Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра программного обеспечения информационных технологий

В. В. Бахтизин, С. Б. Мусин, Е. В. Шостак

ЯЗЫКИ ПРОГРАММИРОВАНИЯ:
язык ассемблера

Методическое пособие
для студентов специальности 1-40 01 01
«Программное обеспечение информационных технологий»
дневной и дистанционной форм обучения

Минск БГУИР 2010

УДК 004.438(076)
ББК 32.973.26-018.1я73
Б30

Р е ц е н з е н т
Заведующий кафедрой вычислительных методов и программирования,
кандидат технических наук, доцент А. А. Иванюк

Бахтизин, В. В.

Б30 Языки программирования: язык ассемблера : методическое пособие для студ. спец. 1-40 01 01 «Программное обеспечение информационных технологий» днев. и дист. форм обуч. / В. В. Бахтизин, С. Б. Мусин, Е. В. Шостак. – Минск : БГУИР, 2010. – 32 с.

ISBN 978-985-488-565-0

Пособие является дополнительным учебным материалом для студентов, изучающих программирование на языке ассемблера. Приведены варианты заданий, позволяющих приобрести навыки в программировании различных задач.

УДК 004.438(076)
ББК 32.973.26-018.1я73

ISBN 978-985-488-565-0

СОДЕРЖАНИЕ

Основные понятия

Ассе́мблер (от англ. *assemble* – собирать) – это компилятор программы, написанной на языке ассемблера, в машинный код. Программа в машинном коде представляет собой список байт, которые группируются в машинные команды (инструкции), каждая из которых представляет собой элементарное действие, выполняемое процессором.

Язык ассемблера – это символическое представление машинного языка. Русифицированное название – *мнемокод*. Команды языка ассемблера представляют собой удобный для запоминания (мнемонический) вид машинных кодов команд: *MOVe, INTterrupt, RETurn*.

Пример.

Демонстрирует возможность ввода программы в машинном коде с использованием утилиты debug.exe, входящей в поставку ОС Windows вплоть до версии 7.

```
C:\WINDOWS\System32>debug.exe
-e 100 B4 09 BA 08 01 CD 21 C3 48 45 4C 4C 4F 24
-g
HELLO!
Program terminated normally
-u
17DD:0100 B409          MOV     AH,09
17DD:0102 BA0801        MOV     DX,0108
17DD:0105 CD21          INT     21
17DD:0107 C3            RET
17DD:0108 48            DEC     AX
17DD:0109 45            INC     BP
17DD:010A 4C            DEC     SP
17DD:010B 4C            DEC     SP
17DD:010C 4F            DEC     DI
-q
```

Для ввода инструкций используется команда “e [адрес]”. После запуска на выполнение с помощью команды “g”, программа выводит сообщение «HELLO!» на экран.

Как видно из примера, программирование с использованием машинного кода не удобно для человека. Язык ассемблера позволяет упростить эту задачу. Воспользовавшись командой “u” можно получить листинг программы на языке ассемблера. По справочнику [1-3] можно определить, что программа вызывает функцию 9h обработки прерывания 21h. Прерывание 21h зарезервировано для вызова функций операционной системы. Функция 9h «Запись строки в стандартное устройство вывода» принимает параметр – адрес строки (в данном примере 108h). Команда RET осуществляет возврат. Заметим важную особенность: утилита отладки не может отличить данные (ASCII коды символов строки «HELLO!») от машинных команд.

Команда “q” завершает работу утилиты.

Язык высокого уровня – это язык программирования, понятия и структура которого удобны для восприятия человеком. Любая программа на языке высокого уровня впоследствии транслируется в машинный код.

Пример.
Следующая программа сложения двух чисел написана на языке С в среде программирования Microsoft Visual Studio.

```
#include <stdio.h>

int a, b, c;

int main()
{
      a = 0x40;
      b = 2;
      c = a+b;

      printf("%d", c);
      return 0;
}
```

Рассмотрим, в какие команды языка ассемблера транлируется программа после компиляции. Поставив точку останова на последнюю строку программы (Toggle breakpoint, F9), запустим отладку программы (Start debugging, F5). После начала отладки вызовем окно Disassemlbly, Alt+8). Далее приведен тот фрагмент программы, который непосредственно выполняет суммирование:

```
      a = 0x40;
00A9138E  mov         dword ptr [a (0A974D4h)],40h
      b = 2;
00A91398  mov         dword ptr [b (0A974D0h)],2
      c = a+b;
00A913A2  mov         eax,dword ptr [a (0A974D4h)]
00A913A7  add         eax,dword ptr [b (0A974D0h)]
00A913AD  mov         dword ptr [c (0A974CCh)],eax
```

Языки высокого уровня отражают потребности программиста, но не возможности машины. Все процессы в машине на аппаратном уровне приводятся в действие только инструкциями машинного языка. Отсюда понятно, что, несмотря на общее название, язык ассемблера для каждого типа компьютера свой. Это касается и внешнего вида программ, написанных на ассемблере, и идей, отражением которых этот язык является. Так как язык ассемблера для компьютера “родной”, то и самая эффективная программа может быть написана только на нем (при условии, что ее пишет квалифицированный программист). Здесь есть одно маленькое “но”: это очень трудоемкий, требующий большого внимания и практического опыта процесс. Поэтому реально на ассемблере пишут в основном программы, которые должны обеспечить эффективную работу с аппаратной

частью. Иногда на ассемблере пишутся критичные по времени выполнения или расходованию памяти участки программы. Впоследствии они оформляются в виде подпрограмм и совмещаются с кодом на языке высокого уровня.

Пример

Модифицируем программу из предыдущего примера, воспользовавшись командой ассемблерной вставки:

```
/*
      a = 0x40;
      b = 2;
      c = a+b;
*/
      __asm {
            mov dword ptr a,40h
            mov dword ptr b,2
            mov eax,dword ptr a
            add eax,dword ptr b
            mov dword ptr c,eax
      }
```

Совместное применение языка ассемблера и языка высокого уровня снижает переносимость программы, но позволяет получить непосредственный доступ к аппаратуре и в некоторых случаях увеличить скорость.

Архитектура машины – абстрактное определение принципов организации и функции машины с точки зрения программиста. Архитектура скрывает внутреннее управление и передачу данных, конструктивные особенности логических схем, специфику технологии производства. Для изучения языка ассемблера необходимо выяснить, какая часть машины оставлена видимой и доступной для программирования.

Архитектура большинства современных компьютеров была предложена Джоном фон-Нейманом. Суть архитектуры фон-Неймана заключается в том, что используется последовательно адресуемая линейная память, отсутствует явное различие между командами и данными, тип данных определяется логикой программы.

Типичная **фон-неймановская архитектура** содержит три основных компонента: центральный процессор (ЦП), запоминающее устройство (ЗУ) и устройства ввода-вывода (УВВ).

ЦП главный компонент компьютера. Его задача – выполнять программы, находящиеся в ЗУ. Он вызывает команды из ЗУ, определяет их тип, а затем выполняет их одна за другой. ЗУ – совокупность последовательных ячеек, каждая из которых имеет свой уникальный адрес. Каждая ячейка состоит из последовательности битов. Значения битов (1 или 0) образуют содержимое ячейки.

ЦП состоит из нескольких частей. Устройство управления отвечает за вызов команд из памяти и определение их типа. Генератор тактовых импульсов служит для синхронизации внутренних операций с другими компонентами

системы. Арифметико-логическое устройство (АЛУ) выполняет арифметические и логические операции над входными данными.

Внутри ЦП находится память для хранения промежуточных результатов и некоторых команд управления. Эта память состоит из регистров, каждый из которых выполняет определенную функцию. Поскольку регистры находятся внутри ЦП, доступ к ним является очень быстрым. По способу доступа регистры разделяются на пользовательские и системные.

Выделяют следующие группы пользовательских регистров: **регистры общего назначения (РОН)**, **сегментные регистры**, **регистры состояния и управления (указатель команд и регистр флагов)**.

Восемь 32-битных РОН могут использоваться программистами для хранения данных и адресов. Регистры **eax/ax/ah/al**; **ebx/bx/bh/bl**; **edx/dx/dh/dl**; **ecx/cx/ch/cl** называют регистрами данных. Программист может использовать их по своему усмотрению для временного хранения любых объектов (данных или адресов) и выполнения над ними требуемых операций. При этом регистры допускают независимое обращение к старшим (**ah, bh, ch** и **dh**) и младшим (**al, bl, cl** и **dl**) половинам. Однако **многие команды требуют для своего выполнения использования определенных регистров.** Например, команда умножения **mul** требует, чтобы один из сомножителей был в регистре **ax** (или **al**), а команда организации цикла **loop** выполняет циклический переход **cx** раз. Регистры **ebp/bp; esi/si; edi/di; esp/sp** называются регистрами-указателями. Индексные регистры **esi/si** и **edi/di** так же, как и регистры данных, могут использоваться произвольным образом. Однако их основное назначение – **хранить индексы** (смещения) относительно некоторой базы (т. е. начала массива) при выборке операндов из памяти. Адрес базы при этом обычно находится в одном из базовых регистров (**bx** или **bp**). Регистр **ebp/bp** служит указателем базы при работе с данными в стековых структурах, но может использоваться и произвольным образом в большинстве арифметических и логических операций или просто для временного хранения данных.Регистр **esp/sp** используется исключительно как указатель вершины стека. Регистры **ebp/bp**, **esi/si**, **edi/di**, **esp/sp**, в отличие от регистров данных, **не допускают побайтовую адресацию.**

Любая программа на языке ассемблера состоит из **сегментов.** Сегмент ассемблера – это логически выделенная в исходном тексте программы область, используемая для определенных целей. Так, сегмент кода содержит команды, сегмент данных – данные, в сегменте стека отводится место под стек. 16-битные **сегментные регистры** предназначены для формирования адреса данных (**ds**, **es**, **gs**, **fs**), кода (**cs**) или стека (**ss**) в памяти. В ОС Windows используется защищенный режим работы, в котором адреса начала всех сегментов равны (flat модель).

Регистр **указателя команд eip** указывает относительный адрес следующей за исполняемой команды. Регистр **eip** программно не доступен. Наращивание адреса в нем выполняет микропроцессор. Регистр **флагов** содержит информацию

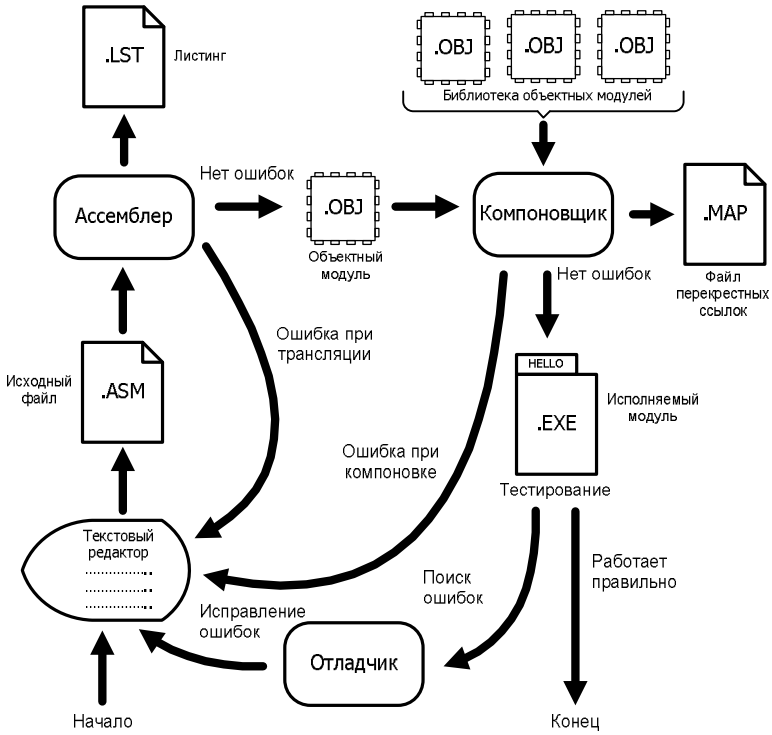
о текущем состоянии процессора. Он включает 6 флагов состояния и 3 бита управления состоянием процессора, которые тоже, обычно называются флагами.

ЦП выполняет каждую команду за несколько шагов (instruction execution cycle):

- Fetch. Вызывает следующую команду из памяти и переносит ее в регистр команд, затем увеличивает значение счетчика команд.
- Decode. Определяет тип вызванной команды, передает ноль или более операндов АЛУ и посылает АЛУ управляющие сигналы, которые определяют тип выполняемой команды.
- Fetch operands. Если команда использует операнды из памяти, инициирует операцию чтения ЗУ для получения операндов.
- Execute. АЛУ выполняет команду, заносит результат в результирующий операнд и обновляет содержимое регистра флагов.
- Store Output Operand. Если результирующий операнд находится в памяти, ЦП инициирует команду записи результата в ЗУ.

Компиляция, выполнение и отладка программы

Процесс подготовки и отладки программы на языке ассемблера включает этапы подготовки файла с исходным текстом, его трансляции и компоновки и, наконец, отладки программы с помощью специальной программы интерактивного отладчика.



Подготовка исходного текста программы выполняется с помощью любого текстового редактора. Файл с исходным текстом должен иметь расширение **.asm**.

Трансляция исходного текста программы состоит в преобразовании предложений исходного языка в машинный код и выполняется с помощью транслятора с языка ассемблера. В результате трансляции образуется объектный файл с расширением **.OBJ**. Опционально может генерироваться файл листинга программы с расширением **.LST**. Он содержит отчет о том, как скомпилировалась программа: исходный код, адреса команд (смещения в коде), объектный код (машинный код), имена сегментов, символы (переменные, процедуры, константы).

Компоновка объектного файла выполняется с помощью программы компоновщика (редактора связей). Компоновщик выполняет следующие функции: 1) подсоединяет к файлу с основной программой файлы с подпрограммами и настраивает связи между ними; 2) изменяет формат объектного файла и преобразует его в выполнимый файл, который может быть загружен в оперативную память и выполнен. В результате компоновки образуется исполняемый файл с расширением **.EXE**. Опционально создается файл перекрестных ссылок с расширением **.MAP**, который содержит следующую информацию по каждому сегменту программы: начальный адрес, конечный адрес, размер, тип сегмента.

Отладка готовой программы может выполняться разными методами, выбор которых определяется структурой и функциями отлаживаемой программы. В целом наиболее удобно отлаживать программы с помощью какого-либо интерактивного отладчика, который позволяет выполнять отлаживаемую программу по шагам или точкам останова. Для удобства отладки на этапе компиляции добавляется отладочная информация, а на этапе линковки программы создается файл символов отладчика, имеющий расширение **.PDB**.

Рассмотрим пример разработки программы вывода сообщения на экран.

1. Вызов командного интерпретатора ОС и настройка переменных окружения:

```
%comspec% /k ""C:\Program Files\Microsoft Visual Studio 9.0\VC\vcvarsall.bat"" x86
```

2. Создание файла:

```
notepad.exe hello.asm
```

3. Ввод исходного текста программы:

```
.686
.model flat, stdcall

ExitProcess PROTO :DWORD
MessageBoxA PROTO :DWORD, :DWORD, :DWORD, :DWORD

.data
  MsgBoxCaption db "First Program",0
```

```
    MsgBoxText db "HELLO!",0
.code

 main:
  push  0
  push  offset MsgBoxCaption
  push  offset MsgBoxText
  push  0
  call  MessageBoxA

  push   0
  call   ExitProcess
END main
```

Директива .686 определяет тип процессора, для которого создается программа. Директива MODEL указывает, что программа компилируется для защищенного режима работы процессора,а параметр STDCALL указывает, что будут вызываться функции ОС Windows. Директивы PROTO объявляют прототипы функций, которые используются в программе. Команда push помещает параметры функции в стек, команда call вызывает функцию. Директива END указывает точку входа программы.

4. Компиляция программы с добавлением отладочной информации и созданием файла листинга:

   ml /c /Zi /Fl hello.asm

5. Создание исполняемого файла и линковка программы, импорт вызываемых функций из библиотек ОС, генерация файла перекрестных ссылок, генерация символов отладчика:

   link /OPT:REF /MAP /DEBUG hello.obj kernel32.lib user32.lib

   При линковке программы следует явно указывать библиотеки для используемых функций, так библиотека kernel32.lib содержит функцию ExitProcess, а функция вывода сообщения на экран MessageBox вызывается из библиотеки user32.lib

6. Исполнение:

   hello.exe

7. Отладка:

   devenv hello.exe

   Для выполнения программы по шагам нажмите комбинацию клавиш Alt+F10 (Step over)

Задание: Создайте программу суммирования двух чисел и выполните ее по шагам.

Простые типы данных

Директивы определения данных

Директивы определения данных служат для задания размеров и содержимого данных, используемых в программе. При их обработке в памяти резервируется место для данных в объектном файле программы.
Директива определения данных имеет следующий формат:

[идентификатор] [директива] значение[,значение…][;комментарий]

Идентификатор – имя переменной, метка, значение которой соответствует смещению переменной относительно начала сегмента, в котором она расположена.

В следующей таблице приведены директивы и их описание.

Директива	Описание	Размер, байт	Тип данных, Си
db	define byte	1	char
dw	define word	2	short
dd	define double word	4	int, float, long
dp	define pointer	6	–
df	define far pointer	6	–
dq	define quad-word	8	double, long double
dt	define ten bytes	10	–

Чтобы отличать двоичные числа от десятичных, в ассемблерных программах в конце каждого двоичного числа ставят букву b или B. Неявно любая совокупность цифр считается десятичным числом, но для явного указания этого можно ставить в конце букву d (D). Для отличия восьмеричных чисел в конце ставят q (Q), шестнадцатеричных – h (H). При записи чисел, начинающихся с a, b, c, d, e, f, в начале приписывается цифра 0, чтобы не перепутать такое число с идентификатором.

Режимы адресации

Операнды команд могут находиться в памяти, в регистрах или в портах ввода-вывода. Где именно располагается операнд и по каким правилам вычисляется его физический адрес (если он расположен в памяти) определяет режим адресации. Различают следующие режимы адресации: **регистровый, непосредственный, прямой, косвенный.** Третья группа включает, в сущности, целый ряд способов адресации. Они обычно носят названия: базовая, индексная, адресация по базе со сдвигом, адресация с масштабированием, адресация по базе с индексированием. Также существуют портовая, строковая и стековая адресации, но эти виды очень

специфичны и используются только с соответствующими командами. То, какой именно режим адресации используется в данной команде, транслятор определяет исходя из синтаксиса задания этой команды в исходном тексте на ассемблере.

Регистровая адресация. Операнды могут располагаться в любых регистрах общего назначения и сегментных регистрах. В этом случае в тексте программы указывается название соответствующего регистра, например команда, копирующая в регистр EAX содержимое регистра EBX, записывается как

```
mov    eax,ebx
```

Непосредственная адресация. Некоторые команды (все арифметические команды, кроме деления) позволяют указывать один из операндов непосредственно в тексте программы, например команда

```
mov    eax,2
```

помещает в регистр EAX число 2.

Прямая адресация. В команде указывается символическое обозначение ячейки памяти, над содержимым которой требуется выполнить операцию.

```
mov    eax,mas
```

Содержимое байта памяти с символическим именем mas пересылается в регистр eax.

Базовая и индексная адресация памяти. Относительный адрес ячейки памяти находится в регистре, обозначение которого заключается в квадратные скобки. При использовании регистров EBX или EBP адресацию называют базовой, при использовании регистров ESI или EDI – индексной. При адресации через регистры EBX, ESI, EDI в качестве сегментного регистра подразумевается DS; при адресации через EBP – регистр SS. Таким образом, косвенная адресация через регистр EBP предназначена для работы со стеком.

```
mov    eax, [ebx]
```

Базовая и индексная адресации памяти со смещением. Относительный адрес операнда определяется суммой содержимого регистра (EBX, EBP, ESI или EDI) и указанного в команде числа.

```
mov edx, [ebx+2]
mov edx, [ebx]+2
```

Адресация по базе с индексированием. В этом методе адресации смещение операнда в памяти вычисляется как сумма чисел, содержащихся в двух регистрах, и смещения, если оно указано. Все следующие команды – это разные формы записи одного и того же действия:

```
mov    eax,[ebx+esi+2]
mov    eax,[ebx][esi]+2
mov    eax,[ebx+2][esi]
mov    eax,[ebx][esi+2]
mov    eax,2[ebx][esi]
```

в регистр EAX помещается слово из ячейки памяти со смещением, равным сумме чисел, содержащихся в EBX и ESI, и числа 2.

Значительная часть рассмотренных выше способов адресации служит для обращения к ячейкам памяти. Таким образом, один и тот же конечный результат можно получить с помощью различных способов адресации. Однако команды с использованием различных методов адресации занимают различный объем памяти и выполняются за разное время. Поэтому тщательный выбор способов адресации позволяет оптимизировать программы по времени выполнения или требуемой памяти, а иногда и по тому, и по другому.

Пример программы с различными типами данных, описанными в сегменте данных и их загрузка в регистры с использованием команды mov.

```
.686
.model flat, stdcall
 ExitProcess PROTO :DWORD

.data
  mes     db   "Begin"        ; строка символов
  x1      db   26             ; число 26 занимает 1 байт
  x2      dw   257            ; число 257 занимает 1 слово
  x3      dw   201h           ; то же число в 16-ричной форме
  x4      dw   1000000001b    ; то же число в двоичной форме
  x5      db   1,2,3,4        ; байтовый массив из 4 членов
  x6      dd   0FFFFFFFFh     ; максимальное целое число в двойном  слове
  x7      db   256 dup (27)   ; массив из 256 байт, заполненным числом 27
  x8      db   256 dup (?)    ; исходное содержимое массива не имеет значения
  x9      db   256 dup ("*")  ; символьный массив

.code

  mov eax, dword ptr [mes]
  mov ah,  x1
  mov ax,  x2

 main:
  push   0
  call   ExitProcess
END main
```

Если посмотреть сегмент данных программы (адрес первой переменной в памяти соответствует адресу начал сегмента данных)**,** то мы увидим, что для слов данных в памяти **сначала расположен младший байт, а затем старший (little-endian byte order)**. Символы строки хранятся в памяти в порядке их следования, т. е. первый символ имеет самый младший адрес, последний – самый старший.

Задание: Создайте исходный модуль **lab.asm**, содержащий сегмент данных с переменными. Загрузите исполняемый модуль в отладчик и найдите в окне дампа памяти все объявленные переменные. Составьте слово из массива данных, представляющего собой алфавит, используя различные способы адресации.

Массивы

Массив – структурированный тип данных, состоящий из некоторого числа элементов одного типа.

Специальных средств описания массивов в программах ассемблера нет.

Чтобы использовать массив в программе, его нужно имитировать одним из перечисленных далее способов.

Можно перечислить элементы массива в поле операндов одной из директив описания данных. При перечислении элементы разделяются запятыми. Например, массив из пяти элементов. Размер каждого элемента 4 байта:

```
mas dd 1,2,3,4,5
```

Можно использовать оператор повторения DUP. К примеру, массив из пяти нулевых элементов. Размер каждого элемента 2 байта:

```
mas dw 5 dup (0)
```

Такой способ определения используется для резервирования памяти с целью размещения и инициализации элементов массива.

Можно использовать директивы LABEL и REPT. Пара этих директив позволяет облегчить описание больших массивов в памяти и повысить наглядность такого описания. Директива REPT относится к макросредствам языка ассемблера и вызывает повторение указанное число раз строк, заключенных между директивой и строкой ENDM. К примеру, определим массив байтов в области памяти, обозначенной идентификатором mas_b. В данном случае директива LABEL определяет символическое имя mas_b аналогично тому, как это делают директивы резервирования и инициализации памяти. Достоинство директивы LABEL в том, что она не резервирует память, а лишь определяет характеристики объекта.

В данном случае объект – это ячейка памяти. Используя несколько директив LABEL, записанных одна за другой, можно присвоить одной и той же области памяти разные имена и типы, что и сделано в следующем фрагменте:

```
mas_b label byte
rnas_w label word
```

Эту последовательность можно трактовать как массив байтов или слов в зависимости от того, какое имя области мы будем использовать в программе – mas_b или mas_w.

Чтобы инициализировать значениями область памяти и впоследствии трактовать ее как массив, можно использовать цикл.

При работе с массивами необходимо четко представлять себе, что все элементы массива располагаются в памяти компьютера последовательно.

К примеру, пусть в программе статически определена последовательность данных:

```
mas dw 0 , 1 , 2 , 3 , 4 , 5
```

Эта последовательность чисел трактуется как одномерный массив. Размерность каждого элемента определяется директивой DW, то есть она равна двум байтам. Чтобы получить доступ к третьему элементу нужно к адресу массива прибавить 6. **Нумерация элементов массива в ассемблере начинается с нуля**. То есть в нашем случае речь фактически идет о 4-м элементе массива – 3, но об этом знает только программист; процессору в данном случае все равно – ему нужен только адрес. В общем случае для получения адреса элемента в массиве необходимо начальный (базовый) адрес массива сложить с произведением индекса (номер элемента минус единица) этого элемента на размер элемента массива:

база + (индекс * размер элемента).

Архитектура процессора предоставляет довольно удобные программно-аппаратные средства для работы с массивами. К ним относятся **базовые и индексные** регистры, позволяющие реализовать несколько режимов адресации данных. Используя данные режимы адресации, можно организовать эффективную работу с массивами в памяти.

Пример: поместить 3-й элемент массива a в регистр eax.

```
.data
  a db  10h, 20h, 30h, 40h, 50h
.code
  mov ebx,2
  mov eax,a[ebx]
```

Для перебора элементов массива можно воспользоваться командой loop, которая позволяет выполнить цикл со счетчиком в регистре ecx. Команда **автоматически** уменьшает содержимое регистра ecx и выполняет переход в начало блока, пока значение регистра ecx не станет равным 0.

Пример: суммирование элементов массива в регистре eax.

```
. data
a dw 1 dup(5),2,3,4,5
.code
mov edi, OFFSET a
mov ecx, LENGTHOF a; задать количество повторений
mov eax,  0; результат формируется в регистре eax
L1:
add eax,[edi] ; выполнить суммирование
add edi,TYPE a; перейти к следующему элементу
loop L1 ; повторить пока ECX != 0
```

Компиляторы Microsoft редко генерируют команду loop. Большинство циклов представляет собой комбинацию условных и абсолютных переходов.

Варианты индивидуальных заданий

1. Подсчитать количество четных элементов в одномерном массиве. Результат вывести на экран, вставив нужные цифры в шаблон текста сообщения. Например, "в массиве 3 четных элемента". Массив задать в сегменте данных.

2. Подсчитать количество нечетных элементов в одномерном массиве. Результат вывести на экран, вставив нужные цифры в шаблон текста сообщения. Например, "в массиве 5 нечетных элементов". Массив задать в сегменте данных.

3. Найти минимальный и максимальный элементы одномерного массива. Массив задать в сегменте данных.

4. Задан одномерный массив из 10 элементов. Подсчитать количество элементов не равных нулю. Результат вывести на экран, вставив нужные цифры в шаблон текста сообщения. Например, "в массиве 5 элементов, не равных 0". Массив задать в сегменте данных.

5. Задан одномерный массив. Вывести на экран элементы массива, кратные пяти, и их количество. Массив задать в сегменте данных.

6. Задан одномерный массив. Вывести на экран сумму положительных и число отрицательных элементов массива. Массив задать в сегменте данных.

7. Задан одномерный массив. Отсортировать его по возрастанию. Массив задать в сегменте данных.

8. Задан одномерный массив. Отсортировать его по убыванию. Массив задать в сегменте данных.

9. Задан одномерный массив. Заменить все большие семи элементы массива числом 7. Подсчитать количество таких замен. Массив задать в сегменте данных.

10. Задан одномерный массив из 20 элементов. Вычислить разность между максимальным и минимальным значением. Массив задать в сегменте данных.

11. Задан одномерный массив. Вывести на экран элементы массива, которые больше пяти и их количество. Результат вывести на экран, вставив нужные цифры в шаблон текста сообщения. Например, "в массиве 3 элемента, больше 5". Массив задать в сегменте данных.

12. Задан одномерный массив. Найти сумму его элементов. Результат вывести на экран, вставив нужные цифры в шаблон текста сообщения. Например, "сумма элементов массива 50". Массив задать в сегменте данных.

13. Заданы два одномерных массива. Вывести на экран и подсчитать количество неповторяющихся в них элементов. Массивы задать в сегменте данных.

14. Заданы два одномерных массива. Подсчитать количество повторяющихся в них элементов. Массивы задать в сегменте данных.

15. В одномерном массиве заменить отрицательные элементы нулями. Подсчитать число замен. Массив задать в сегменте данных.

16. Среди элементов одномерного массива найти наибольший отрицательный и наименьший положительный элементы. Массив задать в сегменте данных.

Строки

В языках высокого уровня выделяют следующие два основных типа строки:

- ASCIIZ – строка, завершающаяся нулем (Си)
- строка с полем, содержащим длину (Паскаль).

Длина ASCIIZ строки ограничена только объемом доступной процессу памяти, причем фактическая длина строки только на один байт больше исходной ASCII строки. К недостаткам этого типа строк следует отнести невозможность содержать нулевые байты. Операции с такими строками выполняются медленнее, так как необходимо проверять, не является ли текущий символ концом строки. Строки с заранее заданной длиной лишены этих недостатков, но их максимальная длина ограничена количеством байтов, выделенных для поля длины.

Команды для работы со строками центрального процессора или цепочечные команды – это мощный инструмент для работы со всеми регулярными типами данных, а не только символьными строками. Под *строкой* здесь понимается последовательность байтов, а *цепочка* – это более общее название для случаев, когда элементы последовательности имеют размер больше байта – слово или двойное слово. Цепочечные команды позволяют проводить действия над блоками памяти, представляющими собой последовательности 1, 2, 4-байтных элементов размера.

Строковые команды делятся на три группы:

- команды пересылки строк;
- команды проверки строк;
- команды префикса повторения.

Всего в системе команд микропроцессора имеется семь *операций-примитивов* обработки цепочек. Каждая из них реализуется в микропроцессоре тремя командами, в свою очередь, 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** к следующему элементу данной цепочки.

Перечислим операции-примитивы и команды, с помощью которых они реализуются, а затем подробно их рассмотрим:

- *пересылка цепочки*: **movs адрес_приемника,адрес_источника**

  movsb; movsw; movsd

  Это аналог функции memcpy языка Си.

- *сравнение цепочек*:

  cmps адрес_приемника,адрес_источника

  cmpsb; cmpsw; cmpsd

  Если сканируемая строка завершается символом NULL, эта команда соответствует функции strlen языка Си.

- *сканирование цепочки*:

  scas адрес_приемника

  scasb; scasw; scasd

  Это аналог функции memcmp языка Си.

- *загрузка элемента из цепочки*:

  lods адрес_источника

  lodsb; lodsw; lodsd

- *сохранение элемента в цепочке*:

  stos адрес_приемника

  stosb; stosw ;stosd

Логически к этим командам нужно отнести и так называемые **префиксы повторения**. Они предназначены для использования цепочечными командами.

Префиксы повторения имеют свои мнемонические обозначения:

rep
repe или **repz**
repne или **repnz**

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

Различия приведенных префиксов в том, на каком основании принимается решение о циклическом выполнении цепочечной команды: *по состоянию регистра ecx/cx или 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 в ecx/cx не станет равным 0*.
  При этом цепочечная команда, перед которой стоит префикс, *автоматически уменьшает содержимое ecx/cx на единицу*. Та же команда, но без префикса, этого не делает;

- 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**.

Важно отметить, что допускается *замена* (с помощью префикса замены сегмента) только регистра **ds**, регистр es подменять нельзя. Вторые части адресов - *смещения цепочек* – также находятся 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 – пара **ds:esi/si**;
- адрес_приемника – пара **es:edi/di**.

Последний 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.

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.

Операция пересылки цепочек

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

Команда **movs**

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

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 64 Кбайт данных (если размер адреса в сегменте 16 бит – use16) или до 4 Гбайт данных (если размер адреса в сегменте 32 бит - use32). Число пересылаемых элементов должно быть загружено в *счетчик* – регистр **cx** или **ecx**.

Пример использования префикса повторения и команды пересылки

```
.data
source DWORD 20 DUP(?)
target DWORD 20 DUP(?)
.code
cld         ; обход в прямом направлении
mov ecx,LENGTHOF source ; задаем количество повторений
mov esi,OFFSET source
mov edi,OFFSET target
rep movsd ; выполняем пересылку
```

Операция сравнения цепочек

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

Команда **cmps**

Синтаксис команды cmps:

cmps адрес_приемника,адрес_источника

Здесь:

- *адрес_источника* определяет **цепочку-источник** в сегменте данных. Адрес цепочки должен быть заранее загружен в пару **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-получателя) 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.

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. 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.

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

Как определить местоположение очередных совпавших или не совпавших элементов в цепочках?

После каждой итерации цепочечная команда автоматически осуществляет инкремент/декремент значения адреса в соответствующих индексных регистрах. Поэтому после выхода из цикла в этих регистрах будут находиться адреса элементов, находящихся в цепочке **после (!) элементов**, которые послужили причиной выхода из цикла. Для получения истинного адреса этих элементов

необходимо скорректировать содержимое индексных регистров, увеличив либо уменьшив значение в них на длину элемента цепочки.

Операция сканирования цепочек

Команды, реализующие эту операцию, производят поиск некоторого значения в области памяти. Логически эта область памяти рассматривается как последовательность (цепочка) элементов фиксированной длины размером 8, 16 или 32 бит. Искомое значение предварительно должно быть помещено в регистр **al/ax/eax**. Выбор конкретного регистра из этих трех должен быть согласован с размером элементов цепочки, в которой осуществляется поиск.

Команда **scas**

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, то искомый элемент должен быть байтом и находиться в al, а сканирование цепочки осуществляется командой scasb; если цепочка описана с помощью директивы dw, 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 результатов 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.

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.

Команда **lods**

lods адрес_источника (LOaD String) – загрузить элемент из цепочки в аккумулятор **al/ax/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 подвергается инкременту или декременту (в зависимости от состояния флага df) на значение, равное размеру элемента.

Префикса повторения в команде lods нет, так как он попросту не нужен.

Перенос элемента из аккумулятора в цепочку

Эта операция позволяет произвести действие, обратное команде lods, то есть сохранить значение из регистра-аккумулятора в элементе цепочки. Эту операцию удобно использовать вместе с операцией поиска (сканирования) scans и загрузки lods, с тем, чтобы, *найдя нужный элемент, извлечь его в регистр и записать на его место новое значение*. Команды, поддерживающие эту операцию, могут работать с элементами размером 8, 16 или 32 бит.

Команда **stos**

stos адрес_приемника (STOrage String) – сохранить элемент из регистра-аккумулятора al/ax/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 подвергается инкременту или декременту (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.

Варианты индивидуальных заданий

1. Сколько раз введенный символ совпадает с элементом строки. Строку описать в сегменте данных. (Использовать команду scas).
2. Сколько раз введенный символ не совпадает с элементом строки. Строку описать в сегменте данных. (Использовать команду scas).
3. Изменить регистр введенного символа на противоположный по всей строке. (Использовать команду scas). Строку описать в сегменте данных.
4. Вывести номера позиций указанного элемента в строке. (Использовать команду scas). Строку описать в сегменте данных.
5. Найти расстояние между указанными элементами в строке. (Использовать команду scas). Строку описать в сегменте данных.
6. Заменить указанный символ на пробел по всей строке. (Использовать команду scas). Строку описать в сегменте данных.
7. Определить, сколько элементов строки превышают код введенного символа. (Использовать команду scas). Строку описать в сегменте данных.
8. Определить, сколько элементов строки не превышают код введенного символа. (Использовать команду scas). Строку описать в сегменте данных.
9. Определить, входит ли в строку А подстрока Б? Строки А и Б описать в сегменте данных.
10. Заменить указанный элемент нулем по всей строке. (Использовать команду scas). Строку описать в сегменте данных.
11. Переслать строку произвольной длины, включив ее в состав другой, более длинной строки. Вывести полученную строку на экран. Строки описать в сегменте данных.
12. Сравнить две одинаковые строки, вывести на экран результат сравнения. Модифицировать программу, сделав строки не одинаковыми.
13. Заменить первый символ строки на второй по всей строке. (Использовать команду scas). Строку описать в сегменте данных.
14. Прибавить число к указанному элементу по всей строке. (Использовать команду scas). Строку описать в сегменте данных.
15. Заполнить массив тридцатью нулями. (Использовать команду stos).
16. Заменить указанный элемент пробелом по всей строке. (Использовать команду scas). Строку описать в сегменте данных.

Процедуры

Процедура (или подпрограмма) – это группа логически выделенных команд, которые могут быть многократно вызваны программой. Вызовом процедуры называют передачу ей управления. При появлении в программе повторяющихся участков кода логично использовать подпрограммы: это уменьшает размер кода программы. Однако вызов процедуры требует дополнительных затрат времени на выполнение перехода и возврата из процедуры.

Для описания процедур в языке Ассемблер используются две директивы – **proc** и **endp**:

```
имя_процедуры proc [тип] [язык] [USES регистры]     ; заголовок процедуры
…                          ; команды , составляющие тело процедуры
имя_процедуры endp
```

Идентификатор, используемый как «имя_процедуры» в директиве **proc**, по сути является меткой и указывает на первую команду процедуры. Директива **endp** определяет место завершения процедуры. Модификаторы, перечисленные в квадратных скобках в директиве proc, являются необязательными. Модификатор «тип» может принимать значения **near** и **far**, соответственно создаваемые процедуры будут определяться как ближние и дальние. При наличии модификатора uses транслятор создаст последовательность команд push в самом начале тела процедуры, таким образом, в стек будут заноситься значения указанных в uses регистров. Перед командой ret при этом будет создана последовательность команд pop. Если тип процедуры не задан, то неявно он выбирается исходя из используемой модели памяти (для моделей tiny, small и compact – тип near).

Процедура может быть размещена в любом месте программы, однако надо учитывать, что после компиляции ее код не отделим от остального кода программы (то есть в запускаемой на выполнение программе все команды расположены линейно друг за другом, как будто в исходном тексте на ассемблере не было директив proc и endp). Если в своем выполнении программа дойдет до первой команды процедуры без ее явного вызова, то это может привести к неожиданным последствиям.

Чтобы избежать ошибки, нужно либо размещать процедуры в начале или конце программы (до первой исполняемой команды или после команды, возвращающей управление операционной системе), либо в теле программы предусматривать обход процедуры с помощью команд перехода.

Для передачи управления процедуре используется команда вызова **call**:

```
call [модификатор] операнд
```

Команда call сохраняет в стеке адрес возврата и передает управление по адресу, который определяется операндом этой команды.

Для возврата из процедуры используется команда **ret** (return – возвращаться):

ret [число]
retn[число]
retf[число]

Команда ret передает управление по адресу, который находится на вершине стека. Команда retn используется для возврата из ближней процедуры, retf – из дальней. Команда ret всегда преобразуется транслятором, либо в retn либо в retf в зависимости от типа процедуры.

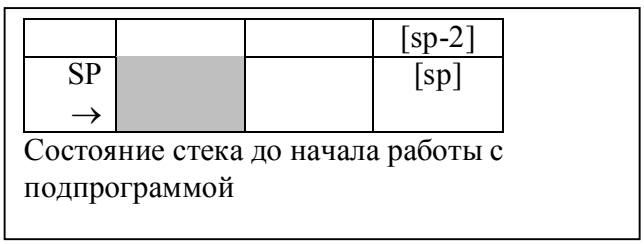
Передача параметров процедуре

При вызове процедуры в большинстве случаев ей передаются параметры из вызывающей программы. Параметры – это данные: байты, слова, строки, массивы или адреса байтов, слов, строк, массивов. Передача параметров по значению означает передачу самих данных, передача по ссылке означает передачу адресов. В обоих случаях возможна передача параметров через регистры, через общую память (по именам) или через стек.

Передача параметров через регистр не требует комментариев, однако, следует отметить принятые неписаные правила: один параметр по значению передается через аккумулятор: байт – через AL, слово – через AX, двойное слово – через EAX (для 32-разрядной модели) или через DX,AX (для 16-разрядной модели). Передача нескольких параметров через регистры – не самое лучшее решение.

Передача параметров через общую память (по имени) реализуется очень просто, если вызывающая и вызываемая процедуры расположены в одном модуле. Однако в этом случае процедуры перестают быть универсальными, т. к. связаны общими именами.

Передача параметров через стек – наиболее универсальный метод.



Состояние стека до начала работы с подпрограммой

Вызывающая программа помещает аргументы в стек и делает вызов процедуры:

```
push x1          ; первый аргумент
push x2          ; второй аргумент
call sub1        ; вызов подпрограммы
```

После такой последовательности команд в стеке оказываются два параметра и адрес возврата. Для ближнего вызова адрес возврата – это одно слово, представляющее собой адрес команды, следующей за командой **call**.

Получив управление, подпрограмма может получить доступ к аргументам

			[sp-2]
SP →	A.B.		[sp]
	X2		[sp+2]
	X1		[sp+4]
			[sp+6]

Состояние стека при входе в подпрограмму

через регистр **BP** (**B**ase **P**ointer), специально для этого введенный в состав процессора. Предварительно сохранив его старое значение в стеке, в него переписывают текущее значение указателя стека:

```
push bp          ; сохранение старого значения BP
mov  bp,sp       ; установка указателя базы BP
```

			[bp-2]
SP →	ст. SP	← **BP**	[bp]
	A.B.		[bp+2]
	X2		[bp+4]
	X1		[bp+6]

Состояние стека после установки **BP**

Из схемы видно, что аргументы доступны по адресам [bp+4], [bp+6] и т.д.:

```
mov ax,[bp+6]    ; пересылка первого аргумента в аккумулятор
mov cx,[bp+4]    ; пересылка второго аргумента в регистр CX
```

В случае дальнего вызова смещения будут: [bp+6], [bp+8] и т.д., т.к. адрес возврата будет занимать два слова в стеке.

Локальные данные процедуры

Если вызываемой подпрограмме необходимы ячейки памяти для хранения своих внутренних данных, то целесообразно разместить их в области стека, занимая память, таким образом, только на время работы подпрограммы. Для того чтобы «захватить» нужное количество слов памяти под локальные переменные, указатель стека перемещается выше (для процессора Intel это означает: в сторону меньших адресов):

```
sub  sp,4        ; захват четырех байтов в области стека
```

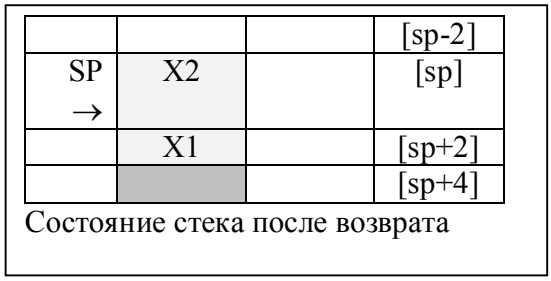
Локальные переменные доступны подпрограмме по адресам [bp-2], [bp-4] и т.д.:

Состояние стека после захвата памяти

```
mov  [bp-2],bx          ; запись значения в локальную переменную
```

После окончания действий в подпрограмме перед возвратом управления необходимо восстановить стек:

```
. . .          ;
mov  sp,bp     ; возврат «захваченной» памяти
pop  bp        ; восстановление старого значения BP
ret            ; команда возврата
```



Состояние стека после возврата

Поскольку аргументы уже не нужны, то можно очистить стек:

```
add  sp,4        ; освобождение стека от двух аргументов
```

Состояние стека после очистки

Очистку стека можно «поручить» подпрограмме, для этого используется команда возврата с параметром:

```
ret  4          ; возврат и очистка стека от двух аргументов
```

Понятно, что в этом случае вызывающая программа **не должна** очищать стек от аргументов.

Пример, директива MODEL указывает соглашение о вызовах

Вызывающая функция (C)

```
push val2
push val1
call AddTwo
add   esp,8
```

Вызываемая (STDCALL):

```
AddTwo PROC
 push  ebp
 mov   ebp,esp
 mov   eax,[ebp+12]
 add   eax,[ebp+8]
 pop   ebp
 ret   8
```

Модификатор Pascal также задает соглашение о вызовах в котором стек очищает вызываемая процедура, но параметры в стек передаются в прямом порядке слева направо.

Задание 1. Программа вызывает процедуру, которая выводит строку, символ или число в соответствии с вариантом. Параметры передаются через регистры. Процедура одна, вызовов несколько.

Задание 2. Повторить задание 1 с передачей параметров через глобальные переменные.

Задание 3. Повторить задание 1 с передачей параметров через стек.

ЛИТЕРАТУРА

1. Ирвин, К. Язык ассемблера для процессоров Intel/ К. Ирвин. – М. : Издательский дом “Вильямс”, 2005. – 912 с.
2. Абель, П. Ассемблер. Язык программирования для IBM PC/ П. Абель. – М. : Век, 2003. – 736 с.
3. Зубков, С. Assembler для DOS, Windows и UNIX для программистов / С. Зубков. – СПб. : Питер, 2005. – 608 с.
4. Пирогов В. Ассемблер и дизассемблирование / В. Пирогов. – СПб. : БХВ-Петербург, 2007. – 464 с.
5. Магда, Ю. Ассемблер для процессоров Intel Pentium / Ю. Магда. – СПб. : Питер, 2006. – 410 с.
6. Голубь, Н. Искусство программирования на Ассемблере / Н. Голубь. – СПб. : Питер, 2006. – 832 с.
7. Пильщиков, В. Assembler. Программирование на языке ассемблера IBM PC / В. Пильщиков. – М. : Диалог-МИФИ, 2004. – 288 с.
8. Касперски, К. Искусство дизассемблирования / К. Касперский, Е. Рокко. – СПб. : БХВ-Петербург, 2008. – 896 с.
9. Роббинс, Д. Поиск и устранение ошибок в программах под Windows / Д. Роббинс. – М. : ДМК Пресс, 2005. – 448 с.

Св.план 2010, поз. 48

Учебное издание

Бахтизин Вячеслав Вениаминович
Мусин Сергей Борисович
Шостак Елена Викторовна

ЯЗЫКИ ПРОГРАММИРОВАНИЯ:
язык ассемблера

Методическое пособие
для студентов специальности 1-40 01 01
«Программное обеспечение информационных технологий»
дневной и дистанционной форм обучения

Редактор Т. Н. Крюкова

Подписано в печать	Формат 60x84 1/16.	Бумага офсетная.
Гарнитура «Таймс».	Отпечатано на ризографе.	Усл. печ.л.
Уч.-изд.л. 2,0.	Тираж 100 экз.	Заказ 230.

Издатель и полиграфическое исполнение: Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
ЛИ №02330/0494371 от 16.03.2009. ЛП №02330/0494175 от 03.04.2009.
220013, Минск, П.Бровки, 6