начала к концу
        lea  edi,[IsxStroka] ; загрузка в esi смещения строки
        lea esi,[PodStroka]  ; загрузка в edi смещения подстроки
        mov  ecx,15   ; количество сравнений строки и подстроки ls-lp+1
cycl:   push edi
        push esi
        push ecx
        mov  ecx,6  ; счетчик повторений (длина подстроки)
 repe   cmpsb
        jne next     ;  если сравниваемые символы различны, а cx=0
        jcxz  equal ; cx=0 и символы одинаковы, то совпадение найдено
next:    pop ecx
        pop esi
        pop edi
        inc edi
        loop cycl
        jmp not_equal ; вышли по счетчику, значит строки не совпадают
equal:  mov ecx,3
        lea  esi,[Yes]
        lea edi,[Result] ; подготовка вывода сообщения YES
    rep  movsb
        jmp kon
not_equal: mov ecx,3
```

```
        lea esi,[No]
        lea edi,[Result] ; подготовка вывода сообщения NOT
    rep  movsb
kon: ...
```

1.3. Операция сканирования цепочек

Команды, реализующие эту операцию, выполняют поиск некоторого значения в области памяти. Логически эта область памяти рассматривается как последовательность (цепочка) элементов фиксированной длины размером 8, 16 или 32 бит.

`scas` Адрес_приемника

Команда имеет один операнд, обозначающий местонахождение цепочки в дополнительном сегменте (адрес цепочки должен быть заранее сформирован в **`es:edi/di`**).

Транслятор анализирует тип идентификатора *Адрес_приемника*, который обозначает цепочку в сегменте данных, и формирует одну из трех машинных команд **`scasb`, `scasw` или `scasd`**.

Условие поиска для каждой из этих трех команд находится в строго определенном месте. Так:

если цепочка описана с помощью директивы **`db (resb)`**, то искомый элемент должен быть байтом и находиться в **`al`**, а сканирование цепочки осуществляется командой **`scasb`**;

если цепочка описана с помощью директивы **`dw (resb)`**, то это — слово в **`ax`**, и поиск ведется командой **`scasw`;**

если цепочка описана с помощью директивы **dd**, то это — двойное слово в **`eax`,** и поиск ведется командой **`scasd`.**

Принцип поиска тот же, что и в команде сравнения **`cmps`**, то есть последовательное выполнение вычитания (содержимое регистра аккумулятора – содержимое очередного элемента цепочки).

В зависимости от результата вычитания производится установка флагов, при этом сами операнды не изменяются.

Так же, как и в случае команды **`cmps`**, с командой **`scas`** удобно использовать префиксы **`repe/repz`** или **`repne/repnz`**:

- **repe** или **repz** — если нужно организовать поиск до тех пор, пока не будет выполнено одно из двух условий:
  - достигнут конец цепочки (содержимое **ecx/cx** равно 0);
  - в цепочке встретился элемент, отличный от элемента в регистре **al/ax/eax**;
- **repne** или **repnz** — если нужно организовать поиск до тех пор, пока не будет выполнено одно из двух условий:
  - достигнут конец цепочки (содержимое **ecx/cx** равно 0);
  - в цепочке встретился элемент, совпадающий с элементом в регистре **al/ax/eax**.

Таким образом, команда scas с префиксом **repe/repz** позволяет найти элемент цепочки, *отличающийся* по значению от заданного в аккумуляторе.

Команда **scas** с префиксом **repne/repnz** позволяет найти элемент цепочки, совпадающий по значению с элементом в аккумуляторе.

Пример 3. Написать программу поиска в исходной строке символа, который указан в константе.

```
    .data
IsxStroka db 'Primer of string mov' ; строка-источник
Symbol db  's' ; символ для поиска
Yes    db  'Yes'   ; выводится, если символ обнаружен
No     db  'Not'   ; выводится, если символ не обнаружен
Result times 22 db ' '  ;строка-результат
    .text
_start:
     cld   ; сброс флага DF - обработка строки от начала к концу
     lea  edi,[IsxStroka] ; загрузка в edi смещения строки-источника
        mov      ecx,20 ; длина строки источника
        mov  al,Symbol ; загрузка символа для поиска
  repne  scasb
     je   equal ; символ найден
      mov ecx,3 ; символ не найден
      lea esi,[No]
      lea edi,[Result] ; подготовка вывода сообщения Нет
   rep  movsb
```

```
        jmp kon
equal: mov ecx,3
        lea esi,[Yes]
        lea edi,[Result]  ; подготовка вывода сообщения Да
rep     movsb
kon:    ...
```

1.4. Загрузка элемента цепочки в аккумулятор

Эта операция позволяет извлечь элемент цепочки и поместить его в регистр-аккумулятор **al**, **ax** или **eax**. Эту операцию удобно использовать вместе с поиском (сканированием) с тем, чтобы, найдя нужный элемент, извлечь его (например, для изменения). Возможный размер извлекаемого элемента определяется применяемой командой.

Программист может использовать четыре команды загрузки элемента цепочки в аккумулятор, работающие с элементами разного размера:

lods Адрес_источника (LOaD String) — загрузить элемент из цепочки в регистр-аккумулятор al/ax/eax;

lodsb (LOaD String Byte) — загрузить байт из цепочки в регистр al;

lodsw (LOaD String Word) — загрузить слово из цепочки в регистр ax;

lodsd (LOaD String Double Word) — загрузить двойное слово из цепочки в регистр eax.

Команда имеет один операнд, обозначающий строку в основном сегменте данных. Работа команды заключается в том, чтобы извлечь элемент из цепочки по адресу, соответствующему содержимому пары регистров **ds:esi/si**, и поместить его в регистр **eax/ax/al**. При этом содержимое **esi/si** увеличивается или уменьшается (в зависимости от состояния флага *DF*) на значение, равное размеру элемента.

Эту команду удобно использовать после команды **scas**, локализующей местоположение искомого элемента в цепочке. Префикс повторения в этой команде может и не понадобиться — все зависит от логики программы.

1.5. Перенос элемента из аккумулятора в цепочку

Эта операция позволяет произвести действие, обратное команде **lods**, то есть сохранить значение из регистра-аккумулятора в элементе цепочки.

Операцию удобно использовать вместе с операцией поиска (сканирования) **scas** и загрузки **lods**, для того, чтобы, *найдя нужный элемент, извлечь его в регистр и записать на его место новое значение*.

Команды, поддерживающие эту операцию-примитив, могут работать с элементами размером 8, 16 или 32 бит:

stos Адрес_приемника (STOre String) — сохранить элемент из регистра-аккумулятора al/ax/eax в цепочке;

stosb (STOre String Byte) — сохранить байт из регистра al в цепочке;

stosw (STOre String Word) — сохранить слово из регистра ax в цепочке;

stosd (STOre String Double Word) - сохранить двойное слово из регистра eax в цепочке.

Команда имеет один операнд *Адрес_приемника*, адресующий цепочку в дополнительном сегменте данных.

Работа команды заключается в том, что она пересылает элемент из аккумулятора (регистра **eax/ax/al**) в элемент цепочки по адресу, соответствующему содержимому пары регистров **es:edi/di**.

При этом содержимое **edi/di** увеличивается или уменьшается (в зависимости от состояния флага *DF*) на значение, равное размеру элемента цепочки.

Префикс повторения в этой команде может и не понадобиться — все зависит от логики программы. Например, если использовать префикс повторения **rep**, то можно применить команду для инициализации области памяти некоторым фиксированным значением.

Пример 4. Используя команду **stosb** для предыдущего примера нахождения символа организовать вывод строки, сформированной из найденного символа в случае его обнаружения, а в случае отсутствия этого символа – сформировать строку из символов «*N*» и вывести ее на экран.

```
      .data
IsxStroka db   'Primer of String mov' ; строка-источник
Symbol    db   'S' ; символ для поиска
Result  times 22 db ' '  ; строка для записи результата
     .text
_start:
      cld       ; сброс флага DF - обработка строки от начала к концу
```

```
        lea   edi,[IsxStroka]      ; загрузка в di смещения строки-источника
        mov   ecx,20  ; длина строки источника
        mov   al,[Symbol] ; символ для поиска
    repne scasb
        je equal ; символ найден
not_equal:  mov ecx,22  ; символ не найден
        mov al,'N'
        lea edi,[Result] ; подготовка вывода строки из 22 символов N
    rep stosb
        jmp kon
equal:    mov ecx,22
        lea edi,[Result]  ; подготовка строки из  22 копий найденного символа
    rep stosb
kon:      ...
```

Пример 5. Дана строка, состоящая из слов, разделенных пробелом, после последнего слова – пробел. Написать программу определения количества слов, длина которых больше трех символов. Строку ввести с клавиатуры, результат вывести на экран.

Рассмотрим 2 варианта решения, 64-х и 32-х разрядные программы.

Вариант 1 – 64-х разрядная программа

```
        section .data    ; сегмент инициализированных переменных
ExitMsg  db "Press Enter to Exit",10
lenExit  equ $-ExitMsg
EnterMsg db "Enter string:",10
lenM     equ $-EnterMsg
Dlina    dw 3
;  комментарий вывода
RezMsg   db  13,10,"Kol. slov >3 simv.= "
lenR     equ $-RezMsg
RezBuf times 16 db  ' '   ; буфер для вывода результата
        section .bss    ; сегмент неинициализированных переменных
InBuf    resb    50     ; буфер для вводимой строки
lenIn    equ     $-InBuf
Rez      resd  1      ;  результат
        section .text   ; сегмент кода
```

```
        global  _start
_start:
    ; write
        mov     rax, 1   ; системная функция 4 (write)
        mov     rdi, 1   ; дескриптор файла stdout=1
        mov     rsi, EnterMsg  ; адрес выводимой строки
        mov     rdx, lenM ; длина выводимой строки
        syscall          ; вызов системной функции
         ; read
        mov     rax, 0   ; системная функция 3 (read)
        mov     rdi, 0   ; дескриптор файла stdin=0
        mov     rsi, InBuf ; адрес буфера ввода
        mov     rdx, lenIn ; размер буфера
        syscall       ; вызов системной функции
;    подсчет длины введенной строки до кода Enter
        lea rdi,[InBuf]  ; загружаем адрес строки в edi
        mov rcx,lenIn    ; загружаем размер буфера ввода
        mov al,0Ah       ; загружаем 0Ah для поиска в буфере ввода
        repne scasb      ; ищем код Enter в строке
        mov rax,50
        sub rax,rcx      ; вычитаем из размера буфера остаток cx
        mov rcx,rax      ; полученная разница – длина строки +1
;     добавление пробела в конец строки
        mov byte[rcx+InBuf-1],' '
;    подсчет количества слов с указанной длиной (> 3 символов)
        lea rdi,[InBuf]   ; загружаем адрес строки в edi
        mov al,' '        ; загружаем в al пробел для поиска
        mov ebx,0         ; обнуляем счетчик слов >3
        cld
        mov dx,0          ; обнуляем счетчик длины слова
cic1:   scasb          ; проверяем очередной символ на пробел
        je  consl         ; если пробел – на проверку длины слова
        inc dx         ; если нет – увеличиваем длину строки
        jmp prod1
```

```
consl:  cmp dx,[Dlina]    ; сравниваем длину слова с заданной
        jle prod          ; если меньше или равна продолжаем просмотр
        inc ebx           ;  иначе - увеличиваем счетчик слов
prod:   mov dx,0          ; обнуляем счетчик длины слова
prod1:  loop  cic1    ;
        mov [Rez],ebx     ; сохраняем результат
        mov eax,[Rez]     ; загрузка результата в eax
        lea rsi,[RezBuf]  ; адреса буфера в esi
        call IntToStr64   ; преобразование числа в строку
        mov rdx,rax       ; загрузка длины строки результата
      ; write
        add rdx,lenR      ; добавление длины строки-комментария
        mov     rax, 1    ; системная функция 4 (write)
        mov     rdi, 1    ; дескриптор файла stdout=1
        mov     rsi, RezMsg  ; адрес выводимой строки
        syscall           ; вызов системной функции вывода
         ; write
        mov     rax, 1    ; системная функция 4 (write)
        mov     rdi, 1    ; дескриптор файла stdout=1
        mov     rsi, ExitMsg ; адрес выводимой строки
        mov     rdx, lenExit ; длина выводимой строки
        syscall              ; вызов системной функции вывода
      ; read
        mov     rax, 0    ; системная функция 3 (read)
        mov     rdi, 0    ; дескриптор файла stdin=0
        mov     rsi, InBuf  ; адрес буфера ввода
        mov     rdx, lenIn  ; размер буфера
        syscall     ; вызов системной функции ввода
       ; exit
        mov     rax, 60   ; системная функция 60 (exit)
        xor     rdi, rdi   ; код возврата 0
        syscall     ; вызов системной функции завершения
%include "lib64.asm"
```

Вариант 2 – 32-х разрядная программа

```
        section .data   ; сегмент инициализированных переменных
ExitMsg  db "Press Enter to Exit",10
lenExit  equ $-ExitMsg
EnterMsg db "Enter string:",10
lenM     equ $-EnterMsg
Dlina    dw 3
;  комментарий вывода
RezMsg   db  13,10,"Kol. slov >3 simv.= "
lenR     equ $-RezMsg
RezBuf times 16 db  ' ' ;  буфер для вывода результата
        section .bss    ; сегмент неинициализированных переменных
InBuf    resb    50      ; буфер для вводимой строки
lenIn    equ     $-InBuf
Rez      resd  1      ;  результат
        section .text   ; сегмент кода
        global  _start
_start:
        mov     eax, 4      ; системная функция 4 (write)
        mov     ebx, 1      ; дескриптор файла stdout=1
        mov     ecx, EnterMsg  ; адрес выводимой строки
        mov     edx, lenM ; длина выводимой строки
        int     80h         ; вызов системной функции
        mov     eax, 3      ; системная функция 3 (read)
        mov     ebx, 0      ; дескриптор файла stdin=0
        mov     ecx, InBuf ; адрес буфера ввода
        mov     edx, lenIn ; размер буфера
        int     80h         ; вызов системной функции
;     подсчет длины введенной строки до кода Enter
        lea edi,[InBuf]     ; загружаем адрес строки в edi
        mov ecx,lenIn       ; загружаем размер буфера ввода
        mov al,0Ah          ; загружаем 0Ah для поиска в буфере ввода
        repne scasb         ; ищем код Enter в строке
```

```
            mov ax,50
            sub ax,cx           ; вычитаем из размера буфера остаток cx
            mov cx,ax           ; полученная разница – длина строки +1
      ; добавление пробела в конец
            mov  byte[ecx+InBuf-1],' '
;       подсчет количества слов с указанной длиной (> 3 символов)
            lea edi,[InBuf]     ; загружаем адрес строки в edi
            mov al,' '          ; загружаем в al пробел для поиска
            mov ebx,0           ; обнуляем счетчик слов >3
            cld
            mov dx,0            ; обнуляем счетчик длины слова
cic1:       scasb               ; проверяем очередной символ на пробел
            je  consl      ; если пробел – на проверку длины слова
            inc dx       ; если нет – увеличиваем длину строки
            jmp prod1
consl:      cmp dx,[Dlina]    ; сравниваем длину слова с заданной
            jle prod          ; если меньше или равна продолжаем просмотр
            inc ebx           ; иначе - увеличиваем счетчик слов
prod:       mov dx,0          ; обнуляем счетчик длины слова
prod1:      loop cic1         ; переходим к новому слову
            mov [Rez],ebx     ; сохраняем результат
            mov eax,[Rez]     ; загрузка результата в eax
            lea esi,[RezBuf] ; адреса буфера в esi
            call IntToStr     ; преобразование числа в строку
            mov edx,eax       ; загрузка длины строки результата
           ; write
            add edx,lenR      ; добавление длины строки-комментария
            mov     eax, 4    ; системная функция 4 (write)
            mov     ebx, 1    ; дескриптор файла stdout=1
            mov     ecx, RezMsg; адрес выводимой строки
            int     80h       ; вызов системной функции вывода
           ; write
            mov     eax, 4    ; системная функция 4 (write)
            mov     ebx, 1    ; дескриптор файла stdout=1
```

```
        mov     ecx, ExitMsg ; адрес выводимой строки
        mov     edx, lenExit ; длина выводимой строки
        int     80h          ; вызов системной функции вывода
        ; read
        mov     eax, 3       ; системная функция 3 (read)
        mov     ebx, 0       ; дескриптор файла stdin=0
        mov     ecx, InBuf   ; адрес буфера ввода
        mov     edx, lenIn   ; размер буфера
        int     80h          ; вызов системной функции ввода
        ; exit
        mov     eax, 1       ; системная функция 1 (exit)
        xor     ebx, ebx     ; код возврата 0
        int     80h          ; вызов системной функции завершения
%include "lib.asm"
```

Порядок выполнения домашнего задания

1. Прочитать и проанализировать задание в соответствии со своим вариантом.

2. Разработать алгоритм программы.

3. Разработать тесты для тестирования и отладки программы в соответствии с ее особенностями.

4. Создать новый файл и закодировать программу в соответствие с разработанным алгоритмом.

6. Отладить программу.

7. Составить отчет по домашнему заданию.

8. Продемонстрировать работу программы преподавателю.

9. Защитить задание преподавателю.

Варианты заданий

Варианты заданий приведены на странице дисциплины на сайте кафедры.

Требования к отчету

Все записи в отчете должны быть либо напечатаны на принтере, либо разборчиво выполнены от руки синей или черной ручкой (карандаш – не допускается). Схемы также должны быть напечатаны при помощи компьютера или нарисованы с использованием чертежных инструментов, в том числе карандаша.

Отчет по каждой части домашнего задания должен содержать:

1) текст задания;

2) схему алгоритма;

3) текст программы;

4) таблицы тестов;

5) выводы.

Кроме того, все отчеты должны иметь титульный лист, на котором указывается:

а) наименование факультета и кафедры;

б) название дисциплины;

в) номер и тема домашнего задания;

г) фамилия преподавателя, ведущего занятия;

д) фамилия, имя и номер группы студента;

е) номер варианта задания.

Контрольные вопросы

1. Дайте определение символьной строки.

2. Назовите основные команды обработки цепочек?

3. Какие операции выполняют строковые команды **MOVS**? Какие особенности характерны для этих команд?

4. Какие операции выполняют строковые команды **CMPS**, **SCAS**? Какие особенности характерны для этих команд?

5. Как обеспечить циклическую обработку строк?

6. Какова роль флага *DF* во флажковом регистре при выполнении команд обработки строк?

7. Как правильно выбрать тестовые данные для проверки алгоритма обработки строки?

Литература

1. Конспект (слайды) лекций по дисциплине «Машинно-зависимые языки и основы компиляции». - М.: МГТУ им. Н.Э. Баумана, кафедра КС и С, 2022. – В эл. виде.

2. Иванова Г.С., Ничушкина Т.Н. Основы программирования на ассемблере IA-32 (IA-64). Учеб. пособие. – М.: МГТУ им. Н.Э. Баумана, 2022. – ЭУИ.