# Ассемблер для блондинок
# ... и блондинов

## *1. Практическая работа на Ассемблере*

Необходимые учебные материалы:

[1] Баула В.Г. Введение в архитектуру ЭВМ и системы программирования.
`http://arch32.cs.msu.ru/`

Здесь можно читать пособие для студентов, разбитое по главам. Каждую главу можно скачать в формате `pdf`, щёлкнув правой кнопкой мыши по её названию в левом окне (оглавлении) и выбрав пункт меню "сохранить как".

[2] Бордаченкова Е. А. Задачи и упражнения по языку MASM (исправленный), 2022.pdf.
`http://arch32.cs.msu.ru/semestr2`

[3] Пильщиков В.Н. Программирование на языке ассемблера IBM PC.
`http://arch32.cs.msu.ru/semestr2`

---

Сразу заметим, что Ассемблер, на котором мы собираемся программировать, описан в [1], поэтому никаких отличий данного материала или работы Ассемблера от [1], быть не должно (кто заметит – пишите по адресу в [1]). Надо сделать следующее.

1. Скачать программное обеспечение. Сайт `http://arch32.cs.msu.ru/semestr2`, папка `masm 6.14.zip`. Это упакованная папка, ее надо разархивировать, причем никакие названия папок менять нельзя. Получившуюся папку с файлами под названием `masm 6.14` лучше всего поместить в корневой каталог любого диска (можно и на флешку), а вот на рабочий стол или в какие-то другие папки со сложными именами (`Мои Документы` и т.д.) лучше не помещать.

2. Попробовать, работает ли. В папке `_Examples` откройте один из первых примеров, кликните 2 раза по файлу `makeit.bat` (расширение у вас может не отображаться, файл может быть помечен как "пакетный файл"). Появится черное окно консоли. Не волнуйтесь, если придется немножко подождать: это система безопасности Windows и антивирус производят свои проверки. Возможно, вам придется успокоить ее, уведомив, что создатель файла вам известен, и запускать этот файл можно. Для тех, кто умеет, рекомендуется внести каталог `masm 6.14` в список исключений для проверки антивирусом. Вскоре сверху окошка появится заголовок, а внутри окошка вы будете видеть работу программы.

Рекомендуется сразу войти в свойства окна (щёлкнув ПРАВОЙ кнопкой мышки на его заголовке), и установить достаточно большой размер шрифта для комфортной работы с экраном.

3. Если что-то не получилось. Проверьте, разархивирована ли папка (Windows может вам показывать, какие файлы находятся в сжатой папке, такие файлы можно только просматривать, работать с ними нельзя). Проверьте, правильное ли у Вас название главной папки, не пропущен ли пробел или точка.

Теперь Вы можете посматривать тексты программ и наблюдать, как они работают. Каждая программа-пример находится в отдельной папке. Текст ассемблерной программы находится в файле `a.asm`, посмотреть (и изменить) его можно любым редактором, например, блокнотом. Лучше сразу настроить компьютер, чтобы файлы с расширением `.asm` по щелчку мыши (а лучше по клавише `ENTER`) открывались блокнотом. В противном случае надо открывать правой кнопкой мыши «открыть с помощью» и выбирать блокнот. В популярных файловых менеджерах Far Commander и Total Commander для входа в редактор можно использовать клавишу `F4`. Можно попробовать редактор `qeditor.exe` из папки `masm 6.14`.

4. Написать свою программу. В папке `_Examples` есть подпапка `_Шаблон`, там содержится все необходимое для изготовления и запуска собственной программы. Первым делом надо продублировать эту папку, чтобы не испортить и использовать в дальнейшем. Копию (например, с именем `_MyProg1`)

надо поместить в той же папке (_Examples). Теперь открываем файл a.asm, в нем находится заготовка для программы: уже написан комментарий (после слова COMMENT) и есть макрокоманда, печатающая заголовок консольного окна (ConsoleTitle "Заготовка программы"). А далее, перед словом exit – место для наших команд. Измените текст заголовка, запишите туда свою фамилию и что делает программа (преподавателю будет удобно проверять такую программу). Далее наберите макрокоманду печати строки:

**outstrln** "Здравствуй, мир"

Можно и более "красиво", написав макрокоманду вывода с большими буквами:

**OutstrLn** "Здравствуй, мир".

Затем сохраним изменённый файл и запустим нашу программу с помощью файла makeit.bat. В окне консоли должен появиться написанный Ваш текст. Поздравляем! Вы написали первую программу на Ассемблере!

Подобным образом можно писать и другие программы: создавать копию папки _Шаблон и в ней писать программу в файле a.asm. Вы можете добавить вывод несколько строчек разноцветного текста, которые будут появляться в разных местах экрана по нажатию клавиши, поэкспериментировав с макрокомандами, в которых не нужны переменные и регистры: **newline**, **outstr** и **outstrln**, **GotoXY**, **SetTextAttr**. Обратите внимание, что по синтаксису Ассемблера в именах макрокоманд большие и маленькие буквы различаются, так что писать команды надо строго теми буквами, какими определено. Правда, некоторые имена макрокоманд имеют СИНОНИМЫ, например **outint**, **Outint**. Описание команд см. [1, раздел 6.5.1].

Чтобы свободнее ориентироваться в шаблоне программы, разберем остальные его составляющие.

Программа на Ассемблере состоит из строчек-предложений. В каждой строке обычно помещается одна команда, описание переменной или директива. Для наглядности группы строчек можно писать со смещением на 2-3 позиции. Как и в Паскале, везде, где стоит пробел или любой другой разделитель (скобка, запятая), можно поставить любое количество пробелов.

Первая строка нашей программы **include** concole.inc подключает крайне нужный файл (в частности, там находятся используемые нами макрокоманды). Эта строка должна быть всегда и самой первой. Далее идёт многострочный комментарий. Он начинается служебным словом **COMMENT**, а вместо звездочки за ним можно ставить любой символ. Этот символ задаёт начало и конец комментария. Таким образом, оформляются длинные комментарии (в несколько строк), короткие комментарии, пояснения к одной строке, можно писать сразу после строки через точку с запятой.

Наша программа состоит из двух секций с заголовками **.data** и **.code** – эти названия менять нельзя. Понятно, что это секции для данных и кода программы (команд). Работа программы начнется с выполнения команды, помеченной меткой Start. А откуда машина узнает, что именно эта метка – самая главная (она не обязана стоять в первой строке секции кода)? Ассемблер узнает это, "посмотрев" на последнюю строчку программы. Там стоит слово **end** и после него указывается та самая "главная" метка. Метку можно называть как угодно (по правилам формирования имён), только надо помнить, что большие и маленькие буквы для имен пользователя в Ассемблере различаются. Макрокоманда **exit** завершает программу.

Собственные программы можно хранить и запускать и по-другому, но для этого надо немножко разобраться, что за файлы у нас в папке, что они делают, как с ними можно работать.

Мы уже знаем, что в файле с расширением .asm находится текст программы на Ассемблере. Чтобы программа заработала, надо перевести этот текст на машинный язык, сохранить полученный результат в так называемом исполняемом файле, а потом запустить данный файл на счёт. При работе с Паскаль-программой надо было произвести, в общем-то, такие же действия, но там у нас была среда, которая сама делала эту работу, вызывала нужные системные программы. На Ассемблере у нас для этого есть специальный пакетный файл makeit.bat.[1] В этом файле собраны команды, которые должен выполнить

[1] В редакторе qeditor.exe для запуска программы на счёт есть пункт меню makeit.bat в разделе Project.

компьютер (точнее, его системная программа – командный интерпретатор). Давайте посмотрим, как этот файл устроен. Открыть этот файл для просмотра можно по правой кнопке мыши командой "изменить" (команда "открыть" производит выполнение записанных в файле команд) или кнопкой F4 в файловом менеджере.

Первая строка файла читается как "выключить эхо" и запрещает вывод на экран протокола работы пакетного файла. Далее команда **set** Name=a указывает, какое имя имеет файл с программой на Ассемблере, у нас это имя "a". Именно поэтому наш файл на Ассемблере обязан называться "a.asm". Теперь вы можете назвать его по-другому, одновременно вписав это имя в строку пакетного файла. Не советуем давать файлам длинные имена, содержащие какие-либо символы кроме букв и цифр.

В следующих строках указываются пути к нужным для работы Ассемблера библиотекам, их менять нельзя. Главные команды, ради которых всё и затеяно, это команды с именами ml и Link, которые задают трансляцию и сборку выполняемой программы (см. [1]). Эти команды вызываются с параметрами, о которых, вы, узнаете позже, так что пока в этих строчках ничего менять нельзя.

Элементарные знания английского языка позволяют догадаться, что если при выполнении каждой из программ произошла ошибка, происходит переход на печать сообщения об этом и завершение работы. Если ошибки не произошло, программа ml создает файлы a.obj и a.lst. Файл с расширением lst – это листинг (протокол работы программы-переводчика), хороший помощник программиста, а файл с расширением obj нужен программе Link. Она его обрабатывает и создает файл a.exe, который и является исполняемым файлом. Посмотрите в Вашей папке _MyProg1 – там теперь присутствуют все эти файлы. А пакетный файл завершает свою работу, запуская на выполнение получившийся файл a.exe. Кстати, вы тоже можете это сделать, дважды кликнув по нему мышкой.

Таким образом, если вам хочется держать в одной папке несколько ассемблерных программ с разными именами, можно для выполнения каждой сделать свой пакетный файл (вписав в него ее имя), а можно использовать один файл, меняя в нем имя программы. Сразу хочу предупредить, что очень много файлов в одной папке держать не следует: в ней будет трудно ориентироваться, так как для каждого исходного ассемблерного файла создается еще ТРИ файла (.obj, .lst и .exe).

## *2. Работа в командной строке*

Командная строка предоставляет ещё один способ выполнять ассемблерные программы. Возможно, кому-то он пригодится (может быть, не сейчас, а к концу семестра). Кто чувствует себя не уверенно, этот раздел можно пока пропустить.

При работе в командной строке у вас не будет для выполнения каждой программы открываться новое окошко, все будет делаться в одном. Но сначала это окошко надо вызвать. Отыщите на своем компьютере программу "командная строка" ("все программы" – "стандартные" или воспользуйтесь поиском). На экране появляется черное окошко с надписью о том, что у вас работает Windows и так называемое "приглашение командной строки" – сообщение системы о том, что она готова к работе и ждет от вас команды.

Этот способ общения появился на заре компьютерной эры, машины тогда были немногословны, так что приглашение выглядит как значок ">". Символы перед ним указывают папку (при работе с командной строкой ее по старинке принято называть "директория"), в которой может вестись работа.

Сначала нам надо "сменить директорию", т.е. перейти в ту папку, в которой находятся наши файлы. Команда смены директории называется cd (Change Directory). Чтобы подняться на уровень вверх, надо набрать `cd..` Для подъёма на два уровня можно набрать команду 2 раза или объединить в одной команде `cd..\..`). Чтобы перейти на другой диск (это понадобится, если файлы находятся на флешке), надо набрать его имя и двоеточие (посмотрите "мой компьютер", какое имя дано вашей флешке), например F:. Чтобы спуститься в нужную директорию, надо набрать cd <имя директории>.

Можно все эти команды собрать воедино, выписав путь к нужной директории в одной строке, но обычно легче выполнять команды по очереди, так как после каждой команды меняется приглашение, вы видите, в какой директории оказались. Иногда полезна команда dir – она показывает содержимое те-

кущего каталога. Особо продвинутые могут собрать нужные команды для смены директории в собственный пакетный файл (только расположить его надо в правильной папке).

Можно сразу открыть окно "командная строка" в нужной папке. Для этого надо навести мышку на нужную папку и, удерживая клавишу SHIFT, нажать правую кнопку мыши. В появившемся меню выбрать пункт "Открыть окно команд".

Итак, в результате этой деятельности перед нами черный экран консоли, где выписан путь к папке с программой на Ассемблере и приглашение к вводу, например:

```
c:\masm 6.14\_Exampes\_Myprog1>
```

В этой строке можно набирать команды, они будут выполняться. Собственно, мы уже набирали команду cd. Исполняемый файл с расширением exe, который получился в результате трансляции ассемблерной программы, также является командой. Если набрать его имя в этой строке (без расширения), он выполнится. Так как окно не исчезнет, результат работы не пропадет. Пакетный файл также рассматривается как команда, так что, если набрать его имя (можно и без расширения), выполнятся содержащиеся в нем команды.

Однако, просто набрав имя имеющегося у нас пакетного файла, мы ничего не приобретем, для запуска разных программ также придется менять файл. Можно изменить пакетный файл один раз, настроив его для запуска любой программы. Для этого надо в нем изменить вторую строку, где прописывается имя файла, на строку

```
set Name=%1
```

Теперь пакетный файл надо запускать с параметром, например, если программа на Ассемблере содержится в файле prog1.asm, надо написать

```
makeit prog1
```

Следует учесть, что измененный таким образом пакетный файл нельзя запускать "старым способом", кликнув по нему мышью, так как имя программы в нем не указано.

Работа в командной строке расширяет наши возможности – можно легко работать в одном окне с файлами из разных папок (перед именем файла пишем путь к нему). При работе в командной строке удобно пользоваться клавишами "стрелки". Они позволяют быстро продублировать в новой строке команды, которые набирались ранее. Таким образом, можно дублировать старые команды, меняя в них, например, название файла.

## 2. Представление данных

Вспомним, что мы ещё в школе узнали: вся информация в памяти компьютера хранится в двоичном виде. Числа и символы, да и сами команды, имеют одинаковое двоичное представление, "внутри" они неразличимы. Каждому символу сопоставляется число размером в байт – его код. В памяти компьютера хранится двоичное представление этого числа. В программе на Ассемблере записать символ можно как в виде символа в апострофах, так и в виде числа.

Целые числа могут иметь размер в байт, слово (2 байта), двойное слово (4 байта) и четверное слово (8 байт). Положительные числа представляются в двоичной системе счисления в прямом коде, отрицательные – в дополнительном коде [1, п.5.5]. Таким образом, исследуя содержимого некоторого байта, невозможно определить, находится в нем символ, положительное или отрицательное число, или это только часть какого-то большого (например, четырехбайтового числа). Как "поймёт" это представление компьютер зависит от программиста. Например, байт $11001000_2$ можно трактовать как беззнаковое число 200, как знаковое число –56 или как символ русской буквы 'И'.

В качестве примера рассмотрим программу, которая будет выводить числа и символы с помощью макрокоманд вывода. Почему такое название "макрокоманды"? Это понятие мы будем изучать ближе к концу курса. Пока лишь скажем, что, так как в Ассемблере (в отличие от языков высокого уровня) нет хороших команд ввода и вывода, для Вас специально подготовлено несколько наборов команд для ввода и вывода разного вида данных. Вот эти-то наборы и вызываются макрокомандами. Имена макрокомандам дал их автор [1], напомним, что в именах большие и маленькие буквы различаются.

Будем использовать следующие макрокоманды вывода [1, разд. 6.5.1]

**outchar** <операнд>

Операндом может быть символ, переменная или число размером в один байт. Если операнд – символ, он выводится на экран, если операнд – число, на экран выводится символ с соответствующим кодом в алфавите. Аналогично трактуется значение переменной.

**outint** <операнд> и **outword** <операнд>

Как уже говорилось, у некоторых макрокоманд могут быть и "красивые" синонимы, например, **Outint**. Операндом может быть число или переменная любого допустимого размера. Операнд трактуется как десятичное число (знаковое для **outint** и беззнаковое для **outword**) и выводится на экран. Операндом может быть и символ. Напомним, что символ для Ассемблера – его код, то есть число размером в байт. Таким образом, если, например, выводить этой командой символ `'A'`, получим его код, т.е. `ord('A')`.

У этих команд есть модификации. Во-первых, можно в конце приписать **ln**, тогда после вывода будет переведена строка (по аналогии с Паскалем). Во-вторых, можно через запятую указать второй необязательный операнд размером в байт, он будет определять количество позиций, отводимых под число на экране.

Внимание! Этими командами можно выводить только ОДНО число. Команда **outword** 12,3 понимается как вывод числа 12 в трех позициях на экране. Для вывода нескольких чисел надо использовать несколько макрокоманд или выполнять макрокоманду в цикле.

Ну и для самых "продвинутых": можно использовать вывод с текстом-пояснением. В этих командах существует третий (необязательный) операнд, строка в кавычках, которая выводится на экран перед числом. Например, если A=52, макрокоманда **outint** `A,4,'A='` выведет `A=  52`.

Советуем аккуратно пользоваться этим дополнительными операндами в этих командах. Лучше, пока вы не чувствуете себя "продвинутым", выводить все по отдельности, использовать команды с одним операндом. Дело в том, что, так как Ассемблер может трактовать буквы как числа, если вы перепутаете порядок операндов, ошибка зафиксирована не будет, но выведено будет «неизвестно что». Также ни в коем случае нельзя опускать второй операнд. Макрокоманда **outint** `A,'A='` зафиксирует ошибку, так как текст задан на месте длины поля вывода, надо писать **outint** `A,,'A='`.

Чтобы покончить с командами вывода, напомним о макрокомандах вывода строки **outstr** и **outstrln**. Операндом является текст в апострофах или в двойных кавычках. Для перевода строки используется макрокоманда **newline** (аналог `WriteLn` в Паскале).

Итак, обещанная программа

```
include console.inc
comment *
   Программа показывает, что одно и то же число разные
   макрокоманды вывода будут выводить по-разному
*
.code
Start:
  ConsoleTitle "  Вывод чисел"
  outcharln 200
  outcharln 'И'
  outwordln 'И'
  outintln  'И'
  exit
  end Start
```

Исследуем выведенное. Первая макрокоманда трактует число 200 как код символа и выводит букву 'И'. Следующая макрокоманда также трактует `'И'` как код символа и выводит ту же букву. А вот две последние макрокоманды трактует букву 'И' как целое (без знака и сл знаком), и, соответственно, выводят числа `200` и `-56`.

Если мы используем команды

```
outintln  -100
outwordln -100
```
то получим знаковое число –100 и беззнаковое (4-байтовое) число 4294967196.

## *4. Резервирование памяти. Ввод и вывод данных*

Напишем программу, которая позволит нам убедиться, что числа в памяти компьютера хранятся именно так, как об этом говорится в [1].

В секции **.data** зарезервируем память для хранения нескольких чисел, то есть, в терминах Паскаля, опишем несколько переменных. Какие имена можно дать переменным? Имя должно начинаться с буквы (латинской), может состоять из букв, цифр и символов из набора @ _ $ ? . Напомним, что большие и маленькие буквы в именах различаются. Имена не должно совпадать со служебными словами и названиями макрокоманд. В частности, нельзя использовать имена Proc, End, Inc, DB, DD, AX, Al, Ah – всего не перечислить, но потихоньку вы все эти ограничения освоите.

Главное в директиве резервирования памяти не имя переменной, а размер выделяемой памяти. Он как раз директивой и определяется. ВОТ ГЛАВНЫЕ ДИРЕКТИВЫ: **db**, **dw**, **dd**, **dq**. Они резервируют место в памяти размером соответственно в 1, 2, 4, 8 байт. Отметим, что это служебные слова, их можно писать любыми буквами, и большими и маленькими. Запомнить названия директив несложно: первая буква от слова **d**efine, вторая – от **b**yte, **w**ord, **d**ouble word и так далее [1, разд. 6.4].

В общем, виде директива резервирования памяти выглядит так

[<имя переменной>] <директива> <операнды> [; <комментарий>]

Имя переменной и комментарий являются необязательными. Операндов может быть один или несколько, если операндов несколько, они отделяются друг от друга запятыми.

Что может быть операндом директивы резервирования памяти?

1. Операнд ?.

Пример `A db ?`; описывается переменная A размером в байт. Переменная не получает начального значения, то есть в ней будет неизвестно что, мусор.

2. Операнд число.

Пример `B dw 500`; АНАЛОГ НА Free Pascal `var B: integer=500;`. Число должно умещаться в отведенную для него память (иначе будет ошибка): в байт можно поместить число от -128 до 255, в слово – от -32768 до 65535, дальше посчитайте сами. Число может быть записано в разных системах счисления: двоичной, восьмеричной, десятичной и шестнадцатеричной. Для определения системы счисления к числу в конце приписывается соответственно b (**b**inary), o (**o**ctal), d (**d**ecimal) и h (**h**ex). Букву d в конце десятичного числа можно не указывать, а в восьмеричном числе вместо o можно писать q (чтобы не путалось с нулем). Буквы могут быть большие и маленькие, но с маленькими буквами число смотрится понятнее.

Небольшое замечание о шестнадцатеричных числах. Как вы помните, в таких числах могут встречаться латинские буквы A–F (можно использовать и большие, и маленькие буквы, но большие смотрятся красивее). Если число начинается с буквы, возникает путаница: понимать запись как число или как имя (например, A2h). Чтобы ликвидировать эту неоднозначность, договорились в таком случае к числам добавлять впереди ноль: A2h – имя, 0A2h – число.

3. Операнд – символ (в кавычках или апострофах).

Пример `Letter db 'a'`; это `var Letter: char='a';`. Этот операнд отличается от операнда <число> только формой записи. Ассемблер понимает символ в кавычках как его код, то есть ord(<символ>). Для переменной размером в N байт можно в кавычках записать N символов, например, `c4 dd 'КАША'`, но обычно так не делают, так как символ хранится в байте и для работы с несколькими символами удобнее использовать массив байтов. В директиве резервирования памяти может быть несколько операндов, но об этом позже.

Заведем несколько переменных разного размера, придадим им начальные значения в виде чисел в разных системах счисления, не забудем про символы в конце не десятичных чисел. Итак, обещанная программа

```
include console.inc
comment *
        Вывод чисел разного вида
*
.data
B  db 200
W1 dw 200
W2 dw 32769
Q  db 0ah
F  dw 11b
L  db 'a'
.code
Numbers:
  ClrScr
  ConsoleTitle "  Числа разного вида"
  outint    B,8
  outwordln B,8
  outint    W1,12
  outwordln W1,12
  outint    W2,12
  outwordln W2,12
  outwordln Q
  outwordln F
  outchar   L
  outwordln L,5
  exit
  end Numbers
```

Смотрим, что получилось: переменная `B` – байтовая, число 200, представленное в ней, трактуется поразному в знаковом и в беззнаковом виде. А переменная `W1` имеет то же значение `200`, но размер ее – слово, поэтому в знаковом и беззнаковом виде она выглядит одинаково. Подробнее о знаковых и беззнаковых числах [1, разд. 5.5].

Также видим, что числа в разных системах счисления напечатались в десятичном виде, а символ можно вывести как в виде символа, так и в виде кода.

Если любопытно, можно посмотреть макрокоманду **`outnum`**, которая умеет выводить числа в 16-ной и 2-ной системе счисления.

## *5. Распределение памяти*

Когда мы описываем переменные в программе на Паскале, компилятор сам как-то располагает их в памяти. На Ассемблере все четко определено. То есть конкретные адреса мы предугадать не можем, но взаимное расположение данных известно точно: они находятся в памяти подряд друг за другом без всяких промежутков точно в том порядке, в каком описаны. В нашей программе адрес переменной `W1` будет на 1 больше адреса `B` (так как `B` занимает один байт), а адрес `W2` на 2 больше адреса `W1` (так как `W1` занимает 2 байта). Убедиться в этом можно, напечатав значение адреса, например:

```
outwordln offset W1; offset W1 – это адрес W1
```

Можно также загрузить адрес в регистр (например, `EAX`):

```
mov eax,offset W1
```

или

```
lea eax,W1; eax:=<адрес W1>
```

Об этой команде, о регистрах – позже, сейчас просто можно этими командами посмотреть, чему равны адреса наших переменных. Теперь можно печатать значение адресов и из регистров, например

```
lea eax,B; записывает в регистр eax адрес B
outwordln eax; печатает содержимое регистра
```

и так далее.

Так как вместо имени переменной Ассемблера подставляет ее адрес, то есть число, то можно обращаться к переменной (то есть к месту в памяти с определенным адресом) не по её имени, а по имени соседних переменных со смещением. Так, по адресу `B+1` из последней программы располагается переменная `W1`, а по адресам `W1+2` и `B+5` – переменная `W2`. Учитывая взаимные расположения переменных в памяти, можно печатать их значения, указывая их адреса, например, выполнив макрокоманду

```
outwordln W2-2
```

получим число 200. А что находится по адресу `W1-1`? Напечатаем

```
outwordln W1-1
```

Получается совсем не 200. Почему так? Дело в том, что, выполняя резервирование памяти, Ассемблер не только отводит место в памяти и запоминает адрес. С именем переменной также связывается тип данных (и их длина). Переменная `B` у нас типа **byte**, а переменная `W1` – типа **word**. Поэтому макрокоманда видит имя `W1` и берет по адресу `W1-1` слово, то есть 2 байта!

Посмотрим, что находится в этих байтах. В первом 200 и во втором тоже 200 (перевёрнутая запись!). Что получается? В позиционной системе счисления по основанию 256 это `200*256+200=51400`. Именно это число и будет напечатано. А можно представить число 200 в двоичной системе, выписать содержимое слова: `11001000 11001000` и перевести это беззнаковое число в десятичную систему, опять получится 51400. Сами разберитесь, что напечатает макрокоманда

```
outintln W1-1
```

Таким образом, имя переменной определяет не только её адрес (т.е. "откуда брать"), но и тип, размер числа (т.е. "сколько байт брать"). Кстати, тип переменной можно узнать с помощью оператора **type**. Можно напечатать

```
OutwordLn type A
OutwordLn type F
```

Получим 1 и 2, так как тип считается в байтах. Кстати, здесь уместно вспомнить, что компьютер хранит числа в "перевернутом" виде: сначала располагаются младшие байты, потом старшие [1, п.5.6,; 3, 1.3.1, с.12-14].

Посчитаем, что находится в байтах с адресами `Q-1` и `Q-2`, а потом проверим себя, напечатав

```
outwordln Q-1
outwordln Q-2
```

`Q-1=W2=32769=32768+1=`$2^{15}$`+1`, в двоичной системе это `10000000 00000001` – число специально разделено на две части, это содержимое байтов. Вспомним, что в ЭВМ числа хранятся в перевернутом виде, сначала располагаются младшие байты, а потом старшие. Поэтому по адресу `Q-1` находится старший байт `W2` (это число 128), а по адресу `Q-2` – младший байт `W2` (число 1).

Байты числа `W2` можно напечатать и без помощи переменной `Q`, можно использовать и адрес `W2`, вот только тип надо изменить. Для этого используется оператор **ptr**. Перед этим словом ставится название типа:

```
outwordln byte ptr W2
outwordln byte ptr W2+1
```

Напечатается, естественно, то же самое. Посмотрим теперь, как хранится в памяти "длинное" число размером в 4 байта. Опишем его `Q` **dq** `01020304h` и распечатаем содержимое байтов

```
OutWordln byte ptr Q
OutWordln byte ptr Q+1   и так далее
```

Увидим, что по адресу Q находится число 4, по адресу Q+1 – число 3, а далее числа 2 и 1. При переносе числа из памяти в регистр (при выполнении арифметических операций) компьютер сам "переворачивает" число, так что о "перевернутом" представлении чисел надо помнить, только если зачем-то хочется работать с частью числа или посмотреть значение числа в листинге программы.

Таким образом, следует усвоить, что в памяти компьютера лежат не числа, которые мы указали в директивах Ассемблера, там лежат нолики и единички. Когда мы хотим работать с содержимым памяти, мы может просто указать адрес и количество байтов, которое нам нужно, начиная с этого адреса.

Про директивы с несколькими операндами. В директиве может быть несколько операндов, они перечисляются через запятую. Это следует понимать как отведение памяти для нескольких переменных, причём имя имеет только первая из них. К остальным можно обращаться, прибавляя к адресу смещение. Например, одни и те же байтовые переменные можно описать по-разному.

```
Z db 'a','b','c','d'
Z db 'abcd'
Z db 'a'
  db 'b'
  db 'c'
  db 'd'
```

Во всех этих фрагментах резервируется 4 байта памяти, первый из них имеет имя Z, остальные безымянные, к ним можно обращаться Z+1, Z+2, Z+3.

Директива `Y dd 1,2,3` резервирует место для трех переменных, каждая размером в двойное слово (4 байта). По адресу Y находится число 1, число 2 находится по адресу Y+4, число 3 по адресу Y+8.

Несколько значений одного типа, располагающихся в памяти подряд, можно считать массивом. В Ассемблере нет никакого специального служебного слова для описания массива, нет и типа с таким названием. Тип переменной, которая дает массиву имя, определяется директивой резервирования памяти и никак не зависит от длины массива. Собственно никакой длины массива нет, ассемблер ее никаким образом "узнать" не может, никаких запретов для обращения к любому участку памяти по любому существующему имени. Для компактного задания массива из, например, 100 элементов используется директива

```
X dd 100 dup (?)
```

Она предписывает зарезервировать в памяти (подряд) 100 переменных размером в двойное слово, первая из этих переменных будет иметь имя X.

## 6. Регистры

Для работы доступны 8 32-битных регистров со служебными именами EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP. При желании можно использовать только младшие 16 бит каждого регистра, при этом надо писать названия без буквы E, например AX, SI. Младшие части первых четырех регистров (тех, которые с буквой X в конце) еще раз делятся пополам, давая нам возможность работать с байтами. Названия байтовых регистров формируются так: первая буква от названия регистра (A, B, C или D), а вторая L или H. Буква L означает low (нижний), так как это самая младшая часть регистра, младшие разряды с 0-ого по 7-ой, а буква H – это hight (верхний), старшая половинка слова, разряды с 8-ого по 16-ый. Например, AL – это один (младший) байт из четырехбайтового регистра EAX, а DH – кусочек регистра EDX. При работе с регистрами надо понимать, что, меняя "кусочек", мы меняем и весь регистр и наоборот. Например, если после действий AL:=1 и EAX:=0, то в регистре AL будет 0, а после действий EAX:=0; AH:=2; AL:=1 в регистре EAX ,будет 0000 0000 0000 0000 0000 0010 0000 0001, это число 513

## 7. Команды Ассемблера

Каждая команда обычно пишется на отдельной строчке (допускается и приветствуется смещение некоторых команд вправо). Команда может быть помечена, после метки ставится двоеточие. Команда состоит из мнемокода (несколько латинских букв, регистр букв значения не имеет) и операндов (обычно 1 или 2, бывают команды без операндов). После команды на той же строке может быть комментарий (через точку с запятой).

### 7.1. Команда mov

Это команда присваивания. У нее два операнда. Общий вид

```
mov op1,op2
```

перед командой может стоять метка, в конце – комментарий. Команда выполняет действие op1:=op2. Исходя из ее смысла, на операнды накладываются следующие ограничения:

Оба операнда не могут быть из памяти (в ассемблере вообще нет команд формата память-память, у которых два явных операнда из памяти). Операнды должны быть одного типа (для работы с операндами разных типов существуют другие команды). Первый операнд не может быть непосредственным (т.е. числом).

Наш 32-хбитный Ассемблер работает с байтами, словами и двойными словами, то есть четверным словом (**dq**) никакой операнд быть не может. Примеры

```
mov ebx,eax; ebx:=eax
mov PER,al:  команда правильная, если описано  PER db ?
mov eax,bl;  неверная команда, разный тип операндов
```

Отдельно скажем про команду **mov** с непосредственным операндом (т.е. второй операнд – число), ведь у числа типа нет. Число должно быть таким, чтобы помещаться в первый операнд. Примеры

```
mov eax,500; eax:=500
mov eax,'И'; eax:=200 (код буквы 'И')
mov al,300;  ошибка, 300 в байт не помещается
```

Команда **mov** флаги не меняет.

### 7.2. Команда xchg

```
xchg op1,op2
```

Меняет значения операндов местами. Они должны быть одного типа, ни один не может быть числом, нельзя, чтобы оба операнда одновременно были из памяти. Флаги команда не меняет. Пример

```
mov  al,1
mov  ah,2
outwordln ax; ax из байтов 00000010 и 00000001 – число 513
xchg al,ah
outwordln ax; байты поменялись местами, стало число 258
```

## *8. Отладка программы. Листинг*

А что, если программа не работает? Увы, такое случается часто. Если программа не запустилась на счет, в окне консоли появляется сообщение и информация об ошибке. Эту информацию следует прочитать, хоть она и по-английски. Если написано, что не найден файл с расширением .asm, значит в bat-файле неправильно указано имя программы на ассемблере, или программа находится не в той папке. Если не найден еще какой-то файл, скорее всего что-то напутано с размещением программы.

Более частые ошибки – синтаксические. Давайте специально сделаем такую ошибку и посмотрим реакцию компьютера. Напишем неправильную директиву

```
A db 300; число 300 в байт не помещается
```

Запустим программу. Увидим, что производилось "ассемблирование" нашего файла (assembling). А дальше выписано, имя файла, и в какой строке замечена ошибка a.asm(xxx), номер ошибки и ее расшифровка. Номер ошибки нам не нужен (вообще-то есть список всех ошибок, в котором можно, зная номер, про каждую из них прочитать), обычно из описания ошибки можно понять, в чем дело. Для нашего случая будет сообщение

```
a.asm(xxx) : error A2070: invalid instruction operand
```

В нашем случае сообщается, что значение операнда недопустимое (слишком большое для данного типа). Если программа небольшая, можно сразу искать неправильную строку в тексте. Место строки с ошибкой можно посмотреть в листинге – это файл с тем же именем, что и исходный файл на ассемблере,

с расширением `lst`. Он создается ассемблером при трансляции программы, в нем программа представлена как в исходном виде, так и в "переводе на машинный язык" – в виде чисел в 16-ричной системе счисления.

Посмотрим на строчки кода, соответствующие нашим директивам. Самая левая колонка – адрес ячейки. Правда, это не абсолютный адрес, по которому данная строка располагается в памяти компьютера, это смещение от начала секции (в нашем случае секции **`.code`**). Все остальное – совершенно "честно". Мы можем видеть, что адреса соседних переменных отличаются ровно на размер переменной (в байтах). После адреса в строке листинга записано содержимое байтов – перевод строки кода. Для директив это представление числа в 16-ричной системе (числа в компьютере, конечно же, представляются в двоичной системе, листинге все переведено в 16-ричную, так как такой текст удобнее читать). В нашей программе пока нет никаких команд (макрокоманды в нашем листинге не отображаются), но как только вы будете писать программы с командами – посмотрите и убедитесь, что в переводе на машинный язык они ничем не отличаются от данных.

Давайте теперь сделаем ошибку в макрокоманде, напишем в ней операнд – неописанную переменную. Ассемблер нам показывает сразу несколько ошибок "неописанная переменная". В листинге видим, что после неправильной макрокоманды появилось несколько строчек с этой переменной, и в них всех ассемблер отмечает ошибку. Дело в том, что в макрокоманде переменная используется несколько раз, вот ассемблер и выписывает все неправильные строки из макрокоманды. Исправлять надо, конечно же, саму макрокоманду. Итак, увидев сообщение об ошибке, смотрим в листинге, в какой строке она обнаружена, исправляем ошибку (исправлять надо, не в листинге, а в своем файле с расширением `.asm`). И не забудьте сохранить исправления!!!

# *9. Арифметические команды*

## 9.1. Сложение и вычитание

Команды сложения (**`add`**) и вычитания (**`sub`**) в чём-то похожи на команду **`mov`**: два операнда, результат кладется в первый. Работает с байтами, словами и двойными словами, то есть четверным словом (**`dq`**) никакой операнд быть не может (для этого надо перейти в 64-хбитный режим работы)

```
add al,5;     AL:=AL+5
sub ebx,ecx; ebx:=ebx-ecx
```

Операнды должны быть одного типа, но результат при этом не обязан помещаться в первый операнд. Если в регистре `AL` лежит число 250, то при выполнении команды **`add`** `al,10` в регистре `AL` окажется число 4. О том, как выполняется сложение и вычитание, как вырабатываются флаги, читайте [1, п. 5.5].

Команды **`inc`** и **`dec`** с одним операндом служат для прибавления или вычитания единицы. Они работают аналогично командам сложения и вычитания со вторым операндом 1 (первый операнд – регистр или память), но у них есть некоторое отличие: они не меняют флаг переноса `CF`.

Команда изменения знака **`neg`**. Один операнд – регистр или память. Меняет знак числа. Особый случай – если значение аргумента самое маленькое знаковое число для данного типа, например, –128 для байта или –32768 для слова (то есть в двоичном виде сначала 1, а все остальные нули). Числа с обратным знаком для этого случая не существует, операнд не меняется. Команда меняет флаги. Подробнее [3, стр. 51].

Команды **`adc`** и **`sbb`** работают точно так же, как **`add`** и **`sub`**, но еще добавляют (вычитают) значение флага `CF`. Подробнее [3, стр. 52].

Использование команды **`adc`**.

Один из случаев, когда используется эта команда – сложение целых чисел по частям. Например, если надо сложить "длинные" числа размера, например, 8 байт. Такие числа можно хранить в памяти, но команд для сложения и умножения для них нет. В этом случае надо складывать сначала четырехбайтовые "хвосты" чисел, а потом командой **`adc`** четырехбайтовые "головы".

Можно использовать команду для сложения "некратных" чисел. У нас есть возможность работать с однобайтовыми, двухбайтовыми и четырехбайтовыми числами. А если задача такова, что удобно ис-

пользовать трехбайтовые числа? В этом случае надо складывать сначала два правых байта числа, а потом с помощью **adc** – первые (левые) байты. После сложения надо поместить результат по правильным адресам.

Сложение делается аналогично. Таким же образом (по частям) делается умножение – см. [3, стр. 62]. Далее приведена программа для сложения трехбайтовых чисел. Так как трехбайтовые числа сами по себе не вывести, для проверки используется вывод четырехбайтового числа (для этого после чисел добавляется дополнительный нолик). Сразу отметим, что с этим дополнительным ноликом можно было не мучиться и складывать **dd**-числа обычной командой сложения **add**, не разделяя их на части. Сложение трехбайтовых чисел по предложенному алгоритму нужно, когда числа хранятся подряд, без дополнительных ноликов между ними (экономия памяти!)

```
include console.inc
.data
a  db 0ffh,0ffh,0h; это 65536
   db 0
;  промежуток в памяти между числами для удобной печати
b  db 1,0,1
   db 0
m  dd 0
s  db 0,0,0
   db 0
.code
Start:
   ClrScr
   ConsoleTitle "   Сложение трехбайтовых чисел"
Begin:
   outstr 'Трехбайтовое число первое'
   outwordln m-8; m-8 это a
   outstr 'Трехбайтовое число второе'
   outwordln m-4; m-4 это b
   mov ax,word ptr a
   add ax,word ptr b
   mov word ptr s,ax
   mov al,a+2
   adc al,b+2
   mov s+2,al
   outwordln m+4,,'Сумма='
   exit
   end Start
```

Далее приведена программа сложения восьмибайтных чисел. Показано, как восьмибайтные числа отображаются на экране.

```
include console.inc
.data
X  dq  11234567899
Y  dq  16712111001
Z  dq  ?
.code
Start:
   ClrScr
   ConsoleTitle "       Сложение длинных чисел"
   outwordln X,,'X='
   outwordln  Y,,'Y='
```

```
  mov eax,dword ptr X
  add eax,dword ptr Y
  mov dword ptr Z,eax
  mov eax,dword ptr X+4
  adc eax,dword ptr Y+4
  mov dword ptr Z+4,eax
  newline 2
  outwordln Z,,'Z='
  exit
  end Start
```

Для проверки работы и лучшего понимания программ рекомендуется распечатать промежуточные результаты, сложить числа, которые не формируют и которые формируют флаг переноса.

## *10. Команды перехода*

Для выполнения перехода чаще всего нужна помеченная команда. Метка (имя с двоеточием) ставится перед командой (можно на той же строчке, можно и на предыдущей). Следует иметь в виду, что метки являются именами, большие и маленькие буквы в них различаются, метки не должны совпадать со служебными словами. Безусловный переход

**jmp** <метка>; **jmp** – от слова «прыгать»

Условные переходы чаще всего связаны со значениями флагов. Команда перехода по флагу=1 состоит из двух букв **j** (от jump) и первой буквы из названия флага: **jo**, **jc**, **jz**, **js**. Команды перехода по флагу в случае его равенства нулю состоят из трех букв **j**, **n** (not) и буквы флага, например, **jno**. При значении флага, обозначенного в названии команды, происходит переход по указанной метке, в противном случае выполняется команда, следующая за командой перехода.

Еще есть команда перехода по значению регистра ECX. Она записывается **jecxz** <метка>, переход происходит, если значение регистра ECX=0.

Следует понимать, что какое-то значение у флага есть всегда (возможно, это "мусорное" значение, данное ему до выполнения Вашей программы). Команда перехода может стоять как непосредственно после команды, которая установила использующийся в ней флаг, так и через несколько команд, если они флаги не меняют.

Из изученных нами команд флаги меняют арифметические команды, не меняют команда **mov** и сами команды перехода. Макрокоманды ввода-вывода (кроме **inint**) флаги не меняют.

Чтобы посмотреть, как работают команды перехода по флагам, напишем программу, которая будет складывать байтовые числа и печатать, правильный ли получился результат для беззнаковых чисел (помещается ли он в байт), то есть будет проверять флаг CF.

```
include Console.inc
ConsoleTitle "  Сложение с проверкой"
Begin:
  inintln al,'Введите целое число размером в байт='
  inintln ah,'А теперь еще одно='
  add al,ah
  jc  Neverno; при неправильном результате – переход
  outwordln al,,'Результат беззнаковый правильный='
  jmp fin
Neverno:
  outstr 'Результат беззнаковый неправильный'
fin:
  MsgBox "Конец программы","Желаете ещё раз?", \
          MB_YESNO+MB_ICONQUESTION
  cmp eax,IDYES
```

```
  je  Begin
  exit
  end Begin
```

Так как приходится довольно часто программировать операторы типа **if** a>b **then**, то в Ассемблер введены команды перехода, имеющие мнемонические обозначения, связанные со сравнением двух чисел. Первая буква у этих команд, естественно **j**, вторая (и третья) при сравнении на равенство **e** (equal) или **ne** (not equal), при переходе по всяким знакам сравнения буквы расставляются так: для знаковых чисел это **g** (greater), **l** (less), для беззнаковых **a** (above, выше) и **b** (below, ниже).

Часто перед выполнением команды перехода бывает надо сравнить числа. Для этого существует команда **cmp**. Работает она точно так же, команды вычитания **sub**, только результат не заносится в первый операнд (операнд не портится). Используется эта команда для выработки флагов. Пример:

```
var A,B: Longword;
if A>B then <Действия1> else <Действия2>
```

На Ассемблере это можно реализовать такими командами

```
  mov eax,A
;  приходится так делать, чтобы в следующей команде оба операнда не были из памяти
  cmp eax,B; if A>B
  ja D1;  если больше перейди к действию 1
;  операторы, выполняющие действие 2
  jmp PROD;  надо перепрыгнуть ветку < Действия1>
D1:
;  операторы, выполняющие действие 1
PROD:
;  операторы, следующие после условного оператора
```

Следует понимать, что мнемонические команды перехода работают четко по флагам (многие команды являются дублями "флаговых" команд). Па самом деле они ничего не сравнивают, они не обязаны стоять сразу после команды **cmp**, они будут работать по флагам, выработанным последней установившей их командой.

При программировании переходов на Ассемблере часто бывает полезно немножко "поколдовать" с выбором команд перехода, чтобы сделать программу более понятной, содержащей поменьше переходов. На примере предыдущего фрагмента мы видели, что программирование полного условного оператора требует двух меток и двух команд перехода. Программирование неполного условного оператора проще и, если его грамотно выполнить, потребует всего одного перехода, например

```
if EAX=EBX then <Действие>
```

реализуем так:

```
  cmp eax,ebx
  jne PROD
;  команды, выполняющие действие
PROD:
;  команды, выполняющиеся после условного оператора
```

Попробуйте запрограммировать этот оператор с помощью команды **je** и посмотрите, насколько это будет менее элегантным.

## Программирование циклов

Цикл с предусловием. Надо ввести последовательность (возможно, пустую) знаковых чисел, в конце последовательности стоит число 0 (не является элементом последовательности). Найти сумму положительных чисел. Напишем фрагмент программы для решения этой задачи:

```
  outstrln 'Вводите числа через пробел. В конце – ноль:'
  sub eax,eax; здесь будем накапливать сумму
CIKL:
  inint ebx;     ввод очередного числа
  cmp ebx,0; проверяем не конец ли последовательности
  je  KON; если последовательность закончилась, переходим на печать результата
; если последовательность не закончилась, надо проверять знак числа, но cmp выше уже есть
  jl  CIKL; если число отрицательное, надо вводить следующее, продолжать цикл
  add eax,ebx; если положительное, его надо прибавить
  jmp CIKL;      и продолжить цикл
KON:
  outintln eax,,'Ответ='
```

Как видно, для организации цикла понадобилось 2 перехода: в начале цикла на метку `KON`, чтобы прекратить цикл, если он не нужен, и в конце цикла на метку `CIKL`, чтобы продолжать цикл.

Цикл с постусловием. Надо вводить беззнаковые числа и суммировать их, пока сумма не превысит 100. Вывести количество введённых чисел.

```
  sub eax,eax; здесь будем накапливать сумму
  mov ecx,eax; здесь будем подсчитывать количество
CIKL:
  inint ebx;     ввод очередного числа
  inc ecx;       увеличиваем счетчик
  add eax,ebx; получаем сумму
  cmp eax,100; сравниваем сумму с числом 100
  jb CIKL;       нужное значение не достигнуто, продолжаем вводить
; иначе цикл заканчивается, выводим ответ
  outwordln ecx,,"Введено чисел "
```

Как видно, для организации цикла потребовался всего один переход. Следовательно, на языке машины удобнее организуются циклы с постусловием.

В Ассемблере есть специальная команд **loop** для реализации цикла с параметром **for**, про неё можно прочитать [2, разд. 6.7.3].

Следует только учесть, что команды цикла **loop**, в отличие от других команд перехода реализуют так называемый короткий переход, т.е. от команды перехода до метки должно быть примерно 30-40 команд. При этом следует иметь в виду, что макрокоманды ввода-вывода обычно содержат много команд, часто добавление внутрь цикла макрокоманды приводит к нарушению работы. Когда такое случается, надо заменить команду цикла `loop L` двумя командами `sub ecx,1` и `jnz L`.

## 11. Массивы

Все команды ассемблера, необходимые для работы с массивами Вы уже знаете. Да, собственно, никаких особых средств нет и быть не может, потому как нет никакого специального отдельного понятия "массив", объекта, который как-то по-особому хранится и обрабатывается. Есть одинаковые данные, хранящиеся в памяти друг за другом, есть желание обрабатывать их в цикле.

Итак, сначала соберем здесь команды и директивы, которые могут нам понадобятся.

Директива эквивалентности **equ**, нам она понадобится в виде

```
<имя> equ <число>
```

С ее помощью мы будем задавать размер (длину) массива. Пример

```
N equ 10; аналог const N=10; в Паскале
```

Эту директиву можно размещать в любом месте программы (желательно до использования описанного в ней имени константы).

## Директивы определения данных

Рассмотрим на примерах. Директива

```
MAS dw 10,255,-3,44,5
```

описывает переменную MAS размером в слово и отводит память под 5 слов, помещая в них числа 10, 255, –3, 44, 5.

```
ALFA db 'абвгдеёжзийклмнопрстуфхцчшщьыъэюя'
```

описывает переменную ALFA размером в байт и отводит память длиной 33 байта, помещая в нее коды (номера) 33-х букв русского алфавита.

Конструкция повторения **dup** часто используется при резервировании памяти

```
K db 8 dup (0);     отводится память 8 байт, они заполняются нулями
T db 4 dup (1,0); отводится 8 байт: 1 0 1 0 1 0 1 0
R dd 100 dup (?); 100 двойных слов без начальных значений
```

Часто эти директивы используются "парой"

```
N   equ 100
R dd 100 dup (?); N двойных слов без начальных значений
```

Можно считать это эквивалентом описания на Паскале

```
const N=100;
var R: array[0..N-1] of longint;
```

Внутри конструкции **dup** в скобках в свою очередь могут содержаться такие же конструкции, что дает возможность описать матрицу, 3-мерный массив и т.п. При этом надо иметь в виду, что матрица (данные, расположенные в виде некоторой прямоугольной таблицы) – это наше представление, на самом деле данные лежат в памяти линейно, просто друг за другом (строка за строкой). Следующие директивы резервируют память одинакового размера

```
A dd  4 dup (3 dup (?))
A dd  3 dup (4 dup (?))
A dd 12 dup (?)
```

## Работа с элементами массива. [3, 5.1 стр.83]

Директивами, описанными выше, мы резервируем память под массив, при этом имя имеет только его первый элемент, к последующим элементам можно обращаться с помощью этого имени, прибавляя к нему некоторое число (смещение от начала массива). Пусть у нас есть массивы, в которых хранятся данные разной размерности

```
B db 10 dup (?)
W dw 10 dup (?)
D dd 10 dup (?)
```

Во всех этих массивах, чтобы работать с первым значением, надо просто написать переменную – имя массива

```
mov B,1
mov D,1
```

Эти команды запишут 1 в первую компоненту массива. Чтобы записать 2 во вторую компоненту массива, придется написать

```
mov B+1,2
mov W+2,2
mov D+4,2
```

Заметим, что вместо B+1, W+2 можно писать и B[1], W[2]. В общем случае, если считать, что индексы элементов начинаются от 0, то

Адрес (X[i])=X+(**type** X)*i

где **type** X – тип переменной (1 для байта, 2 для слова, 4 для двойного слова, 8 для восьми байтового). Если хочется обрабатывать компоненты массива в цикле, надо будет пользоваться модификаторами

(вспоминаем учебные машины – там говорилось о модификации программы для работы с массивами). О регистрах-модификаторах читаем ([1], разд.5.8, формат команд "регистр-память").

Вспомним, как можно записать адрес в команде. Операнд, который берется из памяти, может быть представлен просто адресом, может – адресом с одним регистром-модификатором и с двумя модификаторами (базовым и индексным регистрами).

Операнд – адрес без модификатора

```
mov X,1
mov X+3,8
mov [X][3],0; или mov [X+3],0 или mov X[3],0 или mov X+3,0
```

Адрес может быть взят в квадратные скобки, от этого ничего не меняется, если в скобках нет регистров-модификаторов. Адрес может быть изменен на константу, константу можно писать в квадратных скобках, можно со знаком "+" или "–".

Хоть это к массивам и не относится, напомним еще раз: адреса нельзя складывать, умножать и делить. Можно к адресу прибавлять число (отнимать – т.е. прибавлять отрицательное число) – получается адрес. Адреса можно вычитать – получается число (константа).

Операнд – адрес с одним модификатором. Здесь возможны 2 вида записи

1) регистр-модификатор без множителя. В этом случае в качестве регистра-модификатора может выступать любой 32-разрядный регистр общего назначения. Регистр обязательно пишется в квадратных скобках и называется он базовым. Так же как и в случае записи адреса без модификатора, адрес может быть увеличен или уменьшен на константу, например

```
X[ebx]-2  X[ebx][-2]  X-2[ebx]  X[ebx-2]  X[-2+ebx]
```

Стоит помнить, что перед неотрицательной константой можно опускать знак "+", как мы привыкли в алгебре. Так, все выше написанные адреса – одинаковые, вычисляются `X+EBX-2`.

Такое многообразие возможностей записи адреса часто порождает мнение, что можно использовать вообще любые операции, в том числе и с регистром. Нет. Над числами можно выполнять многие операции, например, вместо константы можно написать арифметическое выражение типа `N div 2+5`. А вот регистр можно писать только в квадратных скобках и перед ним может быть только знак "+", например, запись `X[2-ESI]` – неверная.

2) регистр-модификатор с множителем. Зачем множитель? Вспомним формулу вычисления адреса – там надо умножать индекс элемента массива на тип этого элемента. Множитель может быть равен 1, 2, 4 или 8. Регистр (он называется индексным) – здесь можно использовать любой 32-разрядный регистр общего назначения кроме `ESP` (у `ESP` есть особое предназначение – работа со стеком, так что лучше бы его вообще не для чего кроме стека и не использовать). Множитель и регистр можно записывать в любом порядке, между ними обязательно ставится знак умножения *. Пара регистр-множитель записывается в квадратных скобках. Примеры

```
X[2*ebx+6]   X[esi*4][8]    X[type X * ebx-10]
```

В обоих случаях адрес (переменная) может отсутствовать. Ничего страшного в этом нет. Ведь при вычислении адресного выражения надо сложить адрес с содержимым индексного регистра. В случае отсутствия адреса он считается равным нулю (на деле просто адрес записывается в регистр, позже мы посмотрим, как это делается). Примеры

```
[ebx-2]   [esi*4][8]   [ebx*type X]
```

Заметим, что при этом мы "теряем" тип (размер) данных, с которыми хотим работать. Если такого типа адрес используется в команде с двумя операндами, то неизвестный тип одного операнда Ассемблер пытается установить автоматически, по типу другого:

`mov [esi],al` – работа с байтами.

Если же второго операнда нет или его тип неясен, надо использовать оператор явного указания типа **`ptr`**, пример

```
mov byte ptr [esi],1
outint word ptr [2*esi]
```

Макрокоманда **outchar**, несмотря на то, что она вроде бы "заточена" под работу с символами, тоже требует **byte ptr**.

### Операнд – с двумя регистрами-модификаторами

Адрес, базовый регистр (B1) и индексный регистр (I2) с множителем могут все вместе (или по отдельности) присутствовать в записи индексного выражения, соединяться они могут знаком "+" или вообще без какого бы то ни было знака. Оба регистра-модификатора, присутствующие в записи адреса, должны быть в квадратных скобках, при этом каждый из них может быть в своих скобках, могут быть оба в одних скобках (соединенные знаком +). Получается такая "формула":

<переменная> <B1><I1 с множителем или без><число>.

Перед регистрами может стоять знак "+", а перед числом знак "+" или "–", любые части могут отсутствовать. Примеры

```
A[esi+8][ebp]    A+[esi]+[4*esi]    [A+eax+eax+6]
```

Если используется не число, а константное выражение (например, разность адресов), его надо взять в круглые скобки: `[EAX][EAX](X-Y)`.

### Запись адреса в регистр

При работе с массивами используется 2 способа:

1) с помощью команды **mov** и оператора **offset**. Аргументом оператора является адрес (переменная), а результатом – значение этого адреса. Первым операндом такой команды должен быть 32-разрядный регистр (в другой регистр адрес не поместится)

2) с помощью команды **lea** – загрузка исполнительного адреса в регистр. У этой команды 2 операнда: первый – 32-разрядный регистр, второй – адрес. Команды

```
mov ebx,offset X
lea ebx,X
```

Делают одно и то же (обе не меняют флаги), первая на 1 байт короче. Заметим интересную особенность команды **lea** (лайфхак!!!). В качестве второго аргумента может фигурировать не просто адрес, а адресное выражение, в том числе с модификаторами и множителями (как только что было описано выше). Таким образом, команда

```
lea eax,[ecx][4*ebx]+5
```

фактически выполняет присваивание `eax:=ecx+4*ebx+5`, то есть в некоторых случаях несколько команд сложения-умножения можно заменить одной командой.

### Примеры программ

1. Пусть в памяти хранится текст, некоторое количество символов (байт). Задается номер начального символа и количество символов для вывода. Нужный фрагмент на Ассемблере

```
T db 'Собака Слон Кошка Крокодил'
     ...
  mov ebx,<число>;  номер начального символа
  mov ecx,<число>;  длина слова для вывода
PRINT:
  outchar T[ebx]
  inc ebx
  loop PRINT
```

Для печати слова «Слон» надо задать числа 7 и 4. Можно перед циклом воспользоваться загрузкой адреса в регистр и в цикле имя массива не использовать (это нам пригодится при программировании процедур):

```
  mov ebx,offset T+7; адрес первого символа
  mov ecx,4;          длина слова
PRINT:
  outchar byte ptr [ebx]
  inc ebx
```

```
  loop PRINT
```

Можно адрес начала массива загрузить в другой регистр и в макрокоманде использовать два модификатора:

```
  lea eax,T
  mov ebx,7
  mov ecx,4
PRINT:
  outchar byte ptr [eax][ebx]
  inc ebx
  loop PRINT
```

2. Вести в массив числа в формате **dd** (последнее число 0, чисел на входе не больше 30, иначе будет ошибка). Напечатать введенные числа в обратном порядке.

```
include console.inc
.data
  mas dd 30 dup (?)
.code
BEGIN:
  mov esi,-4
  mov ecx,-1; счетчик количества введенных чисел
L:add esi,4 ; регистр увеличиваем на 4, длина элемента массива
  inint mas[esi],"Введите число="
  inc ecx
  cmp mas[esi],0
  jne L; если не 0, продолжаем цикл
  newline
Comment @
  Теперь вывод. Так удобно оказалось, что к началу этого
  цикла все готово – в регистре ecx количество введённых
  чисел (0, если чисел не было), в esi – смещение от начала
  массива до последнего числа
@
  jecxz Kon; если чисел не было (сразу 0)
M:outint mas[esi],10
  sub esi,4
  loop M
  newline
Kon:
  exit
  end BEGIN
```

Отметим, что при работе с массивами надо очень внимательно следить, чтобы регистр модификатор изменялся не на 1, как мы привыкли в Паскале, а на величину, соответствующую типу элементов: 1 для байтов, 2 для слов, 4 для двойных слов, 8 для восьми байтовых чисел. Иначе мы рискуем получить данные, которые никто не вводил. Скажем, в случае только что рассмотренного массива, если по ошибке изменять регистр ESI на единичку, возникнут числа, состоящие из байтов соседних элементов, например, в качестве второго числа будут использованы 3 байта первого числа и один байт второго. Если вспомнить, что числа хранятся в перевернутом виде, даже сложно представить, какой хаос получается.

3. Циклический сдвиг массива на 2 элемента вправо

```
include console.inc
N equ 5
.data
X dd N dup (?)
.code
```

```
BEGIN:
  mov ebx,-(type X); скобки можно не ставить
  mov ecx,N
VVOD:
  add  ebx,type X
  inint X[ebx],'? '
  loop VVOD
  mov  esi,X[ebx]; сохранили последний
  mov  edi,X[ebx-type X]; сохранили предпоследний
  mov  ecx,N-2; оставшиеся N-2 будем двигать
CIKL:
  mov  eax,X[ebx-2*type X]
  mov  X[ebx],eax
  sub  ebx,type X; уменьшили индекс
  loop CIKL; в цикле передвинем
  mov  X,edi; последние 2 элемента надо поставить на 1 место
  mov  X+type X,esi; и на второе
; печать массива
  mov  ecx,N; количество элементов
  mov  ebx,0; начнем с нулевого
VIV:
  outint X[ebx],10
  add  ebx,type X
  loop VIV
  newline
  exit
  end BEGIN
```

К сожалению, несмотря на использование в работе с массивом оператора **type**, заменить в описании массива тип (чтобы программа при этом осталась работающей) просто так не получится. Элементы массива сохраняются в регистрах, регистр указывается явно.

## Работа с матрицей

В программе ниже показаны разные приемы работы с матрицей. К сожалению, при изменении типа компонент матрицы программа тоже работать не будет, так как для вычислений используются регистры. Обратите внимание, что с матрицей можно работать как с линейным массивом (чем она и является) обычным образом переходя от одного элемента к другому и только при печати "вспомнить", что надо элементы вывести на разных строках. Для такой работы достаточно одного регистра-модификатора.

Можно использовать два модификатора: один отвечает за смещение по строке, другой – по столбцу. При этом надо внимательно следить за правильностью изменения значений регистров: при переходе от одной строки к другой. Модификатор надо увеличить не на единичку, (как в Паскале), и даже не на тип элемента массива (как при переходе к соседнему элементу), а на количество байт, которое занимает строка, то есть надо количество элементов в строке умножить на тип элемента.

```
include console.inc
comment *
    Работа с матрицей. Матрица не вводится,
    а задается константами в переменной  matr
    считается сумма всех элементов матрицы
*
.const
  d equ 5;  количество столбцов матрицы
  h equ 3;  количество строк
.data
```

```
  matr dw 1, 2, 3, 4, 5
       dw 0, 0, 9, 8, 7
       dw 6,-5,-4,-3,-2;  можно задать всё в одной строке
.code
Start:
  ClrScr
  ConsoleTitle "  Матрица"
;  Работа со всеми элементами матрицы. Матрица как
;  линейный массив, использование одного индекса
  xor eax,eax;  здесь будет сумма
  xor ebx,ebx;  обнуляется индексный регистр, чтобы
;               начать работу с первого элемента
  mov ecx,d*h;  количество элементов в матрице
sum:
  add  eax,matr[ebx*type matr]; это ebx*(type matr)
  inc  ebx
  loop sum
  outintln eax,,'Сумма элементов матрицы='
;  Работа со всеми элементами матрицы,
;  использование двух индексов
  outstrln 'Печать матрицы'
  xor ebp,ebp;  обнуление для движения вниз (индекс строки)
r0:
  xor ebx,ebx;  обнуление для движения по строке вправо (индекс столбца)
r1:
  outint matr[ebp][ebx],8
  add ebx,type matr;  индекс столбца увеличивается на размер элемента
  cmp ebx,d*type matr
  jne r1
  newline
  add ebp,d*type matr;  индекс строки увеличивается на размер строки
  cmp ebp,d*h*type matr
  jne r0
;  Вычисление адреса элемента по его позиции (вводится позиция,
;  печатается элемент, который на ней стоит)
  inint eax,'номер строки='
  dec eax
  mov edx,d*type matr
  mul edx
  inint ebx,'номер столбца='
  dec ebx
  outintln matr[eax][ebx*type matr],,'на этом месте стоит='
;  Работа с элементами столбца. Загрузка адреса в регистр
  inint ebx,'введите номер столбца, который надо напечатать='
  lea ebx,matr[ebx*type matr-type matr]
;  адрес первого элемента заданного столбца
  mov ecx,h
  xor edx,edx
sta:
  outint dword ptr [ebx+edx*type matr],8
  add edx,d;  перехд к элементу следующей строки
  loop sta
```

```
  newline
  exit
  end Start
```

6. Подсчет. Входные данные в массиве не хранятся. Массив используется для хранения данных, полученных при обработке входной последовательности.

Условие задачи. Ввести непустую последовательность неотрицательных (беззнаковых) чисел, каждое не превышает 15, последовательность заканчивается –1. Общее количество чисел не превышает 256. Определить, сколько раз каждое число входит в эту последовательность.

Поясним ограничения: число от 0 до 15, чтобы использовать массив размером 16 элементов. Соответственно, при изменении диапазона чисел надо просто изменить длину массива. Количество чисел не превышает 256, чтобы при любом раскладе это количество поместилось в компоненту массива длиной в байт. Соответственно, если количество чисел будет больше, надо взять массив слов, для индексации массива надо будет использовать модификатор с множителем.

```
include console.inc
.data
DOP db 16 dup(0); дополнительный массив для подсчета
;                 количества вхождений каждого числа
;                 вначале массив заполнен нулями
.code
start:
VVOD:
  outstr 'Введите число не больше 15 (конец -1):'
  inint ebx
  cmp ebx,-1
  je  Fin
  cmp ebx,0
  jl  Neprav
  cmp ebx,16
  jaёe Neprav
; сюда попали, если число от 0 до 15
  inc DOP[ebx]; увеличиваем соответствующий элемент массива
  jmp VVOD
Neprav:
  outstrln 'Неправильное число'
  jmp VVOD
Fin:
  newline
  mov ebx,0;   загрузка индекса, распечатываем массив с начала
  mov ecx,16;  в массиве 16 чисел
  mov eax,0
Print:
  outword ebx,8; печатаем число
  outwordln DOP[ebx],8,'-'
; печатаем, сколько раз оно входит в последовательность
  inc ebx
  dec ecx
  jne Print
  exit
  end start
```

## 12. Запись операндов в командах

Мы узнали, что в команде операнд может быть не только числом или просто адресом, а ещё и адресным выражением. Посмотрим, как правильно записывать операнды разных видов.

Про команды, в которых 2 операнда

Если типы обоих операндов известны (регистры, память), то эти типы должны быть одинаковы.

```
A dd ?
  cmp A,eax;  верно
  cmp A,al;   неверно
```

Для изменения типа адресного выражения перед адресом можно использовать оператор **ptr**

```
cmp byte ptr A,al;  возьмет байт по адресу A, то есть младший байт
```

Писать **ptr** перед регистром нельзя – ошибка. По правилам писать **ptr** можно перед любым адресом и применять к любому типу. Практически имеет смысл только "уменьшение" типа – как в примере, от двойного слова взяли один байт. Наоборот, если переменная байт, а ее расширяем до слова – не ошибка, работать будет, но получается неизвестно что, так как захватывается соседняя, следующая ячейка.

Если известен тип одного операнда (например, регистр), а тип второго – не определен (косвенная адресация, квадратные скобочки), в этом случае тип второго операнда определяется по типу регистра. Оператор **ptr** здесь не нужен (будет лишним).

Здесь надо быть особенно внимательным, можно сделать ошибку, которую транслятор не показывает. Например, используется байтовый массив X и есть такие команды

```
lea ebx,X
mov [ebx],eax;  в байтовый массив помещается двойное слово,
;               то есть изменяется не один байт, а сразу 4
```

Если известен тип одного операнда (регистр, адрес), а второй – непосредственный операнд (число). В этом случае число должно помещаться в размер операнда. Если оно больше – ошибка

```
mov eax,12345678; верно
mov eax,'F';      верно
mov al,300;       ошибка
```

Перед первым операндом (адресом) может стоять **ptr** с указанием типа. Синтаксис языка не запрещает поставить **ptr** с указанием типа и перед числом (в этом случае он задаёт *длину* числа в байтах), но делать этого не рекомендуется, потому что **ptr** – оператор указания типа переменной в памяти. Логично, что за **ptr**, то есть, за ссылкой (pointer), стоит адрес переменной. Тем более, что Ассемблер работает в этом случае "странно", число уменьшается только в случае необходимости (если первый операнд маленького типа), иначе указатель типа игнорируется

```
mov al,byte ptr 300; al:=44 (???)
mov ax,byte ptr 300; ax:=300, ptr игнорирует
```

Если тип первого операнда не определён (косвенная адресация), а второй – непосредственный операнд (число). Это ошибка, каким бы ни было число.

```
mov [ebx],12345; Ошибка.
```

Обязательно нужно писать **ptr** с указанием типа. Логичнее это делать для первого операнда, потому что именно он является адресом **mov dword ptr**[ebx],12345 – так правильно.

```
mov [eax],byte ptr 300
```

Так тоже синтаксически правильно, запишет 44 в байт, но лучше так не делать, нелогично.

Если в команде один операнд, то, как правило, должен быть известен его тип

```
div [ebx]; ошибка, надо использовать ptr
```

## 13. Стек

Теорию читаем [1. глава 6]

Стек – выделенная область памяти, с которой можно работать как обычными, так специальными (стековыми) командами. Обычно переменные в стеке не именуется, доступ к ним осуществляется с помощью адресов в регистрах. Главным, отвечающим за адресацию в стеке регистром, является ESP (вспомним, это тот самый регистр, который нельзя использовать как регистр-модификатор с множителем). В начале работы программы ему автоматически присваивается некоторое значение – адрес начала стека, его "дно" (точнее, в начале он указывает не на само "дно", а стоит за ним, на свободном месте за концом стека). При записи информации в стек ESP уменьшается, при чтении – увеличивается. Во время работы программы в ESP хранится адрес вершины стека, то есть адрес последнего записанного в стек элемента.

Стековые команды

Команды для работы со стеком **push** и **pop** имеют один операнд размера двойное слово (на самом деле еще и слово, но мы будем работать со стеком только с двойными словами). Команда **push** уменьшает значение ESP и затем заносит в стек свой операнд по получившемуся адресу (он может быть из памяти или из регистра). Команда **pop** считывает двойное слово из стека (по адресу ESP) в свой операнд (адрес или регистр) и увеличивает значение ESP. Примеры:

```
push 3      pop X      push eax
```

Заметим, что, если операндом стековой команды является адрес, команда выполняет действие типа "память-память". Мы при этом говорили, что команд формата "память-память" не бывает. В чем противоречие? Запрещены только команды, у которых два явных операнда из памяти. Формат у этой команды – один операнд из памяти, второй в явно команде не указывается, его адрес содержится в регистре ESP.

Можно записать и прочитать из стека регистр флагов EFLAFS, это делают команды без операндов **pushfd** и **popfd**. Можно также записать в стек и прочитать из стека все 8 регистров общего назначения командами **pushad** и **popad**.

Чтобы освоить работу со стеком, полезно провести следующие "эксперименты". Посмотрим, чему равно значение ESP в начале программы (это же обычный регистр, его можно распечатать). Потом запишем в стек несколько чисел, посмотрим, как изменится ESP.

```
outwordln esp
push 111
push 222
outwordln esp
```

Чему равно значение ESP в начале – не важно (оно может быть разным на разных компьютерах), важно, что оно уменьшилось на 8, так как в стек записали 2 двойных слова. Если теперь эти значения вытащить из стека двумя командами **pop**, значение ESP станет таким же, каким было в начале. Именно так можно отслеживать, не стал ли стек пустым: сохранить начальное значение ESP и проверять, не достигнуто ли оно. Это бывает нужно для предотвращения ошибок в программе, чтобы не было попыток чтения из пустого стека (при этом, конечно, программа аварийно завершается). Второй способ контроля пустоты стека – считать, сколько чисел в него поместили, сколько удалили.

Описанный выше прием можно использовать и для быстрой очистки стека.от какого-то число "ненужных" байт Понятно, что для очистки стека надо можно команду **pop** столько раз, сколько в нем на данный момент чисел. Однако можно просто в начале работы программы запомнить значение ESP и для очистки стека присвоить ESP это запомненное значение.

С информацией, находящейся в стеке, можно работать не только с помощью команд **push** и **pop**. Числа, находящиеся в стеке, можно считывать и изменять, используя адресацию с помощью регистров, т.е. работая со стеком, как с обычной памятью. При этом нам известен адрес вершины стека (в каждый конкретный момент) он находится в регистре ESP и тип компонент – 4 байта. Поместим в стек 4 числа и выведем их на экран, не убирая из стека (т.е. не пользуясь командой **pop**)

```
push 111
push 222
push 333
```

```
    push 444 ; записали числа в стек
    mov  ebp,esp; запомнили адрес начала массива
    mov  ecx,4; счетчик для работы цикла
cykl1:
    outint dword ptr [ebp],5;
; выводим число, находящееся по адресу [EBP]
    add ebp,4; числа – двойные слова, изменяем регистр на 4
    loop cykl1
```

Заметьте: мы в качестве регистра-модификатора использовали EBP, а не ESP. Это очень важно! Использовать ESP в качестве модификатора можно, но тогда, изменяя ESP, мы будем фактически "доставать" число из стека. То есть, если в данном фрагменте в качестве модификатора использовать ESP, работа фрагмента будет аналогична использованию 4 раза команды **pop**, стек станет пустым.

Работу со стеком с помощью обоих приемов (стековые команды и адресация) можно посмотреть на примере следующей программы.

Задача. Вести последовательность не менее 3-х знаковых чисел формата **dd**, заканчивающуюся нулем. Напечатать числа, стоящие в этой последовательности после максимального (последнего из них, если их несколько).

Сначала за максимальное мы вынуждены принять первое число (других пока нет). Все вводимые числа будем класть в стек, пока они меньше максимального. Если вдруг встретилось число, большее максимального, придется очистить стек и начать в него складывать числа заново, те, которые стоят уже после нового максимального.

```
include console.inc
.code
Start:
  mov ebp,esp; запомним указатель стека
  inint eax,"первое число="; примем за максимальное
  xor ecx,ecx; здесь будем считать количество чисел в стеке
VVOD:       ; ввод чисел
  outchar 'Число='
  inint esi
  cmp esi,0
  je  PRINT; конец последовательности, можно печатать
  cmp esi,eax
  jge IZM;    число больше или равно max – изменяется максимум
  push esi;   максимум не изменился, число надо записать в стек
  inc ecx;    в стеке стало на 1 число больше
  jmp VVOD
IZM:
  mov eax,esi; изменили значение максимума
  mov esp,ebp; очищаем стек, так как числа мы записали зря, они стоят до максимума
  xor ecx,ecx; обнулим счетчик, так как в стеке чисел теперь нет
  jmp VVOD
PRINT:      ; печатаем числа, в ECX их количество
  pop esi   ; достаем из стека
  outint esi,10
  loop PRINT
  MsgBox "Конец задачи","Повторить программу ?", \
         MB_YESNO+MB_ICONQUESTION
  cmp  eax,IDYES
  je   Start
  exit
```

```
  end Start
```

Заметим, что в наших программах мы работали с двойными словами (**dd**), именно таков тип операндов у стековых команд. Если надо работать с числами меньшего размера, писать в стек и брать из стека все равно следует использовать двойные слова, расширяя исходные числа до 4 байт.

Использование стека

1. Вышеприведенная программа, конечно, носит демонстрационный характер – не так часто приходится печатать последовательность после максимума, но, тем не менее, одно из предназначений стека – временное хранение последовательности данных.

2. Присваивание значения переменной (копирование из памяти в память). Действие `X:=Y`, где оба числа – из памяти, в ассемблере приходится делать с использованием регистра. Можно это сделать с помощью стека

```
push Y
pop  X
```

3. Поменять местами значения переменных. Команд столько же, но иногда не хочется портить регистры.

```
push X
push Y
pop  X
pop  Y
```

4. Сохранить, а потом восстановить регистр флагов – если надо использовать ранее полученные значения флагов.

5. Сохранить, а потом восстановить значения регистров – по понятным причинам.

6. При работе с вложенными циклами количество повторений циклов удобно сохранять в стеке, так как эти числа надо извлекать в порядке обратном тому, в котором клали.

7. Иногда в стеке удобно сохранять сравнительно большие массивы информации, необходимой для работы программы (вместо того, чтобы отводить место под такой массив стандартным образом в памяти).

Посмотрим такую работу со стеком на примере, где массив нужен для хранения результатов подсчета результата. Похожий пример был рассмотрен выше при работе с массивами.

Задача. Вводится текст с точкой в конце. Напечатать количество вхождений каждой маленькой латинской буквы.

Для подсчета создадим в стеке массив – в начале запишем в стек 26 (столько латинских букв) нулей. Введя очередную букву, определим ее номер в алфавите и увеличим на 1 соответствующую компоненту в стеке. Доступ к компонентам стека при этом будем осуществлять с помощью модификатора `ESP` – будем использовать его как базу, то есть вершина стека является началом массива, по этому адресу, в `[ESP]` хранится количество вхождений буквы `'a'`. Количество вхождений буквы `'b'` хранится в следующей компоненте стека, по адресу `[ESP+4]` (изменение адреса на 4 потому, что в стеке хранятся двойные слова). Для нахождения нужного адреса в стеке будем использовать модификатор `EAX`, в котором будем хранить номер буквы в алфавите.

```
include console.inc
.code
Start:
  mov ecx,26
Obnul: ; обнуление 26 позиций в стеке
  push 0
  loop Obnul
  outstrln "Вводите текст до точки:"
vvod: ; ввод строки символов и обработка
  inchar al
```

```
  cmp al,'.'; строка заканчивается точкой
  je  kon
  cmp al,'a'
  jb  vvod;   al<'a' не буква
  cmp al,'z'
  ja  vvod;   al>'z' не буква
; сюда попали, только если была введена маленькая буква
  sub al,'a'; получили номер буквы 0..25
  inc dword ptr[esp+4*eax]
; в стеке хранятся двойные слова, поэтому умножаем на 4
  jmp vvod
kon:
  outstrln "Печать:"
  mov al,'a'
print:
  outchar al; очередная буква
  pop ebx; забираем из стека соответствующее ей число
  outword ebx,10,' - '
  inc al; переходим к следующей букве
  cmp al,'z'
  jbe print
  newline
  exit
  end Start
```

Аккуратность при работе со стеком

Стек используется для разных целей, иногда его использование "явно" не видно. Например, макрокоманды ввода-вывода активно пользуются стеком, на работе со стеком основана работа процедур. Поэтому при небрежной работе со стеком можно не просто получить неправильные числа в ответе, а "подвесить" программу, так как она "не туда уйдет". Главное правило – правило подводника: количество погружений равно количеству всплытий – сколько положили в стек, равно столько и достали.

Теперь, когда мы знаем, каково предназначение регистра `ESP`, четко повторим и поясним правило, высказанное ранее. Кстати, название этого регистра происходит от словосочетания Stack Pointer – указатель (вершины) стека. Правила языка не запрещают изменять его значение, использовать его в арифметических командах. Однако следует понимать, что все изменения значения этого регистра приводят к изменению положения вершины стека (если что-то положить в стек, а потом изменить значение `ESP`, то команда **`pop`** уже вытащит что-то другое).

Если пользуемся содержимым стека как массивом, также важно правильно этот массив в стек положить и вовремя его оттуда вытащить.

Немного об очистке стека. При чтении из стека информации командой **`pop`** число, которое было на вершине стека, из стека, конечно же, не стирается, оно там остается, изменяется вершина стека, число оказывается в "незанятой" памяти. Оно может быть стерто (изменено), когда на это место что-то в стек положат. Не надо надеяться, что это число останется в неизменном состоянии до тех пор, пока не будет выполнена явно прописанная в программе команда **`push`**. Стеком пользуются макрокоманды, он используется при работе процедур, да и сам компьютер может писать туда свои данные. Поэтому, например, любой ввод-вывод испортит то, что раньше "числилось" в стеке.

## *14. Процедуры*

Несколько примеров разных процедур с разными видами передачи параметров.

Работа с массивом. Передача параметров через регистры

```
include console.inc
```

```
comment *
     Процедуры работы с массивами
*
.data
N equ 10
X dd N dup (?)
Y dd (N/2) dup (99); N/2=N div 2
.code
VVMAS proc
; ввод массива, передача параметров через регистры
; ebx - адрес начала, ecx - количество
  outstr 'Ввод массива. Количество элементов='
  outwordln ecx
@VV:
  inint dword ptr[ebx]
  add ebx,4
  loop @VV; метка "необычная", чтобы в программе не было такой же
  ret
VVMAS endp
; ================================
PRMAS proc
; вывод массива, передача параметров через регистры
; ebx - адрес начала, ecx - количество
  newline
PP:
  outint dword ptr[ebx],10
  add ebx,4
  loop PP
  newline
  ret
PRMAS endp
; ================================
Start:
  ClrScr
  ConsoleTitle "  Процедуры. Передача параметров через стек"
; перед вызовом процедуры надо в регистры занести нужные значения
  mov  ebx,offset X; адрес X
  mov  ecx,N
  call VVMAS
  lea  ebx,X
  mov  ecx,N
  call PRMAS
  mov  ebx,offset Y; адрес Y
  mov  ecx,N/2
  call PRMAS
   ...
```

Передача параметров через стек.

Написать процедуру нахождения максимального из двух чисел. Параметры в стеке, результат – в регистре EAX. Применить эту процедуру дважды, чтобы найти максимальное из трех чисел.

```
include console.inc
comment *
   eax:=MAX(A,B) - параметры в стеке
*
```

```
.data
G dd 23
M dd 56
P dd 67
.code
Max proc; параметры в стеке, ответ в eax
; пролог
  push ebp
  mov  ebp,esp
  push ebx; сохраняем используемые регистры
  mov  ebx,[ebp+8]; ebx:=первый параметр
  mov  eax,[ebp+12]; eax:=второй параметр
  cmp ebx,eax
  jbe @F; это метка без имени – переход на ближайшую метку @@ вперед
  mov eax,ebx
@@:
; эпилог
  pop ebx
  pop ebp; убрали из стека все, что туда клали с помощью push
  ret 2*4; убрали из стека 2 параметра
Max endp
Start:
  push 123
; положим в стек это число, чтобы проверить, все ли из него убрали в процедуре
  push G
  push M
  call Max; два числа в стеке, вызываем процедуру
  outwordln eax; это максимум первых двух чисел
  push eax
  push P; максимум и третье число кладем в стек
  call Max
  outwordln eax; теперь это максимум всех трех чисел
  pop eax  ; проверяем – то ли в стеке, что мы положили в начале
  outwordln eax; должно быть 123
  exit
  end Start
```

Ввод и вывод массива – передача параметров через стек

```
include console.inc
comment *
  Процедуры работы с массивами
*
.data
N equ 5
X dd N dup (?)
Y dd N/2 dup (99)
.code
VVMAS proc
; procedure VVMAS(var X:Mas; N:Longword);
; ввод массива, передача параметров через стек,
; внизу - количество, выше - адрес начала
  push ebp
```

```
  mov  ebp,esp
  push ecx
  push ebx
  mov  ebx,[ebp+8]
  mov  ecx,[ebp+12]
  outstr 'Ввод массива. Количество элементов='
  outwordln ecx
@@:
  inint dword ptr[ebx]
  add ebx,4
  loop @B; на ближайшую @@: назад
  newline
  pop ebx
  pop ecx
  pop ebp
  ret 2*4
VVMAS endp
; ========================
PRMAS proc
; вывод массива, передача параметров через стек, работа на
; регистрах ebx - адрес начала, ecx - количество
  push ebp
  mov  ebp,esp
  push ecx
  push ebx
  mov  ebx,[ebp+8]
  mov  ecx,[ebp+12]
PP:
  outint dword ptr[ebx],10
  add ebx,4
  loop PP
  newline
  pop ebx
  pop ecx
  pop ebp
  ret 2*4
PRMAS endp
; ===================
Start:
  ClrScr
  ConsoleTitle "  Процедуры. Передача параметров через стек"
  push N;          Сначала второй параметр
  push offset X;   Потом первый параметр
  call VVMAS
  push N
  push offset X
  call PRMAS
  outstrln 'Печать закончена'
  push N/2
  push offset Y
  call PRMAS
  exit
  end Start
```

Вспомним, что передача параметров в Паскале бывает по значению и по ссылке (параметры-значения и параметры-переменные, с **var** и без **var**). Пришла пора узнать, что это такое на уровне Ассемблера. Параметры-значения – это то, что мы кладем в регистры или в стек в виде значений. В программах с массивом – это количество элементов в массиве (кладем в стек или в регистр число), в программе с максимум оба параметра – значения. Параметром-переменной является массив, через регистр или через стек передается адрес массива. Здесь мы видим, что если бы мы вдруг захотели передать массив по значению, пришлось бы записывать в стек все числа массива, а при передаче параметра по адресу мы сообщаем процедуре адрес начала массива и она уже обращается, зная этот адрес, непосредственно к массиву в памяти.

Передача параметров по ссылке также имеет место, когда этот параметр является результатом.

Напишем процедуру на Паскале с тремя параметрами, которая реализует действия `C:=max(A,B)`.

```
procedure max(A,B: Longint; var C: Longint);
begin C:=A; if A<B then C:=B end.
```

Обратите внимание: процедура не просто находит максимум двух чисел (раньше мы писали функцию, которая находила максимум и возвращала его в регистре). Процедура изменяет значение переменной C, кладет найденное значение в память по указанному адресу.

Передадим параметры через регистры. Первые 2 параметра передаются по значению, третий – по ссылке, то есть в регистр надо положить его адрес. При работе с этим параметром надо будет работать не с регистром, а с адресом, который в этом регистре лежит, то есть брать регистр в квадратные скобочки

```
include console.inc
.data
A1 dd 34
B1 dd 45
C1 dd 56
C2 dd 12
.code
MAXADR proc; C:=max(A,B)
; параметры передаются через регистры  A - EAX,
; B - EBX  по значению,  C - ESI  -  по адресу)
; пересылаем значение не в сам регистр ESI, а по адресу в этом регистре
  mov [esi],eax; C:=A
  cmp eax,ebx
  jl KON
  mov [esi],ebx; C:=B
KON:
  ret
MAXADR endp
Start:
  mov eax,A1
  mov ebx,B1;  в регистры заносим параметры-значения
  mov esi,offset C1;  параметр передается по ссылке, в регистр кладем адрес
  call MAXADR; C1:=max(A1,B1)
  outintln C1;  проверяем – значение в памяти изменилось
  mov eax,C1
  mov ebx,C2;  в регистры заносим параметры-значения
  mov esi,offset C2;  параметр передается по ссылке, в регистр кладем адрес
  call MAXADR; C2:=max(C1, C2)
  outintln C2;  проверяем – значение в памяти изменилось
  exit
  end Start
```

Перепишем эту процедуру, чтобы все параметры передавались через стек – тогда она будет записана с учетом стандартных соглашений о связях.

```
include console.inc
.data
A1 dd 34
B1 dd 45
C1 dd 56
C2 dd 12
.code
MAXADR proc
; параметры передаются через стек
  push ebp
  mov  ebp,esp
  push eax
  push ebx
  push esi
  mov  eax,[ebp+8];  eax:=первый параметр  A
  mov  ebx,[ebp+12]; ebx:=второй параметр B
  mov  esi,[ebp+16]; esi:=адрес C, куда надо класть ответ
  mov  [esi],eax; C:=A
  cmp  eax,ebx
  jl   KON
  mov  [esi],ebx; C:=max
KON:
  pop esi
  pop ebx
  pop eax
  pop ebp
  ret 4
MAXADR endp
Start:
; procedure MAXADR(A,B:Longint; var C:Longint);
  push offset C1; сначала третий параметр
  push B1;        теперь второй параметр
  push A1;        теперь первый параметр
  call MAXADR
  outintln C1; проверяем – значение в памяти изменилось
; A теперь C2:=max(C1,C2)
  push offset C2; 3-ий
  push C2;        2-ой
  push C1;        1-ый
  call MAXADR;    C2:=max(C1,C2)
  outintln C2; проверяем – значение в памяти изменилось
  exit
  end Start
```

## Сохранение параметров в стеке

Если процедуре в процессе работы нужно что-то хранить в памяти (локальные переменные), эти данные она может сохранять в стеке.

Задача. Мы писали программу, которая подсчитывала, сколько раз буква входит в текст, и данные сохраняли в стеке. Опишем эти действия в виде процедуры. Пусть параметрами процедуры будут 2 символа – начало и конец сегмента алфавита, в котором процедура будет подсчитывать символы (раньше

программа подсчитывала латинские буквы, процедуре можно задавать, что считать, иначе какая же это процедура).

Свои локальные данные процедура будет содержать в стеке, можно увидеть, что работа со стеком внутри процедуры практически ничем не отличается от работы со стеком внутри программы. Переменную и массив называть никак не будем, будем обращаться к ним по адресам.

```
include console.inc
comment *
      Параметры процедуры – два символа c1 и c2 (они передаются
      в одном двойном слове через стек). Если код первого символа
      больше кода второго – процедура ничего не делает.
      Иначе процедура вводит текст (до точки) и печатает,
      сколько раз в текст входит каждый символ из заданного
      диапазона. В процедуре используется локальная переменная
      для хранения длины заданного диапазона и локальный массив
      (длина массива зависит от заданных параметров)
*
.code
; type P=packed array[1..4] of char;
; procedure Letter(x:P); {x='c1c2#0#0'}
Letter proc
  push ebp
  mov  ebp,esp; стандартные входные действия (пролог)
  sub  esp,4;   локальная переменная для хранения длины диапазона
  push eax;     сохранение регистров
  push ebx
  push ecx
  push edx
  push edi
  mov  edx,[ebp+8]; читаем параметр из стека
  outstr 'Выбран диапазон '
  outchar dl
  outcharln dh,,"-"
  xor  ecx,ecx
  mov  cl,dh
  sub  cl,dl
  jb   @kon; результат <0 - делать ничего не надо
  add  ecx,1; вычислена длина исследуемого диапазона
  outintln cl,,'Длина='
  mov  [ebp-4],ecx; сохранили ее в локальной переменной в стеке
  newline
@@:
  push 0; запись в стек ecx нулей – счетчики для символов
  loop @B
  xor  ebx,ebx; регистр для работы с массивом
  outstrln 'Вводите текст с точкой в конце:'
@@:       ; начало ввода
  inchar bl; ввели очередной символ
  cmp bl,'.'
  je  @F; на метку "обработка содержимого стека"
  cmp bl,dl
  jb  @B; меньше начала диапазона, переход на ввод
  cmp bl,dh
```

```
  ja  @B; больше конца диапазона, переход на ввод
  sub bl,dl; номер буквы (начиная с 0)
  inc dword ptr [esp+ebx*4]
  jmp @B; на метку "начало ввода"
@@:     ; обработка содержимого стека
  mov ecx,[ebp-4]; считали из стека длину диапазона
  outstrln 'Каждый символ встретился '
@@:
  outchar dl
  pop eax
  outintln eax,3
  inc dl
  dec ecx
  jnz @B; это замена LOOP - он здесь не проходит
  newline
@kon:   ; выходные действия процедуры, эпилог
  pop edi
  pop edx
  pop ecx
  pop ebx
  pop eax
  add esp,4; можно mov esp,ebp
  pop ebp
  ret 4*1; удаляем из стека параметр
Letter endp
; ==================
Start:
  ClrScr
  push 1234; для последующей проверки правильности очистки стека
  mov  ebp,223344; для последующей проверки правильности восстановления регистров
  mov  ecx,665577
  ConsoleTitle "  Процедура с локальным массивом"
  xor  eax,eax
  mov  al,'a'
  mov  ah,'m'
  push eax; 'am#0#0'
  call Letter
  outstr 'Введите еще один диапазон - два символа='
  inchar bl
  inchar bh
  push ebx
  call Letter
  newline
  pop ebx
  outword ebp,10
  outword ecx,10
  outwordln ebx,10
  exit
  end Start
```

В написанной процедуре мы со стеком работаем в основном "обычным" образом: записываем в него нули командой **push** (и заодно отводим память под локальный массив), а в конце читаем информацию

командой **pop** (и заодно освобождаем стек от локального массива). Чаще память в стеке отводят и высвобождают с помощью изменения значения регистра ESP.

Задача. Напишем похожую процедуру. Процедура вводит текст с точкой в конце и в качестве результата выдает в регистре AL букву, которая в этом тексте встречается чаще всего (будем считать, что хотя бы одна буква в тексте есть, если одинаково часто встречаются несколько букв, выведем любую из них).

Локальные данные такой процедуры – массив количеств букв – будем хранить в стеке. Никаких параметров у процедуры нет, результат она передает через регистр AL (т.е. это, вообще-то функция).

```
include console.inc
.code
MAXLET proc
  push ebp
  mov  ebp,esp
  sub  esp,26*4;  порождение локального массива из 26 двойных слов
comment *
  [ebp-26*4];  начало массива; это адрес первого элемента,
  адрес второго  [ebp- 26*4 +4] и т.п.
  можно было не заводить массив изменением  ESP,
  так как в него надо положить начальные значения,
  можно было "запушить" 26 нулей.
*
  push ecx
  push ebx
  push edx
  outstrln "Вводите текст с точкой в конце"
  mov  ecx,26
  xor  eax,eax
Obnul: ;  обнуление 26 позиций в стеке
  mov  dword ptr [ebp-26*4+4*eax],0
  inc  eax
  loop Obnul
vvod: ;  ввод строки символов и обработка
  inchar al
  cmp al,'.';  строка заканчивается точкой
  je  kon
  cmp al,'a'
  jb  vvod
  cmp al,'z'
  ja  vvod
;  сюда попали только, если была введена маленькая буква
  sub al,'a';  получили номер буквы
  inc dword ptr [ebp-26*4+4*eax];  двойные слова, поэтому умножаем на 4
  jmp vvod
kon:
  outstrln "Печать"
  mov ebx,0
  mov dl,'a'
  mov ecx,0
print:
  outchar dl;  очередная буква
  outstr ' - '
  outword dword ptr [ebp-26*4+4*ebx]
```

```
; сравниваем текущее значение и максимальное в cl
  cmp dword ptr [ebp-26*4+4*ebx],ecx
  jbe @F
  mov ecx,dword ptr [ebp-26*4+4*ebx]; меняем максимум
  mov al,dl
@@:
  outstr '       '
  inc ebx
  inc dl; переходим к следующей букве
  cmp dl,'z'
  jbe print
  newline
  pop edx; Эпилог работаем со стеком в обратном порядке
  pop ebx
  pop ecx
  add esp,26*4; освобождаем стек от локального массива
  pop ebp
  ret
MAXLET endp
Start:
  call MAXLET
  outchar al
  exit
  end Start
```

## *15. Многомодульные программы*

Чтобы хорошо понять данный материал, надо прочитать соответствующие главы в учебнике [1]. Здесь только приводятся примеры работающих программ.

Как происходит работа многомодульной программы

1. Имеем несколько модулей, написанных на одном или нескольких языках программирования, между модулями имеются некоторых связи (по переменным, по меткам). Эти связи должны быть явно обозначены.

Если из модуля совершается переход по некоторой метке, описанной в другом модуле или используется переменная, описанная в другом модуле, без указания, что эти объекты являются внешними, модуль компилироваться не будет (куда идти, если метки нет?). Компилятору даётся указание **extern** (и описываются внешние объекты), тогда так компилятор поймет, что искать данные объекты в данном модуле не надо, их отсутствие ошибкой не является.

Если переменная (процедура, метка) описана в данном модуле, но ею будет пользоваться другой модуль (например, переходить по этой метке), данный факт тоже надо обозначить. Такой модуль без специальных указаний компилятору компилироваться будет (если есть процедура, которая не вызывается, это не ошибка), но при компиляции теряются все имена, данные пользователем, восстановить их уже будет нельзя.

Модули компилируются по отдельности (каждый – с помощью "своего" компилятора, в зависимости от того, на каком языке он написан). В результате компиляции получается так называемый объектный модуль. Выполняться объектные модули не могут. В нашем случае модули, написанные на Ассемблере, надо откомпилировать с помощью компилятора языка Ассемблер. Сделать это удобно в той папке, в которой вы выполняли программы на ассемблере, так как из нее удобным образом прописаны пути к нужным файлам. Модуль, написанный на Паскале надо откомпилировать компилятором Паскаля. Это удобно делать также в той папке, в которую вы помещали программы, которые запускали для работы на Паскале, так как именно для этой папке прописаны пути к нужным файлам для компилятора Паскаль.

Чтобы программа из нескольких модулей заработала, надо все объектные модули собрать в одну папку, в ту же папку поместить программу, называемую редактором внешних связей (линкер, линковщик). Понятно, что, если все модули на ассемблере, делать это надо в папке с ассемблерными программами. Если один из модулей на Паскале, удобнее переписать объектные файлы всех модулей в папку с Паскаль-программой, тогда сборку можно поручить редактору внешних связей Паскаля (обычный запуск из Паскаль-среды кнопкой RUN).

Отметим, что описанный здесь способ не является единственным, можно помещать модули в разные папки, аккуратно прописывая пути к нужным файлам. Можно компилировать Паскаль-программу не из Паскаль-среды, а из командной строки и т.д.

Примеры.

## 1. Головной и вспомогательный модуль на ассемблере

Головной модуль (вызывает процедуру, которая описана в другом модуле).

```
include console.inc
comment *
  головной модуль, вызывает процедуру на
  Ассемблере из другого модуля
*
.data
  X dd ?
.code
Start:
; в головном модуле (и только в нем) обязательно должна
; быть метка начала работы (повторяется после end)
  extrn PROST@0:near
; внешнее имя. К названию процедуры добавлено @0.
; Эта процедура описана во вспомогательном модуле,
; из головного она вызывается.
  outstrln 'Это работает головной модуль'
  inint X,'Введите число='
  outintln X,10,'Введено='
  push offset X
; параметр-адрес передается в процедуру через стек
  call PROST@0; К имени процедуры добавлено @0
;             Процедура выполняет действие X:=X+111
  outintln X,10,'Получено X='
; можно видеть, что процедура проработала, изменила значение X
  exit
  end Start; у головного модуля после end пишется метка,
;          с которой начинается работа программы
```

Вспомогательный модуль. В нем описана процедура, которая вызывается из головного модуля.

```
include console.inc
; вспомогательный модуль
.code
  public PROST; внешнее имя _prost@0
; Эта процедура будет вызываться из других модулей
PROST proc
; здесь всё обычным образом, имя процедуры без всяких добавок
  push ebp
  mov  ebp,esp;      стандартные действия входа в процедуру
  push ebx;          сохраняем регистр, он понадобится
```

```
  mov  ebx,[ebp+8]; это адрес переменной из стека
  add  dword ptr [ebx],111; увеличиваем переменную на 111
  pop  ebx
  pop  ebp
  ret 4; в стеке был один параметр
PROST endp
  end ; у вспомогательного модуля в конце стоит end без метки
```

## Компиляция модулей

В BAT-файле, которым мы пользуемся для выполнения программ на Ассемблере, содержатся команды для компиляции, сборки и выполнения программы. Модуль надо только откомпилировать (получить объектный файл). Это можно сделать с помощью такого BAT-файла. В первую строку надо записать имя модуля (без расширения). Если компиляция прошла успешно, на экране появится сообщение о создании объектного файла, а в папке появится файл с тем же именем и расширением `obj`.

```
@echo off
set Name=prost
set path=..\bin;..\..\bin
set include=..\include;..\..\include
set lib=..\lib;..\..\lib
ml /c /coff /Fl %Name%.asm
if errorlevel 1 goto errasm
echo ______ Now you have a object file _____
goto TheEnd
:errasm
echo Assembler Error !!!!!!!!!!!!
goto TheEnd
:TheEnd
pause
```

Когда все модули откомпилированы (имеются объектные файлы), их надо собрать (LINK), получить выполняемый файл (EXE) и выполнить его. Это можно сделать с помощью BAT-файла, показанного ниже. В первую и вторую строки надо записать имена модулей (без расширения), в третью строку – имя, которые Вы хотите дать получившемуся выполняемому файлу (может совпадать с именем одного из модулей). Если все пройдет успешно, в папке появится файл с заданным Вами именем и расширением EXE, а сформированная программа будет выполнена, результаты ее работы Вы увидите на экране.

```
@echo off
set Name1=mod_gl
set Name2=mmm
set NameRez=mod_mmm
set path=..\bin;..\..\bin
set include=..\include;..\..\include
set lib=..\lib;..\..\lib
link /subsystem:console /out:%NameRez%.exe %Name1%.obj %Name2%.obj
if errorlevel 1 goto errlink
echo Link is sucseccful
%NameRez%.exe
goto TheEnd
:errlink
echo Link Error !!!!!!!!!!!!!!!!
goto TheEnd
:TheEnd
pause
```

2. Головной модуль на Паскале, вспомогательный на ассемблере. Во вспомогательном модуле процедура, которая вызывается из программы на Паскале. Если при подготовке вспомогательного модуля выполнены все соглашения о связях, описанную в нем процедуру можно вызвать из Паскаля. Вспомогательный модуль транслируется точно также, подготавливается объектный файл, который помещается в папку с программой на Паскале.

Вот программа на Паскале, делающая то же самое, что головной модуль на ассемблере в предыдущем примере (вызывает процедуру из описанного выше модуля).

```
program Connect;
procedure PROST(var a:longint);
{  В процедуре 1 параметр, передается по ссылке, то есть
   в стеке должен быть адрес  }
stdcall;  {  такое соглашение о связях  }
external name '_PROST@0';
{  Внешнее имя у этой процедуры будет _PROST@0  }
{$L PROST.obj}
{  здесь имя файла, в котором находится скомпилированный модуль  }
  var ss:longint;
begin
  Write('Введите целое число=');
  Readln(ss); PROST(ss);
{  процедура вызывается так же, как если бы была
   описана в этой же программе на Паскале  }
  Write('Результат работы процедуры ',ss);
  readln
end.
```

Программы из примера 1 и из примера 2 работают одинаково, пользуясь одним и тем же модулем с процедурой.

3. Головной модуль на Паскале, вспомогательный на Ассемблере. Вызывается процедура с двумя параметрами, которые передаются один по ссылке, другой по значению. Параметры закладываются в стек справа налево. Таким образом, первый (самый левый параметр) будет в стеке по адресу [ebp+8], второй – [ebp+12] и т.д.

Головной модуль на Паскале

```
program Connect;
procedure SUMPR(p:longint; var s:longint);
  stdcall; external name '_SUMPR@0';
  {$L SUMPR.obj}
{  Процедура SUMPR в модуле SUMPR.OBJ,  внешнее имя  _Sumpr@0  }
  var ss:longint;
begin
  Writeln('Программа');
  ss:=23; SUMPR(345,ss);
  Writeln('Результат');
  Writeln(ss);
  readln
end.
```

Вспомогательный модуль с процедурой на ассемблере

```
include console.inc
comment *
  модуль для работы с Паскаль-программой
  Два параметра  sumpr(a,b)  Выполняется действие  b:=a+b
*
```

```
.code
  public SUMPR; внешнее имя _SUMPR@0
SUMPR proc
  push ebp
  mov  ebp,esp
  push eax
  push ebx
  mov  eax,[ebp+8];   первый параметр (слева)
  mov  ebx,[ebp+12]; второй параметр
  add [ebx],eax
  pop ebx
  pop eax
  pop ebp
  ret 8
SUMPR endp
  end.
```

4. Головной модуль на Паскале, вспомогательный – на ассемблере. Вызывается функция, описанная во вспомогательном модуле.

Все точно также. Значение функции возвращается на регистре AL, AX или EAX в зависимости от типа результата (byte, integer, longint).

Файл на Паскале

```
program Connect;
function Fun(a,b:longint):longint;
  stdcall; external name '_FUN@0';
{$L confu.obj} { Функция в модуле, внешнее имя _FUN@0 }
  var ss:longint;
begin
  Writeln('Программа');
  ss:=23;
  ss:=Fun(111,222);
  Writeln('Результат');
  Writeln (ss);
  readln
end.
```

Файл на ассемблере

```
include console.inc
comment *
  функция для работы с Паскаль-программой
  Вычитает из первого параметра второй
*
.code
  public FUN
FUN proc
  push ebp
  mov  ebp,esp
  mov  eax,[ebp+8];   FUN:=1-й параметр-значение
  sub  eax,[ebp+12]; FUN:=1-й параметр-2 параметр
  pop  ebp
  ret  8
FUN endp
  end
```

5. Оба модуля на ассемблере. Передают друг другу управление по меткам. Во вспомогательном модуле описана переменная, в нем она вводится, а в головном – печатается.

Головной модуль

```
include console.inc
comment *
   головной модуль передает управление вспомогательному,
   там вводится значение, которое выводится в головном модуле
*
.code
  extern VVOD:near;
  extern A:dword;  это имя описано в другом модуле
  public METKA
;  это имя описано здесь, будет использоваться в другом модуле
Beg:
  outstrln 'Работает головной модуль'
  outstrln 'Управление передается на вспомогательный модуль'
  jmp VVOD
METKA:
  outstrln 'Работает снова головной модуль'
  outstr 'Было введено значение '
  outintln A
  exit
  end Beg
```

Вспомогательный модуль

```
include console.inc
  public A;  внешнее имя _A
  public VVOD
  extern METKA: near;
.data
  A dd ?
.code
VVOD:
  outstrln 'Работает вспомогательный модуль'
  outstr 'Введите значение переменной='
  inint A
  outstrln 'Значение введено. Управление передается на головной модуль'
  jmp METKA
  end
```