|  |
| --- |
| Министерство образования и науки |
| **ЭВМ и периферийные устройства**  **(Turbo Debugger)**    лабораторный практикум |
|  |
|  |
|  |
|  |
|  |

|  |
| --- |
|  |

[Введение. 5](#_Toc285551922)

[Лекция 1. Архитектура ЭВМ 6](#_Toc285551923)

[Лекция 2. Микропроцессор. Характеристики микропроцессора. Структура универсального микропроцессора. 12](#_Toc285551924)

[Лекция 3. Организация памяти. Модели памяти. 14](#_Toc285551925)

[Лекция 4. Модели памяти. 19](#_Toc285551926)

[Лекция 5. Типы данных. Форматы команд. 23](#_Toc285551927)

[Лекция 6. Архитектура МП 8086. 29](#_Toc285551928)

[Лекция 7. Архитектура 32-разрядного универсального микропроцессора IA32 34](#_Toc285551929)

[Лекция 8. Методы адресации 38](#_Toc285551930)

[Лекция 9. Структура машинной команды. 45](#_Toc285551931)

[Лекция 10. Структура программы на ассемблере. 49](#_Toc285551932)

[Лекция 11. Система команд микропроцессора 8086. Команды пересылки данных 56](#_Toc285551933)

[Лекция 12. Целочисленные арифметические команды двоичной арифметики 60](#_Toc285551934)

[Лекция 13. Команды манипулирования битами 66](#_Toc285551935)

[Лекция 14. Команды передачи управления 73](#_Toc285551936)

[Лекция 15. Стек 79](#_Toc285551937)

[Лекция 16. Модульное программирование. Процедуры 82](#_Toc285551938)

[Лекция 18. Цепочки 91](#_Toc285551939)

[Лекция 19. Массивы 97](#_Toc285551940)

[Лекция 20. Структуры 101](#_Toc285551941)

[Лекция 21. Расширенные возможности современных микропроцессоров 103](#_Toc285551942)

[Лабораторный практикум 111](#_Toc285551943)

[Лабораторная работа №1. Этапы создания программы на ассемблере. Отладчик Turbo Debugger 111](#_Toc285551944)

[Лабораторная работа №2. Команды пересылки данных. Арифметические команды 118](#_Toc285551945)

[Лабораторная работа №3 . Команды для работы с битами. 124](#_Toc285551946)

[Лабораторная работа №4 . Массивы 130](#_Toc285551947)

[Лабораторная работа №5. Команды передачи управления 139](#_Toc285551948)

[Лабораторная работа №6. Прерывания. 146](#_Toc285551949)

[Лабораторная работа №7. Модульное программирование. Подпрограммы 154](#_Toc285551950)

[Лабораторная работа №8. Стек. 168](#_Toc285551951)

[Лабораторная работа №9. Цепочки 173](#_Toc285551952)

[Лабораторная работа №10. Структуры 184](#_Toc285551953)

[Лабораторная работа №11. Файлы 193](#_Toc285551954)

[Литература 201](#_Toc285551955)

# Введение.

На протяжении многих лет крупнейшим разработчиком и производителем универсальных микропроцессоров в мире является компания Intel (70-75 % мирового производства). Микропроцессоры корпорации Intel и персональные компьютеры на их базе прошли не очень длинный во времени, но значительный по существу путь развития, на протяжении которого кардинально изменялись и возможности и даже сами принципы их архитектуры

Первый микропроцессор был разработан фирмой Intel в 1971 году. Он получил название I-4004, имел 4-разрядную структуру и был ориентирован на использование в калькуляторах. Впоследствии этой же фирмой был выпущен еще один 4-разрядный микропроцессор - I-4040.

В 1972 году на рынке появился 8-разрядный МП I-8008, а вслед за ним, в 1974 году,- I-8080. Последний микропроцессор сыграл значительную роль в развитии микропроцессорной техники. Во многом он заложил основы архитектуры для всех последующих поколений микропроцессоров.

Очередным крупным шагом в развитии микропроцессорной техники стало появление в 1978 году 16-разрядных универсальных микропроцессоров. Здесь, прежде всего, следует выделить микропроцессор **I-8086.** Этот микропроцессор, заложивший основы архитектуры **x86**, использовался при производстве первых персональных ЭВМ.

Основными отличительными чертами в архитектуре этого микропроцессора стали:

* увеличение разрядности регистров общего назначения до 16 бит;
* увеличение количества регистров общего назначения до 8;
* увеличение количества режимов адресации операндов;
* расширение количества флагов в регистре признаков, в том числе за счет введения флагов управления, обеспечивающих, например, возможность запрета внешних маскируемых прерываний;
* появление сегментного механизма обращения к памяти, который обеспечил возможность обращения к памяти емкостью до 1 Мбайт при использовании 16-разрядных регистров.

Появившийся вслед за этим в 1982 году микропроцессор **i286** явился переходной ступенью к 32-разрядным универсальным микропроцессорам. В новом процессоре было увеличено количество регистров, добавлены новые инструкции, добавлен новый режим работы процессора — ***защищённый режим.*** Однако процессор 80286 мог работать и в режиме 86-го процессора, который стали называть реальным.

Для защиты от выполнения привилегированных команд, которые могут кардинально изменить состояние всей системы, для защиты доступа к данным и для защиты сегментов кода в процессоре i286 была введена **защита по привилегиям.**

В 1985 году был выпущен 32-разрядный универсальный **микропроцессор i386** - первый полноценный представитель архитектуры IA-32 (Intel Architecture-32). В нем появилась возможность адресовать до 4GiB памяти, что позволило создавать сегменты памяти размером во все адресное пространство.

В дальнейшем на смену процессору 80386 пришли модели i486, различные варианты процессора Pentium. Все они могут работать и в реальном, и в защищенном режимах. Хотя каждая следующая модель была значительно совершеннее предыдущей, однако с точки зрения программиста все эти процессоры весьма схожи.

При этом следует подчеркнуть, что хотя процессор 8086, как микросхема, уже давно не используется, его архитектура, и система команд целиком вошли в современные процессоры. Лишь относительно небольшое число команд современных процессоров специально предназначены для организации защищенного режима и распознаются процессором, только когда он работает в защищенном режиме. Поэтому изучение языка ассемблера целесообразно начинать с изучения архитектуры процессора 8086 или, точнее, того гипотетического процессора, который как бы объединяет часть архитектур средств современных процессоров, предназначенных для использования в реальном режиме, и соответствующих архитектуре процессора 8086.

# Лекция 1. Архитектура ЭВМ

## Понятие архитектуры ЭВМ.

**Система** (от греческого слова “целое”) – это совокупность взаимодействующих элементов, образующих определенное единство. Существует ряд иных определений, из которых мне более всего импонирует следующее: система состоит из подсистем, обладающих всеми свойствами системы.

Под **информационной системой (ИС)** понимается система, организующая, хранящая и преобразовывающая информацию (это определение дается с точки зрения выполняемых функций).

Основу любой автоматизированной ИС составляет ЭВМ (отечественная терминология) или компьютер (зарубежная терминология). ЭВМ – комплекс технических средств, предназначенных для автоматической обработки информации в процессе решения вычислительных и информационных задач.

В 70-х годах 20 века ЭВМ превратились сначала в вычислительные системы, а затем в информационно-вычислительные системы.

**Вычислительная система (ВС)** **-** это совокупность одного или нескольких компьютеров или процессоров, программного обеспечения и периферийного оборудования, организованная для выполнения информационно-вычислительных процессов.

При рассмотрении компьютерных устройств, принято различать их архитектуру и структуру.

**Архитектурой компьютера** называется его описание на некотором общем уровне, включающее описание пользовательских возможностей программирования, системы команд, системы адресации, организации памяти и т.д. Архитектура определяет принципы действия, информационные связи и взаимное соединение основных логических узлов компьютера: процессора, оперативного ЗУ, внешних ЗУ и периферийных устройств. Общность архитектуры разных компьютеров обеспечивает их совместимость с точки зрения пользователя.

## 

## Архитектура фон Неймана.

В основу построения подавляющего большинства компьютеров положены следующие общие принципы, сформулированные в 1945 г. американским ученым Джоном фон Нейманом.

1. Принцип двоичности**.**

Для представления данных и команд используется двоичная система счисления.

2. Принцип программного управления. Из него следует, что программа состоит из набора команд, которые выполняются процессором автоматически друг за другом в определенной последовательности.

Выборка программы из памяти осуществляется с помощью счетчика команд. Этот регистр процессора последовательно увеличивает хранимый в нем адрес очередной команды на длину команды.

А так как команды программы расположены в памяти друг за другом, то тем самым организуется выборка цепочки команд из последовательно расположенных ячеек памяти.

Если же нужно после выполнения команды перейти не к следующей, а к какой-то другой, используются команды условного или безусловного переходов, которые заносят в счетчик команд номер ячейки памяти, содержащей следующую команду. Выборка команд из памяти прекращается после достижения и выполнения команды “стоп”.

Таким образом, процессор исполняет программу автоматически, без вмешательства человека.

3. Принцип однородности памяти. Программы и данные хранятся в одной и той же памяти. Поэтому компьютер не различает, что хранится в данной ячейке памяти — число, текст или команда. Над командами можно выполнять такие же действия, как и над данными. Это открывает целый ряд возможностей. Например, программа в процессе своего выполнения также может подвергаться переработке, что позволяет задавать в самой программе правила получения некоторых ее частей (так в программе организуется выполнение циклов и подпрограмм). Более того, команды одной программы могут быть получены как результаты исполнения другой программы. На этом принципе основаны методы трансляции — перевода текста программы с языка программирования высокого уровня на язык конкретной машины.

4. Принцип адресности. Структурно основная память состоит из перенумерованных ячеек; процессору в произвольный момент времени доступна любая ячейка. Отсюда следует возможность давать имена областям памяти, так, чтобы к запомненным в них значениям можно было впоследствии обращаться или менять их в процессе выполнения программ с использованием присвоенных имен.

Компьютеры, построенные на этих принципах, относятся к типу фон-неймановских.

Архитектура первой ЭВМ предусматривала наличие в составе пяти функциональных устройств, обязательных для архитектуры фон Неймана. Это *арифметико-логическое устройство (АЛУ), запоминающее устройство (ЗУ), входное устройство, выходное устройство* и *устройство управления*.

АЛУ

ЗУ

Увыв

Увв

УУ

Рис.1.1.

*АЛУ* реализует выполнение команд, составляющих программу, используя предусмотренный в нем *набор базовых операций* (арифметических, логических, условного перехода и т.п.).

*ЗУ* (*память*) хранит программы, исходные и текущие данные. Место данных и команд в памяти определяется их *адресом* (адресуемая память) – номером ячейки ЗУ. Содержимое ячейки при чтении не меняется, при записи старое стирается и заменяется на новое. В командах-преобразователях указываются не сами операнды, а их адреса. Это придает программе универсальность формулы, в которую можно подставлять различные данные.

*Входное устройство* предназначено для ввода в ЭВМ информации, необходимой для решения задач. Это программы, исходные данные, управляющие символы, представленные в двоичных кодах.

*Выходное устройство* обеспечивает вывод получаемых результатов, представленных в удобной для пользователя форме, а также специальных управляющих символов.

*Устройство управления* организует:

-взаимодействие АЛУ с ЗУ (поочередную передачу команд программы в АЛУ, поиск в памяти нужных для выполнения очередной команды данных, запись получаемых результатов в память);

-прием информации от пользователя;

-выдачу информации пользователю.

Пройдя пять поколений ЭВМ, архитектура фон Неймана сохраняется и в современных микропроцессорных вычислительных системах в качестве «ядра», вокруг которого создается «интеллектуальная» оболочка.

В основе организации современных цифровых вычислительных систем лежит так называемый *магистрально-модульный* принцип. Суть его заключается в наличии набора магистралей (шин), к которым свободно подсоединяются функциональные модули.

Микропроцессор

Арифметико-логическое устройство (АЛУ)

Микропроцессорная память

Устройство управления

Интерфейсная система

С

И

С

Т

Е

М

Н

А

Я

Ш

И

Н

А

Основная память

ПЗУ ОЗУ

видео

аудео

нжмд

нгмд

принтер

модем

*Центральный процессор (CPU, от англ. Central Processing Unit)* **—** это основной рабочий компонент компьютера, который выполняет арифметические и логические операции, заданные программой, управляет вычислительным процессом и координирует работу всех устройств компьютера**.**

Современные процессоры выполняются в виде*микропроцессоров.*

В состав микропроцессора входят:

  - устройство управления  (УУ) - формирует и подаёт во все блоки машины, в нужные моменты времени определённые сигналы управления; формирует адреса ячеек памяти и передаёт эти адреса в соответствующие блоки ЭВМ;

- арифметико-логическое устройство (АЛУ) - предназначено для выполнения всех арифметических и логических операций над числовой и символьной информацией;

- микропроцессорная память (МПП) - служит для кратковременного хранения, записи и выдачи информации, непосредственно используемой в вычислениях в ближайшие такты работы машины. Микропроцессорная память строится на  регистрах и используется для обеспечения высокого быстродействия машины;

-  интерфейсная система микропроцессора  - реализует связь с другими устройствами персонального компьютера.

*Основная память (ОП)* предназначена для хранения и оперативного обмена информацией с другими блоками машины.

Основная память содержит два вида запоминающих устройств:

  - постоянное запоминающее устройство (ПЗУ) служит для хранения неизменяемой программной и справочной информации. ПЗУ состоит из специальных чипов, информация из которых, будучи однажды записана, в дальнейшем только считывается. В силу этого свойства, записанные в таких чипах данные и инструкции не могут быть изменены (в английской транскрипции ПЗУ называется ROM – Read Only Memory, что означает “память только для чтения”). Расширенное ПЗУ включает постоянную BIOS и отвечает за устройства ввода/вывода, такие, как контроллер (устройство управления) винчестера и др. Основное ПЗУ системы обеспечивает самопроверку при включении компьютера, загрузку операционной системы с диска, рисование точек при выводе графики. Когда Вы включаете компьютер, ПЗУ управляет различными проверками и загрузкой специальных данных с диска в ОЗУ (заметим, что для программиста эта память недоступна). Микросхема постоянного запоминающего устройства способна длительное время хранить информацию, даже когда компьютер выключен;

- оперативное запоминающее устройство (ОЗУ) - это массив кристаллических ячеек, способных хранить данные. ОЗУ доступна программисту в качестве пространства для временного хранения данных и выполнения программ. При включении компьютера часть операционной системы загружается с винчестера в оперативную память. Остальная память из основного ОЗУ может использоваться программистом. Выполняемая прикладная программа находится в этой части ОЗУ и выводит результаты обработки на экран дисплея, принтер или устройства внешней памяти (например, НГМД или винчестер). После завершения одной прикладной программы операционная система может загрузить на ее место другую программу. Выключение компьютера приводит к потере данных в ОЗУ, но не влияет на ПЗУ. Вот почему требуемые для дальнейшей обработки данные необходимо сохранять в долговременной памяти.

Чтобы компьютер мог работать, необходимо, чтобы в его оперативной памяти находились программа и данные. А попадают они туда из различных устройств компьютера - клавиатуры, дисководов для магнитных дисков и т.д. Обычно эти устройства называют внешними, хотя некоторые из них могут находиться не снаружи компьютера, а встраиваться внутрь системного блока. Результаты выполнения программ выводятся на внешние устройства - монитор, диски, принтер и т.д. Таким образом, для работы компьютера необходим обмен информацией между оперативной памятью и внешними устройствами. Такой обмен называется *вводом-выводом*. Но этот обмен не происходит непосредственно: между любым внешним устройством и оперативной памятью в компьютере имеются целых два промежуточных звена:

1.Для каждого внешнего устройства в компьютере имеется электронная схема, которая им управляет. Эта схема называется контроллером, или адаптером.*Контроллеры и адаптеры* представляют собой наборы электронных цепей, которыми снабжаются устройства компьютера с целью совместимости их интерфейсов. Контроллеры, кроме этого, осуществляют непосредственное управление периферийными устройствами по запросам микропроцессора. Некоторые контроллеры (например, контроллер дисков) могут управлять сразу несколькими устройствами.

2.Все контроллеры и адаптеры взаимодействуют с микропроцессором и оперативной памятью через системную магистраль передачи данных, которую в просторечии обычно называют *шиной.*

*Шина - это совокупность линий, по которым информация передаётся из одного устройства в другое.*

Системная шина включает в себя: кодовую шину данных (КШД), кодовую шину адреса (32-разрядная), кодовую шину инструкций (управления).

Системная шина обеспечивает три направления передачи информации между:

- микропроцессором и основной памятью;

- микропроцессором и портами ввода-вывода внешних устройств;

- основной памятью и портами ввода-вывода внешних устройств.

Каждый из функциональных элементов (память, монитор или другое устройство) связан с шиной определённого типа — адресной, управляющей или шиной данных.

Выделим **общие архитектурные свойства** **ЭВМ,** присущие большинству типов машин так называемой фон-неймоновской архитектуры [Юров,2002,с.28-29]:

(1)  принцип хранимой программы. Согласно ему, код программы и её данные находятся в одном адресном пространстве в оперативной памяти;

(2)  принцип микропрограммирования. Суть этого принципа заключается в том, что машинный язык всё-таки ещё не является той конечной субстанцией, которая физически приводит в действие процессы в машине. В состав процессора входит  блок микропрограммного управления. Этот блок для каждой машинной команды имеет набор действий-сигналов, которые нужно сгенерировать для физического выполнения требуемой машинной команды;

(2) линейное пространство памяти - совокупность ячеек памяти, которым последовательно присваиваются номера (адреса) 0, 1, 2,...;

(3) последовательное выполнение программ. Процессор выбирает из памяти команды строго последовательно. Для изменения прямолинейного кода выполнения программы или осуществления ветвления необходимо использовать специальные команды, которые называются  командами условного и безусловного перехода;

(4) с точки зрения процессора нет принципиальной разницы между данными и командами. Данные и машинные команды находятся в одном пространстве памяти в виде последовательности нулей и единиц. Это свойство связано с предыдущим. Процессор, исполняя содержимое некоторых последовательных ячеек памяти, всегда пытается трактовать его как коды машинной команды, а если это не так, то происходит аварийное завершение программы, содержащей некорректный фрагмент. Поэтому важно в программе всегда чётко разделять пространство данных и команд;

(5)  безразличие к целевому назначению данных. Машине всё равно, какую логическую нагрузку несут обрабатываемые ею данные.

Следует отметить, что существует также и альтернативный тип архитектуры - *гарвардская архитектура*. Гарвардская архитектура отличается от архитектуры фон Неймана тем, что программный код и данные хранятся в разной памяти. В такой архитектуре невозможны многие методы программирования (например, программа не может во время выполнения менять свой код; невозможно динамически перераспределять память между программным кодом и данными); зато гарвардская архитектура позволяет более эффективно выполнять работу в случае ограниченных ресурсов, поэтому она часто применяется во встраиваемых системах.

# Лекция 2. Микропроцессор. Характеристики микропроцессора. Структура универсального микропроцессора.

**Микропроцессор** (МП) - это программно управляемое устройство, которое предназначено для обработки цифровой информации и управления процессом этой обработки.

Микропроцессор является "мозгом" компьютера. Он осуществляет выполнение программ, работающих на компьютере, и управляет работой остальных устройств компьютера. Скорость его работы во многом определяет быстродействие компьютера.

По своей архитектуре микропроцессоры разделяются на несколько типов.

**Универсальные микропроцессоры** предназначены для решения задач цифровой обработки различного типа информации от инженерных расчетов до работы с базами данных, не связанных жесткими ограничениями на время выполнения задания.

**Однокристальные микроконтроллеры** (ОМК или просто МК) предназначены для использования в системах промышленной и бытовой автоматики.

**Секционированные микропроцессоры** (другие названия: микропрограммируемые и разрядно-модульные) - это микропроцессоры, предназначенные для построения специализированных процессоров.

Большинство современных процессоров для персональных компьютеров в общем основаны на той или иной версии циклического процесса последовательной обработки данных, изобретённого Джоном фон Нейманом.

Этапы цикла выполнения:

1. Процессор выставляет число, хранящееся в регистре счётчика команд, на шину адреса и отдаёт памяти команду чтения.
2. Выставленное число является для памяти адресом; память, получив адрес и команду чтения, выставляет содержимое, хранящееся по этому адресу, на шину данных и сообщает о готовности.
3. Процессор получает число с шины данных, интерпретирует его как команду (машинную инструкцию) из своей системы команд и исполняет её.
4. Если последняя команда не является командой перехода, процессор увеличивает на единицу (в предположении, что длина каждой команды равна единице) число, хранящееся в счётчике команд; в результате там образуется адрес следующей команды.

Данный цикл выполняется неизменно, и именно он называется ***процессом*** (откуда и произошло название устройства).

Универсальные микропроцессоры принято разделять на **CISC**- и **RISC-микропроцессоры.** **CISC-микропроцессоры** (completed instruction set computing - вычисления с полной системой команд) имеют в своем составе весь классический набор команд с широко развитыми режимами адресации операндов. Именно к этому классу относятся, например, микропроцессоры семейства Intel x86 и Pentium. В то же время **RISC-микропроцессоры** (reduced instruction set computing - вычисления с сокращенной системой команд) используют, как следует из определения, уменьшенное количество команд и режимов адресации.

## Характеристики процессора

Основными характеристиками процессора являются его ***тактовая частота, разрядность и размеры кэша 1-го и 2-го уровня.***

***Тактовая частота***

Частота — это количество колебаний в секунду. Тактовая частота — это количество тактов в секунду. В применении к процессору:

Тактовая частота — это количество операций, которое процессор может выполнить в секунду.

Т.е. чем больше операций в секунду может выполнять процессор, тем быстрее он работает. Например, процессор с тактовой частотой 40 МГц выполняет 40 миллионов операций в секунду, с частотой 300 Мг — 300 миллионов операций в секунду, с частотой 1 ГГц - 1 миллиард операций в секунду.

Существует два типа тактовой частоты — внутренняя и внешняя.

Внутренняя тактовая частота — это тактовая частота, с которой происходит работа внутри процессора.

Внешняя тактовая частота или частота системной шины — это тактовая частота, с которой происходит обмен данными между процессором и оперативной памятью компьютера.

Разрядность шины данных говорит о том, какое количество информации (сколько байт) можно передать за раз (за такт). От разрядности шины адреса зависит максимальный объем оперативной памяти, с которым процессор может работать вообще.

До 1992 года в процессорах внутренняя и внешняя частоты совпадали, а в 1992 году компания Intel представила процессор 80486DX2, в котором внутренняя и внешняя частоты были различны — внутренняя частота была в 2 раза больше внешней. С этого времени остальные компании-производители также стали выпускать процессоры с удвоенной внутренней частотой, а компания IBM стала выпускать процессоры с утроенной внутренней частотой (25/75 МГц, 33/100 МГц и 40/120 МГц).

В современных процессорах, например, при тактовой частоте процессора 3 ГГц, частота системной шины 800 МГц.

***Разрядность процессора*** определяется разрядностью его регистров.

Компьютер может оперировать одновременно ограниченным набором единиц информации. Этот набор зависит от разрядности внутренних регистров. Разряд — это хранилище единицы информации. За один рабочий такт компьютер может обработать количество информации, которое может поместиться в регистрах. Если регистры могут хранить 8 единиц информации, то они 8-разрядне, и процессор 8-разрядный, если регистры 16-разрядные, то и процессор 16-разрядный и т.д. Чем большая разрядность процессора, тем большее количество информации он может обработать за один такт, а значит, тем быстрее работает процессор.

На мощность (производительность) процессора влияют не только его тактовая частота и разрядность шины данных, также важное значение имеет ***объем кэш-памяти.***

Кэш-память — это высокоскоростная память произвольного доступа, используемая процессором компьютера для временного хранения информации.

# Лекция 3. Организация памяти. Модели памяти.

**Оперативная память** компьютера представляет собой электронное устройство, состоящее из большого числа двоичных запоминающих элементов, а также схем управления ими. Минимальный объем информации, к которому имеется доступ в памяти, составляет один *байт* (8 двоичных разрядов, или битов).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

7 6 5 4 3 2 1 0

Байты условно нумеруются от нуля и располагаются (при их изображении на бумаге) в порядке возрастания номера справа налево, так что слева оказываются байты с большими номерами, а справа - байты с меньшими номерами. Крайний слева байт принято называть *старшим,* а крайний справа - *младшим.* Следующее число, если его написать за предыдущим, опять начнется со старшего разряда и закончится младшим. *Однако в памяти компьютера данные располагаются в более естественном порядке непрерывного возрастания номеров байтов и, таким образом, каждое слово или двойное слово в памяти начинается с его младшего байта и заканчивается старшим.*

Нужные байты отыскиваются в памяти по их номерам, выполняющим функции адресов.

Некоторые данные (например, коды символов) требуют для своего хранения одного байта, для других данных этого места не хватает и под них в памяти выделяется 2, 4, 8 или еще большее число байтов. Обычно пары байтов называют *словами,* а четверки - *двойными словами***.** Адресом слова  называется адрес его первого байта (с меньшим адресом).

Двоичная система счисления, в которой работают все цифровые электронные устройства, неудобна для человека. Для удобства представления двоичного содержимого ячеек памяти или регистров процессора используют иногда восьмеричную, а чаще - шестнадцатеричную системы счисления. Для процессоров Intel используется шестнадцатеричная система.

Каждый разряд шестнадцатеричного числа может принимать 16 значений, из которых первые 10 обозначаются обычными десятичными цифрами, а последние 6 - буквами латинского алфавита от А до F, где А обозначает 10, В - И, С - 12, D - 13, Е - 14, a F - 15. Поскольку одна шестнадцатеричная цифра требует для записи ее в память компьютера четырех двоичных разрядов, то содержимое байта описывается двумя шестнадцатеричными цифрами (от 00h до FFh, или от 0 до 255) , а содержимое слова - четырьмя (от 0000h до FFFFh, или от 0 до 65535).

Помимо ячеек оперативной памяти, для хранения данных используются еще запоминающие ячейки, расположенные в процессоре и называемые регистрами. Достоинство регистров заключается в их высоком быстродействии, гораздо большем, чем у оперативной памяти, а недостаток в том, что их очень мало - всего около десятка. Поэтому регистры используются лишь для кратковременного хранения данных. В режиме МП 86, который мы здесь обсуждаем, все регистры процессора имеют длину 16 разрядов, или 1 слово (в действительности в современных процессорах их длина составляет 32 разряда, но в МП 86 от каждого регистра используется лишь его половина). За каждым регистром закреплено определенное имя (например, АХ или DS), по которому к нему можно обращаться в программе. Состав и правила использования регистров процессора будут подробно описаны ниже.

Диапазон адресов, формируемых процессором, называют **адресным пространством процессора**.

В зависимости от модификации персонального компьютера и состава его периферийного оборудования, распределение адресного пространства может несколько различаться. Тем не менее, размещение основных компонентов системы довольно строго унифицировано. Типичная схема использования адресного пространства компьютера приведена на рис. Значения адресов на этом рисунке даны в шестнадцатеричной системе счисления.

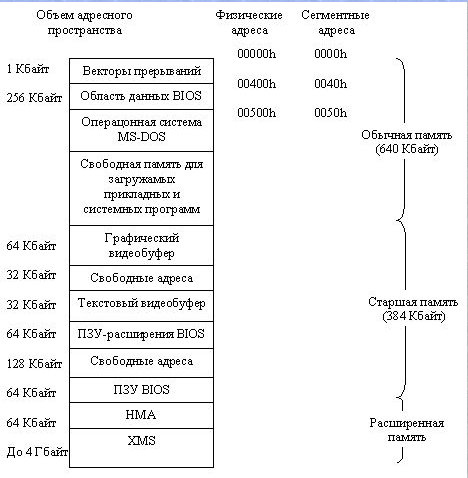


Рис.Типичное распределение адресного пространства.

Первые 640 Кбайт адресного пространства с адресами от 00000h до 9FFFF11 отводятся под основную оперативную память, которую еще называют стандартной (conventional). Начальный килобайт оперативной памяти занят векторами прерываний, которые обеспечивают работу системы прерываний компьютера, и включает 256 векторов по 4 байта каждый. Вслед за векторами прерываний располагается так называемая область данных BIOS, которая занимает всего 256 байт, начиная с сегментного адреса 40h. Сама BIOS (от Basic In-Out System, базовая система ввода-вывода) является частью операционной системы, хранящейся в постоянном запоминающем устройстве. Это запоминающее устройство (ПЗУ BIOS) располагается на системной плате компьютера и является, таким образом, примером встроенного, или "зашитого" программного обеспечения. В функции BIOS входит тестирование компьютера при его включении, загрузка в оперативную память собственно операционной системы MS-DOS, хранящейся на магнитных дисках, а также управление штатной аппаратурой компьютера - клавиатурой, экраном, дисками и прочим. В области данных BIOS хранятся разнообразные данные, используемые программами BIOS в своей работе. Так, здесь размещаются:

* входной буфер клавиатуры, куда поступают коды нажимаемых пользователем клавиш;
* адреса видеоадаптера, а также последовательных и параллельных портов;
* данные, характеризующие текущее состояние видеосистемы (форма курсора и его текущее положение на экране, видеорежим, используемая видеостраница и проч.);
* ячейки для отсчета текущего времени и т.д.

Область данных BIOS заполняется информацией в процессе начальной загрузки компьютера, а затем динамически модифицируется системой по мере необходимости. Многие прикладные программы, особенно, написанные на языке ассемблера, обращаются к этой области с целью чтения или модификации содержащихся в них данных.

В области памяти, начиная с адреса 500h, располагается собственно операционная система MS-DOS, которая обычно занимает несколько десятков Кбайт. Программы MS-DOS, как и другие системные составляющие (векторы прерываний, область данных BIOS) записываются в память автоматически в процессе начальной загрузки компьютера.

Вся оставшаяся память до границы 640 Кбайт свободна для загрузки любых системных или прикладных программ. Как правило, в начале сеанса в память загружают резидентные программы (русификатор, антивирусные программы). При наличии резидентных программ объем свободной памяти уменьшается.

Оставшиеся 384 Кбайт адресного пространства между границами 640 Кбайт и 1 Мбайт, называемые старшей, или верхней (upper) памятью, первоначально были предназначены для размещения постоянных запоминающих устройств (ПЗУ). Практически под ПЗУ занята только небольшая часть адресов, а остальные используются в других целях.  
Часть адресного пространства старшей памяти отводится для адресации к графическому и текстовому видеобуферам графического адаптера.

В самом конце адресного пространства, в области адресов F0000h...FFFFFh, располагается ПЗУ BIOS - постоянное запоминающее устройство, о котором уже говорилось выше.

Часть адресного пространства, начиная с адреса C0000h, отводится еще под одно ПЗУ - так называемое ПЗУ расширений BIOS для обслуживания графических адаптеров и дисков.

В состав компьютера, наряду со стандартной памятью (640 Кбайт), входит еще расширенная (extended) память, максимальный объем которой может доходить до 4 Гбайт. Эта память располагается за пределами первого мегабайта адресного пространства и начинается с адреса 100000h. Поскольку функционирование расширенной памяти подчиняется "спецификации расширенной памяти" (Extended Memory Specification, сокращенно XMS), то и саму память часто называют XMS-памятью. Доступ к расширенной памяти осуществляется в защищенном режиме, поэтому для MS-DOS, работающей только в реальном режиме, расширенная память недоступна. Однако в современные версии MS-DOS включается драйвер HIMEM.SYS, поддерживающий расширенную память, т.е. позволяющий ее использовать, хотя и ограниченным образом. Первые 64 Кбайт расширенной памяти, точнее, 64 Кбайт - 16 байт с адресами от l00000h до l0FFEFh, носят специальное название область старшей памяти (High MemoryArea, HMA).

## Режимы работы микропроцессора

Микропроцессор имеет сложную систему управления памятью, работа которой зависит от режима работы микропроцессора.

Особый интерес представляют три режима работы микропроцессора: реальный, защищенный и режим виртуального МП i8086.

В **реальном режиме** обеспечивается совместимость на уровне объектных кодов с микропроцессором i8086 и микропроцессором i286, работающих в реальном режиме. В этом режиме архитектура 32-разрядного микропроцессора почти полностью идентична архитектуре 16-разрядного МП. Для программиста же он вообще представляется как МП i8086, выполняющий написанные программы с большей скоростью и обладающий расширенной системой команд и регистрами. Благодаря этим качествам фирма Intel сохранила прежних клиентов, которые хотели модернизировать свои системы, не отказываясь от имевшегося задела в области программного обеспечения, и привлекла тех, кому изначально требовалась высокая скорость обработки информации.

Одно из основных ограничений реального режима было связано с предельной емкостью адресуемой памяти, равной 1 Мбайт. От него свободен **защищенный режим,** позволяющий воспользоваться всеми преимуществами архитектуры нового МП. Размер адресного пространства в этом случае увеличивается до 4 Гбайт, а общий объем поддерживаемого адресного пространства - до 64 терабайт (1 Тбайт = 240 байт). МП, работающие в защищенном режиме, обладают более высоким быстродействием и возможностями организации истинной многозадачности.

Наконец, **режим виртуального МП** открывает возможность одновременного исполнения программ, написанных для МП i8086, i286 и i386. Поскольку емкость памяти, адресуемой микропроцессором, не ограничена значением 1 Мбайт, этот режим позволяет формировать несколько виртуальных сред i8086.

В новых поколениях МП Intel появился еще один режим работы - **режим системного управления**. Впервые он был реализован в МП 80386SL и i486SL. Начиная с расширенных моделей Intel-486, этот режим стал обязательным элементом архитектуры IA-32. Он обеспечивает операционную систему механизмом для выполнения машинно-зависимых функций, таких как перевод компьютера в режим пониженного энергопотребления или выполнение действий по защите системы. Функционирование микропроцессора в этом режиме подобно его работе в режиме реальных адресов.

# Лекция 4. Модели памяти.

В зависимости от режима работы, микропроцессор аппаратно поддерживает две модели использования оперативной памяти:

* ***сегментированную модель;***
* ***страничную модель.***

## Сегментированная модель памяти.

**Сегментация** [Юров,2002] - это механизм адресации, обеспечивающий существование нескольких независимых адресных пространств как в пределах одной задачи, так и в системе в целом для защиты задач от взаимного влияния.

Суть сегментной адресации заключается в следующем.

Каждая программа в общем случае может состоять из любого количества сегментов. Сегменты могут располагаться в памяти по отношению друг к другу произвольным образом, ограничений на их расположение не вводится. Например, все сегменты могут начинаться с одного и того же адреса или непосредственно друг за другом. Операционная система размещает сегменты программы в оперативной памяти по определённым физическим адресам, после чего помещает значения этих адресов в определённые места (зависит от режима работы микропроцессора). Например, в реальном режиме они помещаются в соответствующие сегментные регистры, и называются  сегментными адресами.

Существуют три модели сегментированной организации памяти:

* ***сегментированная модель памяти реального режима;***
* ***сегментированная модель памяти защищенного режима;***
* ***сплошная модель памяти защищенного режима.***

### Формирование линейного адреса в реальном режиме.

Рассмотрим процесс формирования физического (линейного) адреса в реальном режиме.

В **реальном режиме** сегментные регистры процессора содержат старшие 16 бит физического адреса начала сегмента**.** Максимальное значение, которое могут содержать 16-битные регистры, составляет 216-1 или 64 Кбайт. Но адресное пространство реального режима составляет 1 Мбайт.

Возникает вопрос о том, как адресовать остальную часть оперативной памяти вплоть до 1 Мбайта, если размер самого сегмента не превышает 64 Кбайт. Чтобы организовать работу с мегабайтным адресным пространством памяти, используется искусственное выделение из адресного пространства сегментов по 64К (так как в таком объеме мы можем использовать шестнадцатиразрядные адреса). Недостающие младшие четыре бита 20-битного адреса получаются сдвигом значения в сегментном регистре на 4 разряда влево.

Физический адрес получается путем сложения этого адреса с 16-разрядным значением смещения в сегменте, формируемого по заданному режиму адресации для операнда или извлекаемому из регистра IP для команды

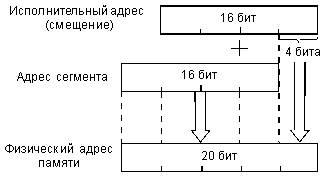
****

Рис.4.1. Формирование физического адреса памяти из адреса сегмента и смещения.

На рисунке видно, что сегментная часть извлекается из одного из сегментных регистров, сдвигается на четыре разряда влево и суммируется со смещением. Получившиеся 20-битное значение и является настоящим физическим адресом, соответствующим началу сегмента.

Процедура умножения сегментного адреса на 16 (или, что то же самое, на 10h) является принципиальной особенностью реального режима, ограничивающей диапазон адресов, доступных в реальном режиме, величиной 1 Мбайт. Действительно, максимальное значение сегментного адреса составляет FFFFh, или 64К-1, из чего следует, что максимальное значение начального адреса сегмента в памяти равно FFFF0h, или 1 Мбайт - 16. Если, однако, учесть, что к начальному адресу сегмента можно добавить любое смещение в диапазоне от 0 до FFFFh, то адрес последнего адресуемого байта окажется равен 10FFEFh, что соответствует величине 1 Мбайт + 64 Кбайт - 17.

### Формирование линейного адреса в защищенном режиме

При переходе к 32-разрядной архитектуре стало необходимым обеспечить возможность адресации памяти емкостью до 232 байт. Кроме того, введение защищенного режима работы микропроцессора потребовало хранения большого количества дополнительной информации о сегменте: его длине, которая стала переменной, уровне привилегий, его типе и т. д. Простое увеличение разрядности сегментных регистров до 32 бит не обеспечило бы возможности хранения всей этой информации. Поэтому все данные о сегменте стали размещаться в специальных структурах - **дескрипторах (описателях) сегментов**, которые хранятся в **таблицах дескрипторов**, расположенных в памяти, а сегментные регистры, сохранив свою первоначальную длину в 16 разрядов, содержат так называемый **селектор** (указатель), который используется для того, чтобы найти нужный дескриптор в этих таблицах.

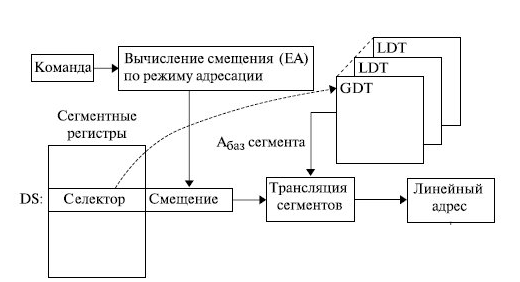


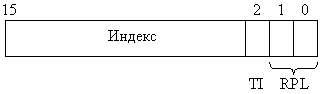
Рис.4.2. Формирование линейного адреса в защищенном режиме

Существуют две обязательные дескрипторные таблицы - глобальная (GDT) и дескрипторная таблица прерывания (IDT), а также множество (до 8192) локальных дескрипторных таблиц (LDT), из которых в один момент времени процессору доступна только одна. Дескрипторы сегментов могут находиться в GDT или LDT. Расположение дескрипторных таблиц определяется регистрами процессора GDTR (Global Descriptor Table Register), IDTR (Interrupt Descriptor Table Register), LDTR (Local Descriptor Table Register).

Регистры GDTR и IDTR - 6-байтные, они содержат 32 бита линейного базового адреса дескрипторной таблицы и 16 бит предела таблицы.

Программно доступная часть регистра LDTR - 16 бит, которые являются селектором LDT. Дескрипторы LDT находятся в GDT. Однако чтобы не обращаться каждый раз к GDT, в процессоре имеется теневая (программно недоступная) часть регистра LDTR, в которую процессор помещает дескриптор LDT при каждой перегрузке селектора в регистре LDTR.

Значение сегментного регистра (селектор) содержит индекс дескриптора в дескрипторной таблице; бит, определяющий, к какой дескрипторной таблице производится обращение LDT (TI = 1) или GDT (TI = 0); а также запрашиваемые права доступа к сегменту PRL.



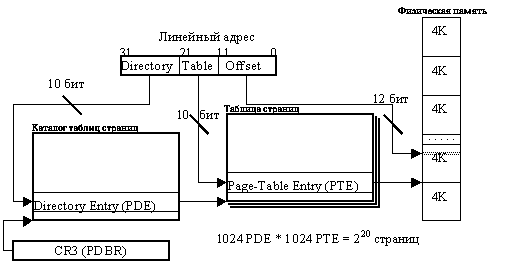
Для вычисления линейного адреса МП выполняет следующие действия:

* 1. МП использует селектор сегмента для нахождения дескриптора сегмента. Селектор содержит индекс дескриптора в дескрипторной таблице (Index), бит TI, определяющий, к какой дескрипторной таблице производится обращение (LDT или GDT), а также запрашиваемые права доступа к сегменту (RPL). Если селектор хранится в сегментном регистре, то обращение к дескрипторным таблицам происходит только при загрузке селектора в сегментный регистр, т. к. каждый сегментный регистр хранит соответствующий дескриптор в программно-недоступном ("теневом") регистре-кэше.
  2. МП анализирует дескриптор сегмента, контролируя права доступа (сегмент доступен с текущего уровня привилегий) и предел сегмента (смещение не превышает предел);
  3. МП добавляет смещение к базовому адресу сегмента и получает линейный адрес.

## Страничная модель памяти

Особенностью этого преобразования является то, что процессор в этом случае оперирует с блоками физической памяти равной длины (4 Кбайт) - *страницами*. Страницы не имеют непосредственного отношения к логической структуре программы. Страничное преобразование действует только в защищенном режиме и включается установкой в 1 бита PG в регистре CR0.

В страничном преобразовании участвуют два типа структур: *каталоги таблиц (Page Directory) и таблицы страниц (Page Table).* Эти структуры состоят из 1024 32-битных элементов. Элементы содержат старшие 20 бит физического адреса адресуемых объектов. Элементы таблицы страниц (Page Table Entry - PTE) адресуют страницы, а элементы каталога таблиц (Page Directory Entry - PDE) адресуют таблицы страниц. Старшие 20 бит физического адреса каталога таблиц хранятся в регистре CR3 (Page Directory Base Register - PDBR) (это единственный регистр процессора, который содержит физический адрес памяти). Все структуры выровнены по границе страницы.



В процессе страничной трансляции адресов полученный линейный адрес разбивается на три части. Старшие десять бит (Directory) линейного адреса являются индексом элемента из каталога таблиц. По этому элементу определяется физический адрес таблицы страниц. Биты 21-12 (Table) линейного адреса выбирают элемент из этой таблицы страниц. Выбранный элемент определяет физический адрес страницы. Младшие 12 бит (Offset) линейного адреса определяют смещение от начала страницы.

Страницы начинаются на границах 4 Кбайт областей памяти, поэтому младшие 12 бит адреса страницы всегда равны нулю. В каталоге таблиц элементы хранят физические адреса таблиц страниц. В таблице страниц элементы хранят физические адреса самих страниц.

# Лекция 5. Типы данных. Форматы команд.

## Форматы и типы данных

В связи с необходимостью решения различных прикладных задач на ЭВМ общего назначения, уже начиная с машин второго поколения отчетливо наметилась тенденция к созданию универсального набора команд. Такой универсальный набор охватывал как разнообразные ***форматы данных***, т.е. количество занимаемых объектом битов, так и различные ***типы данных***, т.е. внутреннюю структуру формата данных.

Рассмотрим наиболее используемые в рамках универсальных ЭВМ форматы и типы данных.



Рис.5.1. Форматы данных

***Форматы данных***

* *байт* — восемь последовательно расположенных битов, пронумерованных от 0 до 7, при этом бит 0 является самым младшим значащим битом;
* *слово* — последовательность из двух байт, имеющих последовательные адреса. Размер слова — 16 бит; биты в слове нумеруются от 0 до 15. Байт, содержащий нулевой бит, называется младшим байтом, а байт, содержащий 15-й бит - старшим байтом. Микропроцессоры Intel имеют важную особенность — младший байт всегда хранится по меньшему адресу. Адресом слова считается адрес его младшего байта. Адрес старшего байта может быть использован для доступа к старшей половине слова.
* *двойное слово* — последовательность из четырех байт (32 бита), расположенных по последовательным адресам. Нумерация этих бит производится от 0 до 31. Слово, содержащее нулевой бит, называется младшим словом, а слово, содержащее 31-й бит, - старшим словом. Младшее слово хранится по меньшему адресу. Адресом двойного слова считается адрес его младшего слова. Адрес старшего слова может быть использован для доступа к старшей половине двойного слова.
* *учетверенное слово* — последовательность из восьми байт (64 бита), расположенных по последовательным адресам. Нумерация бит производится от 0 до 63. Двойное слово, содержащее нулевой бит, называется младшим двойным словом, а двойное слово, содержащее 63-й бит, — старшим двойным словом. Младшее двойное слово хранится по меньшему адресу. Адресом учетверенного слова считается адрес его младшего двойного слова. Адрес старшего двойного слова может быть использован для доступа к старшей половине учетверенного слова.

***Типы данных***

Микропроцессор может обрабатывать числа следующих типов:

* Целый тип без знака;
* Целый тип со знаком;
* Двоично - десятичный тип;
* Типы данных с плавающей точкой.

Рассмотрим сначала целые числа без знака и со знаком.

*Целый тип без знака* - двоичное значение без знака, размером 8, 16 или 32 бита. Числовой диапазон для этого типа следующий:

байт - от 0 до 255;

слово - от 0 до 65 535;

двойное слово - от 0 до 232-1.

Отметим некоторую особенность представления чисел в ПК: числа размером в слово и двойное слово хранятся в памяти в «перевернутом виде». Так у числа размером в слово старшие (левые) 8 битов размещаются во втором байте слова, а младшие (правые) 8 битов – в первом байте. Зачем это сделано? Первые модели ПК были 8-разрядными, в них за раз можно было считать из памяти один байт. Поскольку в этих условиях многозначное число нельзя считать из памяти сразу целиком, то в первую очередь считывался байт с младшими цифрами числа, а для этого надо, чтобы такой байт хранился в памяти первым. По этой причине в первых ПК и появилось «перевернутое» представление чисел. В последующих моделях ради сохранения преемственности, ради того, чтобы ранее составленные программы могли без изменений выполняться на новых ПК, сохранили это «перевернутое» представление чисел.

*Целый тип со знаком* - двоичное представление со знаком размером 8, 16 или 32 бита. Знак в этом двоичном числе содержится в 7, 15 или 31 бите соответственно.

7 6 5 4 3 2 1 0

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

Модуль числа

Знаковый разряд

*Ноль в этих битах в операндах соответствует положительному числу, а единица - отрицательному*. Числовые диапазоны для этого типа данных следующие:

8-разрядное целое - от -128 до +127;

16-разрядное целое - от -32 768 до +32 767;

32-разрядное целое - от -2 31  до +2 31 -1.

В вычислительной технике принято записывать отрицательные числа в так называемом дополнительном коде, который образуется из прямого путем замены всех двоичных нулей единицами и наоборот (обратный код) и прибавления к полученному числу единицы. Это справедливо как для байтовых (8-битовых) чисел, так и для чисел размером в слово или в двойное слово.

доп(x)=

Такой способ образования отрицательных чисел удобен тем, что позволяет выполнять над ними арифметические операции по общим правилам с получением правильного результата. Так, сложение чисел +5 и -5 дает 0; в результате вычитания 3 из 5 получается 2; вычитание -3 из -5 дает -2 и т.д.

Среди команд процессора, выполняющих ту или иную обработку чисел, можно выделить команды, безразличные к знаку числа (например, inc, add, test), команды, предназначенные для обработки чисел без знака (mul, div, ja, jb и др.), а также команды, специально рассчитанные на обработку чисел со знаком (imul, idiv, jg, jl и т.д.). Особенности использования этих команд будут описаны в следующей главе.

*Двоично-десятичные числа* – специальный вид представления числовой информации, в основу которого положен принцип кодирования каждой десятичной цифры числа группой разрядов из 4-х бит. При этом каждый байт числа содержит 1 или 2 десятичные цифры в так называемом *двоично-десятичном коде (BCD – Binary Coded Decimal)*, Микропроцессор может хранить такие числа в 2-х форматах:

* упакованный формат – в байте 2 десятичные цифры, при этом старшая цифра занимает старшие 4 бита, диапазон представления чисел в одном байте составляет 00-99;

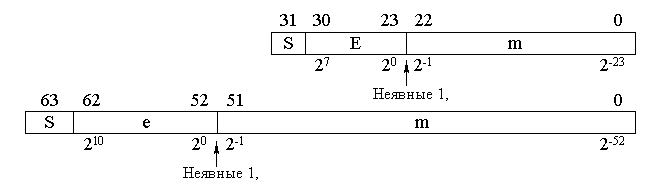
|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Пример. Десятичное число 8754031   |  |  |  |  | | --- | --- | --- | --- | | 0 8 | 7 5 | 4 0 | 3 1 |   представимо в упакованном формате BCD-числа следующим образом:   |  |  |  |  | | --- | --- | --- | --- | | 0000 1000 | 0111 0101 | 0100 0000 | 0011 0001 | |

* неупакованный формат - в байте 1 цифра в 4 младших битах. Старшие биты все имеют нулевое значение и называются ***зоной***.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Пример. Десятичное число 5801   |  |  |  |  | | --- | --- | --- | --- | | 0 5 | 0 8 | 0 0 | 0 1 |   представимо в неупакованном формате BCD-числа следующим образом:   |  |  |  |  | | --- | --- | --- | --- | | 0000 0101 | 0000 1000 | 0000 0000 | 0000 0001 | |

*Представление чисел в двоичном коде с плавающей точкой.*

Существует два общих формата для хранения значений с плавающей точкой, которые были разработаны для процессоров Intel, а затем были стандартизированы организацией IEEE. Существует стандарт IEEE 754 для представления чисел с одинарной точностью (float) и с двойной точностью (double). Для записи числа в формате с плавающей точкой одинарной точности требуется тридцатидвухбитовое слово. Для записи чисел с двойной точностью требуется шестидесятичетырёхбитовое слово. Чаще всего числа хранятся в нескольких соседних ячейках памяти процессора. Форматы числа в формате с плавающей точкой одинарной точности и числа в формате с плавающей точкой удвоенной точности приведены на рисунке



Для одинарной точности 1 бит используется для знака, 8 бит для экспоненты и 23 бита для мантиссы.

Для двойной точности 1 бит используется для знака, 11 бит для экспоненты и 52 бита для мантиссы.

На рисунке буквой S обозначен знак числа, 0 - это положительное число, 1 - отрицательное число.

Е (e) – экспонента. Экспоненты коротких реальных значений хранятся как 8-разрядные целые числа без знака, с уклоном 127.  
Например, число 1.101 x 25. Экспонента (5) добавляется к 127 и сумма (132) в бинарном виде = 10100010. Двоичная экспонента - без знака, поэтому она не может быть отрицательной. Максимальное значение экспоненты - 128, которое при добавлении 127 дает 255 - максимальное значение для 8 бит.  
Примерный диапазон - 1.0 x 2-127 to 1.0 x 2+128.

m-мантисса. В десятичной мантиссе после запятой могут присутствовать цифры 1:9, а в двоичной - только 1. Поэтому для хранения единицы после двоичной запятой не выделяется отдельный бит в числе с плавающей точкой. Единица подразумевается, как и двоичная запятая. Кроме того, в формате чисел с плавающей точкой принято, что мантисса всегда больше 1. То есть диапазон значений мантиссы лежит в диапазоне от 1 до 2.

Рассмотрим несколько примеров:

1) Определить число с плавающей точкой, лежащее в четырёх соседних байтах: 11000001 01001000 00000000 00000000

- Знаковый бит, равный 1 показывает, что число отрицательное.

- Экспонента 10000010 в десятичном виде соответствует числу 130. Вычтя число 127 из 130, получим число 3.

- Теперь запишем мантиссу: 1,100 1000 0000 0000 0000 0000

- И, наконец, определим десятичное число: 1100,1b = 12,5d

2) Определить число с плавающей точкой, лежащее в четырёх соседних байтах:

11000011 00110100 00000000 00000000

- Знаковый бит, равный 1 показывает, что число отрицательное.

- Экспонента 10000110 в десятичном виде соответствует числу 134. Вычтя число 127 из 134, получим число 7.

- Теперь запишем мантиссу: 1,011 0100 0000 0000 0000 0000

- И, наконец, определим десятичное число: 10110100b=180d

## Формат машинной команды

Машинная команда представляет собой закодированное по определенным правилам указание микропроцессору на выполнение некоторой операции или действия. Команды в архитектуре х86 имеют большое разнообразие форматов, которые зависят от типа операции, режимов адресации операндов, длины используемых непосредственных операндов и смещений и ряда других факторов. Они имеют длину от 1 до 15 байт.

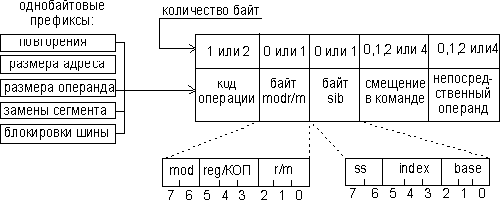


Рис.5.2. Формат машинной команды

Опишем назначения полей машинной команды [Юров,2002].

***Префиксы.*** Необязательные элементы машинной команды, каждый из которых состоит из одного байта или может отсутствовать. В памяти префиксы предшествуют команде. Назначение префиксов — модифицировать операцию, выполняемую командой. Прикладная программа может использовать следующие типы префиксов:

*Префикс замены сегмента*. В явной форме указывает, какой сегментный регистр используется в данной команде для адресации стека или данных. Префикс отменяет выбор сегментного регистра по умолчанию.

*Префикс разрядности адреса* уточняет разрядность адреса (32 или 16-разрядный). Каждой команде, в которой используется адресный операнд, ставится в соответствие разрядность адреса этого операнда. С помощью префикса разрядности адреса можно изменить действующее по умолчанию значение разрядности адреса. Это изменение будет касаться только той команды, которой предшествует префикс.

*Префикс разрядности операнда* аналогичен префиксу разрядности адреса, но указывает на разрядность операндов (32 или 16-разрядные), с которыми работает команда.

*Префикс повторения* используется с цепочечными командами (командами обработки строк). Этот префикс “зацикливает” команду для обработки всех элементов цепочки.

***Код операции***. Обязательный элемент, описывающий операцию, выполняемую командой. Многим командам соответствует несколько кодов операций, каждый из которых определяет нюансы выполнения операции.

***Байт режима адресации modr/m.*** Значения этого байта определяет используемую форму адреса операндов. Операнды могут находиться в памяти в одном или двух регистрах. Если операнд находится в памяти, то байт modr/m определяет компоненты (смещение, базовый и индексный регистры), используемые для вычисления его эффективного адреса. В защищенном режиме для определения местоположения операнда в памяти может дополнительно использоваться байт sib (Scale-Index-Base — масштаб-индекс-база). Байт modr/m состоит из трех полей (см. рис. 1):

*поле mod* определяет количество байт, занимаемых в команде адресом операнда (mod = 00 - поле смещение в команде отсутствует и адрес операнда определяется содержимым базового и (или) индексного регистра, если mod = 01, это означает, что поле смещение в команде присутствует, занимает один байт и модифицируется содержимым базового и (или) индексного регистра, если mod = 10, это означает, что поле смещение в команде присутствует, занимает два или четыре байта (в зависимости от действующего по умолчанию или определяемого префиксом размера адреса) и модифицируется содержимым базового и (или) индексного регистра, если mod = 11, это означает, что операндов в памяти нет: они находятся в регистрах);

*поле reg/коп* определяет либо регистр, находящийся в команде на месте первого операнда, либо возможное расширение кода операции;

*поле r/m* используется совместно с полем mod и определяет либо регистр, находящийся в команде на месте первого операнда (если mod = 11), либо используемые для вычисления эффективного адреса (совместно с полем смещение в команде) базовые и индексные регистры.

***Байт масштаб-индекс-база*** (байт sib) используется для расширения возможностей адресации операндов. Байт sib состоит из трех полей:

*поля масштаба ss*. В этом поле размещается масштабный множитель для индексного компонента index, занимающего следующие три бита байта sib. В поле ss может содержаться одно из следующих значений: 1, 2, 4, 8.

При вычислении эффективного адреса на это значение будет умножаться содержимое индексного регистра.

*поля index* — используется для хранения номера индексного регистра, который применяется для вычисления эффективного адреса операнда;

*поля base* — используется для хранения номера базового регистра, который также применяется для вычисления эффективного адреса операнда. Напомню, что в качестве базового и индексного регистров могут использоваться практически все регистры общего назначения.

***Поле смещения в команде***. 8, 16 или 32-разрядное целое число со знаком, представляющее собой, полностью или частично (с учетом вышеприведенных рассуждений), значение эффективного адреса операнда.

***Поле непосредственного операнда***. Необязательное поле, представляющее собой 8, 16 или 32-разрядный непосредственный операнд.

# Лекция 6. Архитектура МП 8086.

## 

## Программная модель микропроцессора.

***Программной моделью*** микропроцессора (по [Юров,2002,с.37]) называется набор ресурсов микропроцессора, получаемых в своё распоряжение любой выполняющейся программой и необходимых для выполнения и хранения в памяти команд программы, данных и информации о текущем состоянии программы и микропроцессора.

Программная модель МП представляется набором его регистров. *Регистр* является устройством временного хранения данных и используется с целью облегчения арифметических, логических и пересылочных операций.

Программную модель микропроцессора Intel  80**86** составляют:

-  пространство адресуемой памяти;

-   набор регистров для хранения данных общего назначения;

-   набор сегментных регистров;

-   набор регистров состояния и управления;

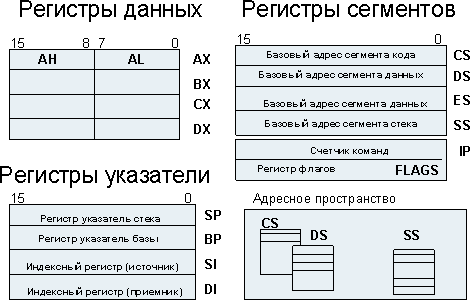
****

Рис.6.1.Программная модель МП 8086

**Регистры микропроцессора**

Регистры микропроцессора  представляют собой быстродействующую память микропроцессора размером, как правило, в одно или два машинных слова, предназначенную для хранения основных или промежуточных данных (либо их частей) и используемую при выполнении машинных команд. Достоинство регистров заключается в их высоком быстродействии, гораздо большем, чем у оперативной памяти, а недостаток в том, что их очень мало - всего около десятка. Поэтому регистры используются лишь для кратковременного хранения данных.

Программная модель включает: четыре 16-битных регистра общего назначения AX, BX, CX, DX, четыре регистра-указателя SI, DI, BP и SP, четыре регистра сегментов CS, DS, ES, SS, один 16-битовый регистр флагов FLAGS и указатель программ IP.

В процессоре i8080 были байтовые регистры A, B, C, D, но затем их расширили до размеров слова, поэтому в их названии добавилась буква X (от англ.  eXtended - " расширенный ").

По назначению и способу использования регистры можно разбить на следующие группы (имена регистров можно записывать строчными буквами):

- AX, BX, CX, DX, SI, DI, BP, SP (регистры общего назначения);

- CS, DS, SS, ES (сегментные регистры);

- IP (указатель команд);

- Flags (регистр флагов).

***Регистры общего назначения***.

Эти регистры физически находятся в микропроцессоре внутри арифметико-логического устройства, поэтому их ещё называют регистрами арифметико-логического устройства.

Регистры общего назначения используются для хранения:

* операндов логических и арифметических операций;
* компонентов адреса;
* указателей на ячейки памяти.

Первые четыре регистра AX, BX, CX и DX, в отличии от остальных, предоставляют возможность независимого доступа к их старшей и младшей половинам, имена которых содержат соответственно букву H (от англ.  higth  - "старший ") или букву L (от англ.  low  - " младший ").

Приведём назначения указанных выше регистров [Юров,2002,с.40]:

* ax/ah|al ( accumulator ) -  аккумулятор  применяется для хранения промежуточных данных (в некоторых командах использование этого регистра обязательно);
* bx/bh|bl ( base ) -  базовый регистр  применяется для хранения базового адреса некоторого объекта в памяти;
* cx/ch|cl ( counter ) -  регистр-счётчик  применяется в командах, производящих некоторые повторяющиеся действия (его использование зачастую неявно и скрыто в алгоритме работы соответствующей команды);
* dx/dh|dl ( data ) -  регистр данных  используется как вторичный аккумулятор для хранения промежуточных данных и результатов.

Следующие два регистра используются для поддержки  цепочечных  операций, т.е. операций, производящих последовательную обработку цепочек элементов:

* si (sourse  index) - индексный регистр цепочки относительно  сегмента, адресуемого  ds. Этот регистр в цепочечных операциях содержит текущий адрес элемента в цепочке-источнике;
* di (destination  index) - индексный регистр цепочки относительно сегмента, адресуемого  es. Этот регистр в цепочечных операциях содержит текущий адрес элемента в цепочке-приёмнике.

В архитектуре микропроцессора на программно-аппаратном уровне поддерживается такая структура данных как стек. Для работы со стеком в системе команд микропроцессора существуют специальные команды, а в программной модели микропроцессора для этого имеются специальные регистры:

* sp ( stack  pointer ) - регистр указателя стека. Содержит указатель вершины стека в текущем сегменте стека;
* bp ( base  pointer) - регистр указателя базы кадра стека. Предназначен для организации произвольного доступа к данным внутри стека.

Некоторые команды используют фиксированные регистры для выполнения своих действий. Знание жёсткого функционального назначения регистров общего назначения позволит Вам при необходимости хотя бы на несколько байтов сэкономить память, занимаемую кодом. Однако, при программировании большинство регистров можно использовать для простого хранения операндов команд.

***Регистры состояния и управления.***

В микропроцессор включены несколько регистров, которые постоянно содержат информацию о состоянии, как самого микропроцессора, так и программы, команды которой в данный момент загружены на конвейер.

К этим регистрам относятся:

- ip ( instruction  pointer) - регистр указателя команды  позволяет следить за ходом выполнения программы, указывая в каждый момент времени относительный адрес команды, следующей за исполняемой. При этом абсолютный адрес команды определяется парой регистров CS:IP (адрес сегмента кода и смещение относительно начала указанного сегмента кода). Регистр IP непосредственно недоступен программисту, но загрузка и изменение его значения производятся различными командами управления, к которым относятся команды условных и безусловных переходов, вызова процедур и возврата из процедур, возникновение прерываний;

- flags ( flag register ) -  регистр флагов, разрядность которого составляет 16 бита. Отдельные биты данного регистра имеют определённое функциональное назначение и называются флагами. *Флаг – это бит, принимающий значение 1 (флаг установлен), если выполнено некоторое условие, и значение 0 (флаг сброшен) в противном случае.*

В программе нередко требуется принять решение на основании результата только что исполненной микропроцессором команды. В регистре флагов как раз и фиксируется информация, которая может помочь программе "принять решение".

Биты (или флаги) регистра признаков FLAGS разделяются на условные, отражающие результат предыдущей операции ALU, и управляющие, от которых зависит выполнение специальных функций.



Рис. 4.3 Содержимое регистра flags

*Флаги состояния*. Эти флаги могут изменяться после выполнения машинных команд. Флаги отражают следующие условия:

* флаг переноса *CF* устанавливается в 1 при переносе/заеме, возникающем при сложении/вычитании байтов или слов, а также принимает значение выдвигаемого бита при сдвигах операнда;
* флаг четности *PF* устанавливается в 1, если младшие 8 бит результата имеют четное число единиц;
* флаг вспомогательного переноса *AF* устанавливается в 1, если есть перенос/заем из младшей тетрады в старшую 8- или 16-битного результата в операциях десятичного сложения/вычитания;
* флаг нуля *ZF* устанавливается в 1 при нулевом результате операции;
* флаг знака *SF* устанавливается в 1 при отрицательном результате операции при использовании дополнительного кода;
* флаг переполнения *OF* устанавливается в 1 при потере старшего бита результата сложения или вычитания;

*Флаг управления DF*. Если флаг направления *DF* установлен в 1, то используется автодекрементная адресация при выполнении операции обработки строк, если *DF* установлен в 0, то используется автоинкрементная адресация;

*Системные флаги*:

* *IF* - флаг прерывания (если установлен в 1, то внешние маскируемые прерывания разрешены, иначе запрещены);
* *TF* - флаг трассировки (если установлен в 1, то процессор переходит в состояние прерывания после выполнения каждой операции, что позволяет проводить пошаговую отладку программ);

***Сегментные регистры.***

Диапазон адресов, формируемых процессором, называют **адресным пространством процессора**. Адресное пространство памяти процессора, откуда происходит выборка команд и данных, разделено на сегменты (области памяти) емкостью до 64 Кбайт каждый. Процессор 8086 имеет прямой доступ одновременно к четырем сегментам: cs, ss, ds, es.

Начальный адрес сегмента может быть установлен прикладной программой и всегда должен начинаться с 16-байтовых границ. Базовый адрес сегмента получается делением действительного физического адреса начальной ячейки сегмента на 16. Базовые адреса содержатся в одном из четырех 16-битных сегментных регистров CS, DS, ES и SS. На расположение сегмента не накладывается никаких специальных ограничений, кроме одного: он должен быть на границе 16 байт (т.е. физический адрес начальной ячейки должен делиться на 16). Сегменты могут быть смежными, разделенными, перекрываться частично или полностью.

Регистры сегментов используются для идентификации текущего сегмента адресного пространства.

* cs ( code  segment) -  сегментный регистр кода. Он содержит адрес сегмента с машинными командами, к которому имеет доступ микропроцессор;
* ds ( data  segment) -  сегментный регистр данных. Он хранит адрес сегмента данных текущей программы;
* ss ( stack  segment) -  сегментный регистр стека. Он содержит адрес сегмента стека, в котором последний записанный элемент выбирается первым;
* es ( extra  segments) -регистр дополнительного сегмента данных. Он содержит адрес дополнительного сегмента данных текущей программы.

Ни в каких арифметических, логических и т.п. операциях эти регистры не участвуют, можно только записывать в них и считывать из них.

# Лекция 7. Архитектура 32-разрядного универсального микропроцессора IA32

Программная модель микропроцессора Intel  IA32включает восемь регистров общего назначения, шесть регистров сегментов, указатель команд, регистр системных флагов, регистры системных адресов, четыре регистра управления и шесть регистров отладки.

**Набор регистров общего назначения** включает в себя регистры 16- разрядных процессоров 8086/8088, однако они имеют разрядность 32 бита.

К обозначениям регистров добавлена приставка E (Extended - расширенный): EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP. Возможно обращение к младшим 16 разрядам расширенных регистров по именам без приставки, например, AX, BX, а также отдельно к младшим и старшим байтам, например, AL, AH.

**Сегментные регистры** CS, SS, DS, ES, как и в процессорах 8086/8088, имеют разрядность 16 бит. К ним добавлены регистры FS и GS – дополнительные сегментные регистры данных.

**Регистр FLAGS** расширен до 32 разрядов и обозначается – EFLAGS. Флаги 16-разрядных процессоров с 0 по 11 разряд занимают такое же положение в EFLAGS. Добавлены новые флаги:

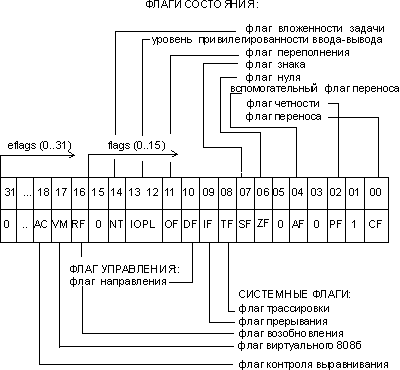


Рис. 7.1. Содержимое регистра eflags

• Nested Task (NT, 14 бит) – флаг вложенной задачи. Устанавливается, когда текущая задача связана с прерванной задачей, очищается, если такой связи нет.

• I/O Privilege Level (IOPL, 12-13 биты) – определяет уровень привилегий ввода/вывода для текущей задачи.

• Virtual-8086 Mode (VM, 17 бит) – включает/выключает режим V86 (виртуального микропроцессора i8086) в защищенном режиме.

* RF - Resume Flag (16 бит) – флаг возобновления исполнения при отладке.
* AC - Alignment Check (18 бит) – флаг контроля выравнивания. При установке этого флага во время обращении к невыровненному операнду возникает исключение.

***Регистры управления***

В защищенном режиме используются также 32-разрядные *регистры управления CR0-CR3.* Эти регистры предназначены для общего управления системой. Регистры управления доступны только программам с уровнем привилегий 0. Хотя микропроцессор имеет четыре регистра управления, доступными являются только три из них — исключается cr1, функции которого пока не определены (он зарезервирован для будущего использования).

Регистр **cr0** содержит системные флаги, управляющие режимами работы микропроцессора и отражающие его состояние глобально, независимо от конкретных выполняющихся задач.

Назначение системных флагов:

* pe (Protect Enable), бит 0 — разрешение защищенного режима работы.

Состояние этого флага показывает, в каком из двух режимов — реальном (pe=0) или защищенном (pe=1) — работает микропроцессор в данный момент времени.

* mp (Math Present), бит 1 — наличие сопроцессора. Всегда 1.
* ts (Task Switched), бит 3 — переключение задач. Процессор автоматически устанавливает этот бит при переключении на выполнение другой задачи.
* am (Aligment Mask), бит 18 — маска выравнивания. Этот бит разрешает (am = 1) или запрещает (am = 0) контроль выравнивания.
* cd (Cache Disable), бит 30, — запрещение кэш-памяти. С помощью этого бита можно запретить (cd = 1) или разрешить (cd = 0) использование внутренней кэш-памяти (кэш-памяти первого уровня).
* pg (PaGing), бит 31, — разрешение (pg = 1) или запрещение (pg = 0) страничного преобразования. Флаг используется при страничной модели организации памяти.

Регистр **cr2** используется при страничной организации оперативной памяти для регистрации ситуации, когда текущая команда обратилась по адресу, содержащемуся в странице памяти, отсутствующей в данный момент времени в памяти. В такой ситуации в микропроцессоре возникает исключительная ситуация с номером 14, и линейный 32-битный адрес команды, вызвавшей это исключение, записывается в регистр cr2. Имея эту информацию, обработчик исключения 14 определяет нужную страницу, осуществляет ее подкачку в память и возобновляет нормальную работу программы;

Регистр **cr3** также используется при страничной организации памяти.

Это так называемый регистр каталога страниц первого уровня. Он содержит 20-битный физический базовый адрес каталога страниц текущей задачи. Этот каталог содержит 1024 32-битных дескриптора, каждый из которых содержит адрес таблицы страниц второго уровня. В свою очередь каждая из таблиц страниц второго уровня содержит 1024 32-битных дескриптора, адресующих страничные кадры в памяти. Размер страничного кадра — 4 Кбайт.

***Регистры отладки***

Это очень интересная группа регистров, предназначенных для аппаратной отладки. Средства аппаратной отладки впервые появились в микропроцессоре i486. Аппаратно микропроцессор содержит восемь регистров отладки, но реально из них используются только 6.

Регистры dr0, dr1, dr2, dr3 имеют разрядность 32 бит и предназначены для задания линейных адресов четырех точек прерывания. Используемый при этом механизм следующий: любой формируемый текущей программой адрес сравнивается с адресами в регистрах dr0...dr3, и при совпадении генерируется исключение отладки с номером 1.

Регистр dr6 называется регистром состояния отладки. Биты этого регистра устанавливаются в соответствии с причинами, которые вызвали возникновение последнего исключения с номером 1.

Перечислим эти биты и их назначение:

* b0 — если этот бит установлен в 1, то последнее исключение (прерывание) возникло в результате достижения контрольной точки, определенной в регистре dr0;
* b1 — аналогично b0, но для контрольной точки в регистре dr1;
* b2 — аналогично b0, но для контрольной точки в регистре dr2;
* b3 — аналогично b0, но для контрольной точки в регистре dr3;
* bd (бит 13) — служит для защиты регистров отладки;
* bs (бит 14) — устанавливается в 1, если исключение 1 было вызвано состоянием флага tf = 1 в регистре eflags;
* bt (бит 15) устанавливается в 1, если исключение 1 было вызвано переключением на задачу с установленным битом ловушки в TSS t = 1.

Все остальные биты в этом регистре заполняются нулями. Обработчик исключения 1 по содержимому dr6 должен определить причину, по которой произошло исключение, и выполнить необходимые действия.

Регистр dr7 называется регистром управления отладкой. В нем для каждого из четырех регистров контрольных точек отладки имеются поля, с помощью которых можно уточнить следующие условия, при которых следует сгенерировать прерывание:

* место регистрации контрольной точки — только в текущей задаче или в любой задаче. Эти биты занимают младшие восемь бит регистра dr7 (по два бита на каждую контрольную точку (фактически точку прерывания), задаваемую регистрами dr0, dr1, dr2, dr3 соответственно). Первый бит из каждой пары — это так называемое локальное разрешение; его установка говорит о том, что точка прерывания действует если она находится в пределах адресного пространства текущей задачи. Второй бит в каждой паре определяет глобальное разрешение, которое говорит о том, что данная контрольная точка действует в пределах адресных пространств всех задач, находящихся в системе;
* тип доступа, по которому инициируется прерывание: только при выборке команды, при записи или при записи/чтении данных. Биты, определяющие

***Регистры системных адресов***

Эти регистры еще называют регистрами управления памятью.

Они предназначены для защиты программ и данных в мультизадачном режиме работы микропроцессора.

При работе в защищенном режиме микропроцессора адресное пространство делится на:

глобальное — общее для всех задач;

локальное — отдельное для каждой задачи.

Этим разделением и объясняется присутствие в архитектуре микропроцессора следующих системных регистров:

* регистра таблицы глобальных дескрипторов **gdtr** (Global Descriptor Table Register) имеющего размер 48 бит и содержащего 32-битовый (биты 16—47) базовый адрес глобальной дескрипторной таблицы GDT и 16-битовое (биты 0—15) значение предела, представляющее собой размер в байтах таблицы GDT;
* регистра таблицы локальных дескрипторов **ldtr** (Local Descriptor Table Register) имеющего размер 16 бит и содержащего так называемый селектор дескриптора локальной дескрипторной таблицы LDT. Этот селектор является указателем в таблице GDT, который и описывает сегмент, содержащий локальную дескрипторную таблицу LDT;
* регистра таблицы дескрипторов прерываний **idtr** (Interrupt Descriptor Table Register) имеющего размер 48 бит и содержащего 32-битовый (биты 16–47) базовый адрес дескрипторной таблицы прерываний IDT и 16-битовое (биты 0—15) значение предела, представляющее собой размер в байтах таблицы IDT;
* 16-битового регистра задачи **tr** (Task Register), который подобно регистру ldtr, содержит селектор, то есть указатель на дескриптор в таблице GDT. Этот дескриптор описывает текущий сегмент состояния задачи (TSS — Task Segment Status). Этот сегмент создается для каждой задачи в системе, имеет жестко регламентированную структуру и содержит контекст (текущее состояние) задачи. Основное назначение сегментов TSS — сохранять текущее состояние задачи в момент переключения на другую задачу.

# Лекция 8. Методы адресации

Большая часть команд процессора работает с кодами данных (операндами). Одни команды требуют входных операндов (одного или двух), другие выдают выходные операнды (чаще один операнд). Входные операнды называются еще операндами-источниками, а выходные называются операндами-приемниками. Все эти коды операндов (входные и выходные) должны где-то располагаться. Они могут находиться во внутренних регистрах процессора (наиболее удобный и быстрый вариант). Они могут располагаться в системной памяти (самый распространенный вариант). Наконец, они могут находиться в устройствах ввода/вывода (наиболее редкий случай). Определение места положения операндов производится кодом команды. Причем существуют разные методы, с помощью которых код команды может определить, откуда брать входной операнд и куда помещать выходной операнд. Эти методы называются *методами адресации.* Эффективность выбранных методов адресации во многом определяет эффективность работы всего процессора в целом.

Далее мы подробно расскажем о способах адресации микропроцессора, а пока определим понятие "адресация".

***Адресацией*** (по [Таненбаум,2002,с.353]) называется процесс определения местоположения операндов команды микропроцессора (т.е. их адреса).

В зависимости от используемого метода адресации получение исполнительного адреса может заключаться только в извлечении его как составной части исполняемой команды, а могут потребоваться дополнительные операции сложения составной части команды с содержимым других регистров.

***Методы адресации***

Количество методов адресации в различных процессорах может быть от 4 до 16. Рассмотрим несколько типичных методов адресации операндов, используемых сейчас в большинстве микропроцессоров.

* регистровая;
* непосредственная, которая заключается в указании в команде самого значения операнда, а не его адреса;
* прямая, предполагающая указание в команде непосредственно исполнительного адреса;
* косвенная, при которой в команде указывается адрес регистра или ячейки памяти, в которых хранится адрес операнда или его составляющие;

Из перечисленных режимов адресации самыми быстрыми являются  регистровая адресация и  непосредственная адресация, поскольку в этом случае операционный блок микропроцессора извлекает их либо из регистров, либо из самой команды. В других режимах адресация выполняется дольше, т.к. интерфейс шины вначале должен вычислить адрес ячейки памяти, извлечь операнд, а лишь после этого передать его операционному блоку.

Когда операционному блоку требуется прочитать или записать значение операнда, находящегося в памяти, он передает значение смещения адреса интерфейсу шины. Последний добавляет это смещение к содержимому регистра сегмента и тем самым получает 20-битовый физический адрес, который и используется для доступа к операнду.

*Регистровая адресация.*

Операнд (байт или слово) находится в регистре. Этот способ адресации применим ко всем программно-адресуемым регистрам процессора.

inc СН; Плюс 1 к содержимому СН

push DS; DS сохраняется в стеке

xchg ВХ,ВР; ВХ и ВР обмениваются содержимым

mov ES, АХ; Содержимое АХ пересылается в ES

*Непосредственная**адресация* имеет место, если операнд-источник является константой или переменной, которой присвоено постоянное значение.

Например:

MOV AX, 5 ;загружает значение 5 в РОН AX

MOV AL,'\*'; Код ASCII символа "\*' загружается в AL

. . . . .

K EQU 1024

.. . . . . . . .

MOV CX, K ;загружает в РОН СХ константу 1024, определенную идентификатором К.

Команда MOV, использованная в последнем предложении, имеет два операнда; первый операнд определяется с помощью регистровой адресации, второй - с помощью непосредственной.

Следует отметить, что непосредственный операнд может быть задан простым выражением, в котором константы или идентификаторы констант связаны арифметическими операциями +, -, \* или / (в таких выражениях не должно быть скобок). Например:

MOV AX, 156\*10Н/2.

Следует помнить, что диапазон посылаемых чисел (значений непосредственного операнда) определяется вместимостью приемника - если это однобайтовый регистр (AH, AL, BL …), то в него можно посылать беззнаковые числа в диапазоне от 0 до 255, знаковые – от –128 до 127.

Важным применением непосредственной адресации является пересылка относительных адресов (смещений). Чтобы указать, что речь идет об относительном адресе данной ячейки, а не об ее содержимом, используется описатель offset (смещение):

; Сегмент данных

msg db "Режимы адресации' ; Строка символов

;Сегмент команд

mov DX,offset msg ; Адрес строки засылается в DX

В приведенном примере относительный адрес строки msg, т.е. расстояние в байтах первого байта этой строки от начала сегмента, в котором она находится, заносится в регистр DX.

*Прямая адресация памяти.*

Это простейший вид адресации в памяти, т.к. исполнительный адрес содержится в самой команде и для его формирования не используется никаких дополнительных источников или регистров. При прямой адресации в команде указывается то смещение, которое соответствует началу размещения в памяти соответствующего операнда.

По умолчанию считается, что все операнды находятся в сегменте данных, поэтому при вычислении адреса используется текущее содержимое регистра DS.

;Сегмент данных

meml dw 0 ;Слово памяти содержит 0

mem2 db 230 ;Байт памяти содержит 230

;Сегмент команд

…….

inc meml ;Содержимое слова meml увеличивается на 1

mov DX, meml ; Содержимое слова с именем mem1 загружается в DX

mov AL,mem2 ; Содержимое байта с именем mem2 загружается в АL

Сравнивая этот пример с предыдущим, мы видим, что указание в команде имени ячейки памяти обозначает, что операндом является содержимое этой ячейки; указание имени ячейки с описателем offset - что операндом является адрес ячейки.

Прямая адресация памяти на первой взгляд кажется простой и наглядной. Если мы хотим обратиться, например, к ячейке meml, мы просто указываем ее имя в программе. В действительности, однако, дело обстоит сложнее. Вспомним, что адрес любой ячейки состоит из двух компонентов: сегментного адреса и смещения. Обозначения meml и mem2 в предыдущем примере, очевидно, являются смещениями. Сегментные же адреса хранятся в сегментных регистрах. Однако сегментных регистров четыре: DS, ES, CS и SS. Каким образом процессор узнает, из какого регистра взять сегментный адрес, и как сообщить ему об этом в программе?

Как уже говорилось ранее, процессор различает группу кодов, носящих название *префиксов.* Имеется несколько групп префиксов: *повторения, размера адреса, размера операнда, замены сегмента.* Здесь нас будут интересовать префиксы замены сегмента.

Команды процессора, обращающиеся к памяти, могут в качестве первого байта своего кода содержать префикс замены сегмента, с помощью которого процессор определяет, из какого сегментного регистра взять сегментный адрес. Для сегментного регистра ES код префикса составляет 26h, для SS – 36h, для CS - 2Eh. Если префикс отсутствует, сегментный адрес берется из регистра DS (хотя для него тоже предусмотрен свой префикс).

Если в начале программы с помощью директивы assume указано соответствие сегменту данных сегментного регистра DS: assume DS:data , то команды обращения к памяти транслируются без какого-либо префикса, а процессор при выполнении этих команд берет сегментный адрес из регистра DS.

Если в директиве assume указано соответствие сегмента данных регистру ES: assume ES:data (в этом случае сегмент данных должен располагаться перед сегментом команд), то команды обращения к полям этого сегмента транслируются с добавлением префикса замены для сегмента ES. При этом предложения программы выглядят обычным образом; в них по-прежнему просто указываются имена полей данных, к которым производится обращение.

Однако в ряде случаев префикс замены сегмента должен указываться в программе в явной форме. Такая ситуация возникает, например, если данные расположены в сегменте команд, что типично для резидентных обработчиков прерываний. Для обращения к таким данным можно, конечно, использовать регистр DS, если предварительно настроить его на сегмент команд, но проще выполнить адресацию через регистр CS, который и так уже настроен должным образом. Если в сегменте команд содержится поле данных с именем mem, то команда чтения из этого поля будет выглядеть следующим образом:

mov AX,CS:mem

В этом случае транслятор включит в код команды префикс замены для сегмента CS. Другие примеры команд с заменой сегмента будут приведены ниже.

Мы рассмотрели три важнейших способа адресации: регистровую, непосредственную и прямое обращение к памяти. Все остальные режимы адресации относятся к группе косвенной адресации памяти, когда в определении адреса ячейки памяти участвует один или несколько регистров процессора. Рассмотрим последовательно эти режимы.

*Регистровая косвенная (базовая и индексная).*

Адресуется память (байт или слово). В МП 86 косвенная адресация допустима только через регистры ВХ, ВР, SI и DI. При использовании регистров ВХ или ВР адресацию называют базовой, при использовании регистров SI или DI - индексной.

Косвенные регистровые операнды необходимо заключать в квадратные скобки, чтобы отличать их от регистровых операндов. Так как содержимое регистра легко изменить в ходе работы программы, данный способ адресации позволяет динамически назначить адрес операнда для некоторой машинной команды. Это свойство очень полезно, например, для организации циклических вычислений и для работы с различными структурами данных типа таблиц или массивов.

Например, команда mov ax,[bx] загружает в регистр AX содержимое ячейки памяти по адресу из сегмента данных со смещением, хранящимся в регистре BX.

 Пример (косвенной базовой адресации).

mov [bx],al ; Пересылка байта из регистра AL по адресу,

; определяемому регистрами DS:BX

mov dx,[bp] ; Пересылка слова из ячейки памяти, определяемой

; регистрами SS:BP в регистр DX

Пример (косвенной индексной адресации).

  mov dx,[si] ; Пересылка слова из ячейки памяти, определяемой

; регистрами DS:SI в регистр DX

mov [si],cx ; Пересылка слова из регистра CX по адресу,

; содержащемуся в регистре DS:SI

*Регистровая косвенная адресация со смещением (базовая и индексная).*

Адресуется память (байт или слово). Относительный адрес операнда определяется, как сумма содержимого регистра BX, BP, SI или DI и указанной в команде константы, иногда называемой смещением. Смещение может быть числом или адресом. Так же, как и в случае базовой адресации, при использовании регистров BX, SI и DI подразумевается сегмент, адресуемый через DS, а при использовании ВР подразумевается сегмент стека и, соответственно, регистр SS.

Регистр BX удобно использовать при доступе к структурированным записям данных, расположенным в разных областях памяти. При этом базовый адрес помещается в базовый регистр BX и доступ к её отдельным элементам осуществляется по их сдвигу относительно базы.

Пример. Рассмотрим массив MASS, определенный в сегменте данных как

MASS DB 1,2,3,4,5,6,

Так, командами

MOV BX, offset MASS

M1: MOV AX, [BX]

INC BX

INC BX

возвращаясь в цикле на метку М1 можно организовать последовательную пересылку в регистр АХ всех элементов массива MASS.

 Пример (косвенной базовой адресации со смещением).

  mov [bp\*4+2],si ;Эта команда пересылает содержимое регистраSI

; (слово) в переменную, адрес которой определя-

; ется как (BP)\*4+2, причём эта переменная счи-

; тается расположенной в сегменте стека SS

Пример применения косвенной адресации со смещением на примере прямого вывода в видеобуфер.

mov AX,0B800h ;Сегментный адрес

mov ES,AX ;видеобуфера в ES

mov DI, 80\*2\*24 ;Смещение к нижней строке экрана

mov byte ptr ES: [DI] ,'О' ;Символ на экран

mov byte ptr ES:2[DI],'К' ;Запишем символ в следующую позицию

mov byte ptr ES:4[DI],' ! ' ;Запишем символ в следующую позицию

В этом примере в качестве базового выбран регистр DI; в него заносится базовый относительный адрес памяти, в данном случае смещение в видеобуфере к началу последней строки экрана. Модификация этого адреса с целью получить смещение по строке экрана осуществляется с помощью констант 2 и 4, которые при вычислении процессором исполнительного адреса прибавляются к содержимому базового регистра DI.

*Базово-индексная адресация.*

Адресуется память (байт или слово). Относительный адрес операнда определяется, как сумма содержимого следующих пар регистров:

[ВХ] [SI] (подразумевается DS:[BX][SI])

[ВХ][DI] (подразумевается DS:[BX][DI])

[ВР] [SI] (подразумевается SS:[BP][SI])

[ВР] [DI] (подразумевается SS:[BP][DI])

Это чрезвычайно распространенный способ адресации, особенно, при работе с массивами. В нем используются два регистра, при этом одним из них должен быть базовый (ВХ или ВР), а другим - индексный (SI или DI). Как правило, в одном из регистров находится адрес массива, а в другом - индекс в нем, при этом совершенно безразлично, в каком что.

Пример (адресации по базе с индексированием).

  mov ax,[bx][di] ; Адрес второго операнда: (BX)+(DI)

mov ax,[bx+di]

*Базово-индексная адресация со смещением.*

Адресуется память (байт или слово). Относительный адрес операнда определяется как сумма содержимого двух регистров и смещения

*<Базовый регистр>+<Индексный регистр>+<Константа>*

Это способ адресации является развитием предыдущего. В нем используются те же пары регистров, но полученный с их помощью результирующий адрес можно еще сместить на значение указанной в команде константы. Как и в случае базово-индексной адресации, константа может представлять собой индекс (и тогда в одном из регистров должен содержаться базовый адрес памяти), но может быть и базовым адресом. В последнем случае регистры могут использоваться для хранения составляющих индекса.

Пример (косвенной базовой индексной адресации со смещением).

  mov ax,[si+5h][dx]

mov ax,[bx+di+5]

Приведём таблицу, в которой указаны  форматы операндов команд микропроцессора для перечисленных режимов адресации, и для каждого формата указано, какой из регистров сегмента используется для вычисления физического адреса:

|  |  |  |
| --- | --- | --- |
| Режим адресации | Формат операнда | Регистр сегмента |
| Регистровый | Регистр | Не используется |
| Непосредственный | Данное | Не используется |
| Прямой | Сдвиг  Метка | DS  DS |
| Косвенный базовый  (регистровый) | [BX]  [BP] | DS  SS |
| Косвенный базовый  (регистровый)  со смещением | [BX+ сдвиг]  [BP+ сдвиг]  Метка [BX]  Метка [BP]  [ Метка +BX]  [ Метка +BP] | DS  CS  DS  CS  DS  CS |
| Косвенный индексный  (регистровый) | [DI]  [SI] | DS  DS |
| Косвенный индексный  (регистровый)  со смещением | [DI+ сдвиг]  [SI+ сдвиг]  Метка [DI]  Метка [SI]  [ Метка +DI]  [ Метка +SI] | DS  DS  DS  DS  DS  DS |
| Косвенный базовый  индексный | [BX+SI]  [BX+DI]  [BP+SI]  [BP+DI] | DS  DS  SS  SS |
| Косвенный базовый  индексный со смещением | [BX+SI+ сдвиг]  [BX+DI+ сдвиг]  [BP+SI+ сдвиг]  [BP+DI+ сдвиг]  Метка [BX+SI]  Метка [BX+DI]  Метка [BP+SI]  Метка [BP+DI]  [Метка +BX+SI] [Метка +BX+DI]  [Метка +BP+SI]  [Метка +BP+DI] | DS  DS  SS  SS  DS  DS  SS  SS  DS  DS  SS  SS |

# Лекция 9. Структура машинной команды.

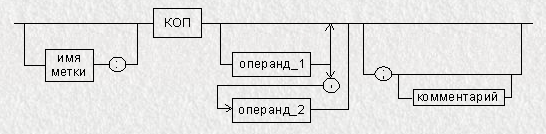
Программа на языке ассемблера представляет собой последовательность операторов, описывающих выполняемые операции.

Оператором (строкой) исходной программы может быть или ***команда****,* или (***директива***) Ассемблера.

Команды выполняются в процессе решения задачи на компьютере, а директивы - в процессе ассемблирования (трансляции) программы.

Команда (по [Максимов,Партыка,Попов,2006,с.460]) (instruction ) - описание элементарной операции, которую должен выполнить процессор. Обычно содержит код выполняемой операции, указания по определению операндов (или их адресов), указания по размещению получаемого результата.

Основной формат кодирования команд ассемблера имеет следующий вид:



Пример:

**Метка Команда Операнд Комментарий**

COUNT DB 1 ;Имя, команда, один операнд

MOV AX,0 ;Команда, два операнда

**Метка** нужна для ссылок на команду из других мест программы. Если метка есть, то после нее обязательно ставится двоеточие.

Метка в языке ассемблера может содержать следующие символы:

Буквы: от A до Z и от a до z

Цифры: от 0 до 9

Спецсимволы: (?), (.) (только первый символ), (@), (\_), ($)

Первым символом в метке должна быть буква или спецсимвол. Ассемблер не делает различия между заглавными и строчными буквами. Максимальная длина метки - 31 символ. Примеры меток: PAGE25, $E10. Рекомендуется использовать описательные и смысловые метки.

**Код операции (КОП)** - это мнемонические обозначения соответствующей машинной команды.

Операнды[Юров,2002,с.90].  - это  программные объекты  (числа, регистры или ячейки памяти), на которые действуют команды или директивы, либо это объекты, которые определяют или уточняют действие команд или директив.

В качестве операнда в команде может фигурировать *константа*, которая может быть введена в следующих формах:

* двоичной, как последовательность цифр 0 и 1 , заканчивающихся буквой В, например, 10111010В;
* десятичной (в привычной десятичной системе счисления с необязательной буквой D на конце), например, 129D или просто 129;
* шестнадцатеричной, как последовательность цифр от 0 до 9 и букв от А до F, заканчивающаяся буквой Н. Первым символом может быть только цифра от 0 до 9, например, 0Е23Н;
* литералом, строка букв, цифр и других символов, заключенная в кавычки или апострофы.

Операнды команды могут задаваться по-разному:

1) неявно на микропроцессорном уровне. Такие команды работают либо с регистрами, либо с флагами и в конкретной команде объект определяется кодом команды, например:

pushf - команда пересылки флагов в стек. Источник (откуда берутся данные) и приемник (куда пересылаются) определяются кодом команды;

cld - обнуление флага направления DF;

2) явно в самой команде (непосредственный операнд), например:

mov ах, 0;

3) одним из регистров общего назначения (РОН) или сегментным регистром;

add al,dl ;команда с регистровыми операндами

4) идентификатором переменной (меткой переменной, описанной в сегменте данных). В этом случае операнд располагается в ОЗУ. При косвенной адресации адрес-смещение переменной в ОЗУ помещается в РОН или индексный регистр и в команде используется ссылка на этот регистр;

data segment

mas\_w dw 25 dup (0)

…

code segment

…

lea si,mas\_w

5) номером порта ввода-вывода. В этом случае номер задается шестнадцатеричной константой;

in al,60h ;ввести байт из порта 60h

6) регистром стека, когда используются команды работы со стеком и в этом случае операнд либо заносится в вершину стека, либо считывается из вершины стека.

Важной особенностью машинных команд является то, что они не могут манипулировать одновременно 2-мя операндами, находящимися в ОЗУ. Это означает, что в команде только 1 операнд может указывать на ячейку ОЗУ, 2- должен быть либо регистром, либо непосредственно указывать значение. По этой причине возможны следующие сочетания операндов в команде:

*регистр - регистр;*

*регистр - память;*

*память - регистр;*

*регистр - непосредственный операнд;*

*память - непосредственный операнд.*

Для команд характерно, что если в них присутствуют 2 операнда, то 1- является приемником, а 2-источником, иначе говоря результат операции сохраняется по 1-ому адресу, вот почему 1-ый операнд никогда не может быть непосредственным операндом или иначе говоря, константой.

Местонахождение операнда команды определяет его адрес в памяти, который называется  исполнительным адресом операнда.

По количеству записываемых адресов (обозначаются а1, а2, а3,...) команды делят на безадресные, одно-, двух- и трехадресные.

Приведём типовые структуры команд:

(а)  трёхадресная команда;

Код операции

Адрес ячейки 1 или регистр

Адрес ячейки 2 или регистр

Адрес ячейки 3 или регистр

данные

результат

(б)  двухадресная команда;

(б)  двухадресная команда;

Код операции

Адрес ячейки 1 или регистр

Адрес ячейки 2 или регистр

данные

результат

Замечания (важные).

   1. Оба операнда команды не могут одновременно обозначать адрес ячейки памяти.

  2.  Поскольку первый операнд команды является "приёмником" результата, то, в отличии от второго операнда, он никогда не может представлять собой число.

(в)  одноадресная команда;

Код операции

Адрес ячейки 1 или регистра

данные

Замечание.

В зависимости от модификации команды операнд  Адрес\_ячейки\_1  может обозначать либо адрес ячейки (регистра), в которой хранится одно из чисел, участвующих в операции, либо адрес ячейки (регистра), куда следует поместить число - результат.

(г)  безадресная команда;

Код операции

  Информация для неё должна быть заранее помещена в определённые регистры машины.

**Команда Операнд Комментарий**

Нет операндов RET ;Вернуться

Один операнд INC CX ;Увеличить CX

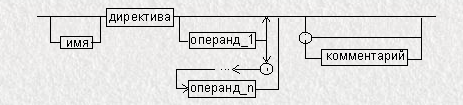
Два операнда ADD AX,12 ;Прибавить 12 к AX

**Комментарий**. В конце команды можно поместить комментарий. Для этого ставится точка с запятой и за ней помещается текст, поясняющий действие этой команды.

**Директивы**

Директивы ([Юров,2002,с.76])  - конструкции, которые не генерируют машинных команд, а являются указаниями компилятору на выполнение некоторых действий или служат для задания его работы. Другими словами,  директивы  [Ирвин,2002,с.71] - утверждения для информирования компилятора языка ассемблера о том, как создавать выполняемый код. У директив нет аналогов в машинном представлении.

Синтаксис директив следующий:

****

Как видно, формат директив в целом совпадает с форматом команд. Но в отличие от команд, в директиве после имени не ставится двоеточие.

**имя** — идентификатор, отличающий данную директиву от других одноименных директив. В результате обработки ассемблером определенной директивы этому имени могут быть присвоены определенные характеристики.

К директивам ассемблера относятся обозначения начала и конца сегментов segment и ends; ключевые слова, описывающие тип используемых данных (db, dw,dd, dup); специальные описатели сегментов вроде stack и т.д.

Директивы служат для передачи транслятору служебной информации, которой он пользуется в процессе компиляции программы. Однако в состав выполнимой программы, состоящей из машинных кодов, эти строки не попадут, так как процессору, выполняющему программу, они не нужны. Другими словами, операторы типа segment и ends не компилируются в машинные коды, а используются лишь самим ассемблером на этапе компиляции программы.

# Лекция 10. Структура программы на ассемблере.

Приступим к ответу на вопрос, как правильно оформить последовательность команд, чтобы компилятор языка ассемблера мог их обработать, а микропроцессор - выполнить результаты обработки.

Большинство программ на языке ассемблера можно разделить на пять основных частей:

* заголовок, который содержит установочную информацию;
* макроопределения, в которых определяются значения некоторых переменных;
* данные. В этой части определяются переменные, значения которых будут храниться в памяти;
* тело, которое содержит код программы;
* заключение, которое отмечает конец исходного текста.

Такое разделение осуществляется с помощью директив сегментации. Компилятор TURBO Assembler предоставляет два типа таких директив:  стандартные и  упрощённые.

Стандартные директивы используются, когда программист желает получить полный контроль над размещением сегментов в памяти и их комбинированием с сегментами других модулей.

Упрощенные директивы целесообразно использовать для простых программ и программ, предназначенных для связывания с программными модулями, написанными на языках высокого уровня. Это позволяет компоновщику эффективно связывать модули разных языков за счет стандартизации связей и управления.

Исполняемые программы подразделяются:

-  позиционно-независимые программы  (или  com-программы).

-  позиционно-зависимые программы  (или  exe-программы);

*Позиционно-зависимая программа* - программа, содержащая сведения обо всех используемых ею сегментах (данных, кода, стека и т.п.) и позиционно-зависимых командах  для их правильной настройки загрузчиком. Файл с данной программой имеет расширение ".exe".

*Позиционно-независимая программа* - программа, имеющая минимальный размер в связи с описанием и данных и команд в одном сегменте кода. Файл с данной программой имеет расширение ".com".

Образ программы в памяти начинается с  сегмента префикса программы (PSP), образуемого и заполняемого системой. Префикс программы всегда имеет размер 256 байтов и содержит таблицы и поля данных, используемые системой в процессе выполнения программы.

 Сегменты программы располагаются вслед за сегментом префикса программы в том порядке, как они объявлены в программе.

Сегментные регистры DS и ES указывают на начало сегмента префикса программы, CS - на начало сегмента команд, SS - на начало сегмента стека.

В указатель команд IP загружается относительный адрес точки входа в программу, а в указатель стека SP - величина, равная объявленному размеру стека, в результате чего указатель стека указывает на конец стека.

Теперь мы можем следующим образом изобразить отличие размещения в памяти com- и exe-программ:

PSP (256байт)

Сегмент стека

Сегмент данных

Сегмент кодов

Программа

IP

SS

DS,ES

CS

Сегмент кодов

Программа

PSP (256байт)

DS,ES

CS

IP

COM-файл EXE- файл

Таким образом, можно перечислить основные различия между exe- и com-программами:

1. exe-программа может иметь любой размер, в то время как размер com-программ не превышает размера сегмента (64 Кбайт);
2. com-программа содержит единственный сегмент code;
3. когда com-программа начинает работать, то все сегментные регистры содержат адрес префикса PSP размером 256 байт (100h).

Префикс программного сегмента организуется системой как при выполнении com-файла, так и при выполнении exe-файла. Разница заключается в том, что префикс программного сегмента в com-файлах расположен в кодовом сегменте, в то время как для exe-файла он создаётся вне кодового сегмента.

Таким образом, адресация в сегменте кода для com-файла должна начинаться со смещения 100h от начала PSP, и, следовательно, после директивы segment в программе должна стоять директива org 100h для пропуска PSP.

 Рассмотрим два варианта оформления *позиционно-зависимой* программы - в режиме MASM и в режиме IDEAL с использованием упрощенных и стандартных директив соответственно :

Режим MASM   Режим IDEAL

|  |  |
| --- | --- |
| ; Текстовый заголовок   MASM  MODEL  Small  STACK  100h  .DATA  ...  < Данные\_программы >  …  .CODE  ASSUME  ds:@data, es:@data    Start:  mov ax,@data  mov ds,ax  ...  < Код\_программы >  ...  mov ax,4C00h  int 21h  END  Start | %TITLE  "Текстовый заголовок"  IDEAL  MODEL  Small  STACK  256  DATASEG  ...  < Данные\_программы >  …  CODESEG  ASSUME  ds:@data, es:@data  Start:  mov ax,@data  mov ds,ax  ...  < Код\_программы >  ...  mov ax,4C00h  int 21h  END  Start |

**(1) Заголовок.**

Программа на языке ассемблера начинается с заголовка, в котором содержатся директивы, не приводящие к созданию машинного кода при компиляции.

%TITLE  "Текстовый заголовок"

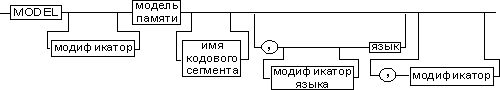
IDEAL

MODEL  Small

STACK  256

* Директива %TITLE (необязательная) предназначена для размещения некоторой строки, описывающей назначение программы. Директива IDEAL переводит Turbo Assembler в соответствующий режим. Если же Вы программируете на языке ассемблера MASM, то используйте директиву MASM.
* Директива MODEL (MODEL  Small) позволяет выбрать одну из нескольких моделей памяти, которые определяют набор сегментов программы, размеры сегментов данных и кода, способ связывания сегментов и сегментных регистров. Кроме того, модель памяти определяет размерность некоторых операций или данных, используемых в программе по умолчанию.

Синтаксис директивы MODEL:



Необходимость в использовании параметров < модификатор языка > и < модификатор> появляется при написании и связывании программ на различных языках программирования. Необязательный параметр <имя кодового сегмента> задаёт имя сегмента для моделей памяти с несколькими сегментами кода.

Обязательным параметром директивы MODEL является *модель памяти*. Этот параметр определяет модель сегментации памяти для программного модуля

Значениями параметра *модель памяти* директивы MODELмогут быть: 

* TINY - код, данные и стек содержатся в одном сегменте 64 Кбайт. используется только для com-программ
* SMALL - код и данные содержатся в различных сегментах, размером до 64 Кбайт. Используется для небольших и средних exe-программ
* Директива STACK резервирует место для  стека программы – области памяти, в которой содержатся два типа данных: промежуточные значения и адреса, на которые подпрограммы возвращают управление.

Формат  директивы описания стека: STACK [< Размер>], где необязательный параметр <Размер> задаёт размер стека в байтах. По умолчанию под стек выделяется 1 Кбайт памяти

**(2) Макроопределения.**

После заголовка программы следуют описания констант и переменных. В языке ассемблера существует возможность дать константам символические имена: это делается с помощью макроопределений , использующих директивы ассемблера EQU или = (читается: "равно").

Эти директивы предназначены для присвоения некоторому выражению символического имени или идентификатора. Впоследствии, когда в ходе компиляции этот идентификатор встретится в теле программы, макроассемблер подставит вместо него соответствующее выражение. В качестве выражения могут быть использованы константы, имена меток, символические имена и строки в апострофах. После присвоения этим конструкциям символического имени его можно использовать везде, где требуется размещение данной конструкции.

Пример

Count EQU 10

Element EQU 5

MyName EQU "Tanya"

Size = 0

Size = Count\*Element

Макроопределения могут располагаться в любом месте программы, но для удобства располагайте их сразу же после заголовка.

**(3) Данные.**

 Сегмент данных обычно располагается между макроопределениями и командами программы. Он должен начинаться с директивы DATASEG (.DATA), которая позволяет разместить данные, указанные в этом сегменте, в памяти (для модели small - максимальный размер памяти 64 Кбайт).

Сегмент данных может содержать два типа переменных:

-инициализированные переменные. При выполнении программы инициализированные переменные имеют значения, определённые в тексте программы. Эти переменные автоматически загружаются в память и их значения доступны при выполнении программы;

-неинициализированные переменные. Они имеют неопределённые значения и не занимают памяти при выполнении программы.

Для описания  простых типов данных языка ассемблера в программе используются специальные директивы резервирования и инициализации данных, которые, по сути, являются указаниями компилятору на выделение определённого объёма памяти.

 Семантику директив резервирования и инициализации данных простых типов языка ассемблера представим в виде следующей таблицы:

|  |  |  |  |
| --- | --- | --- | --- |
| Директива | Название | Число выделяемых байтов | Характерное использование |
| DB | Определить байт | 1 | Байты, строки |
| DW | Определить слово | 2 | Целые числа |
| DD | Определить двойное слово | 4 | Длинные целые числа |
| DP | Определить указатель | 6 | 32-битовый указатель |
| DF | Определить дальний  указатель | 6 | 48-битовый указатель |
| DQ | Определить учетверённое  слово | 8 | Вещественные числа |
| DT | Определить десять байтов | 10 | Двоично-десятичные числа |

Например

a DB 25

b DB 80

c DW 0B00h

Имена переменных (a, b, c) являются метками, которые указывают на выделяемые в памяти области - на пространство, резервируемое под значения переменных. Компилятор ассемблера преобразует метки в адреса памяти, по которым содержатся переменные.

**(4) Тело программы**.

После сегмента данных располагается тело программы, известное под названием кодового сегмента - раздела памяти, который содержит исполняемый код Вашей ассемблерной программы. Тело программы должно начинаться с директивы (.CODE) CODESEG.

Все сегменты сами по себе равноправны. Чтобы использовать их как сегменты кода, данных или стека, необходимо предварительно сообщить компилятору об этом, для чего используется директива ASSUME:

***ASSUME*** *<имя сегментного регистра>****:****<имя сегмента или* ***nothing****>[, <имя сегментного регистра>****:****<имя сегмента или* ***nothing****>, …]*

Здесь  имя сегмента - это имя одного из сегментных регистров: CS, DS, SS, ES, FS, GS. Эта директива сообщает компилятору о том, какой сегмент к какому сегментному регистру привязан. В свою очередь, это позволит компилятору корректно связывать символические имена, определённые в сегментах.

ASSUME  ds:@data, es:@data

Если в качестве операнда используется только ключевое слово nothing, то предшествующее назначение сегментных регистров аннулируется, причём сразу для всех шести сегментных регистров. Если же nothing используется вместо аргумента <Имя сегмента>, то разрывается связь между сегментом с именем <Имя сегмента> и соответствующим сегментным регистром.

Например, ASSUME cs:code, ds:data, es:nothing

Далее идет последовательность команд, составляющая "тело" программы

**(5) Заключение**

Завершение программы и возврат управления операционной системе: (AH)=4Ch- номера DOS-функции, (AL)=0-код выхода–успешного завершения программы

mov ax,4C00h

int 21h

  END  Start

Директива END помечает конец текста исходной программы. Справа от директивы END Вы должны определить метку, с которой Вы хотите начать выполнение программы. Обычно эта метка совпадает с меткой, указывающей на первую команду, следующую за директивой ассемблера CODESEG.

Пример.  Демонстрация использования директив резервирования и инициализации данных (упрощенные директивы).

%TITLE "Директивы резервирования и инициализации данных"

MASM ; Режим

MODEL small ; Модель памяти

.STACK 100h ; Сегмент стека

.DATA ; Сегмент данных

message DB "Запустите эту программу в отладчике",'$'

perem\_1 DB 0FFh

perem\_2 DW 3A7Fh

perem\_3 DD 0F54D567Ah

mas DB 10 DUP (" ")

pole\_1 DB 5 DUP (?)

adr DW perem\_3

adr\_full DD perem\_3

fin DB "Конец сегмента данных программы $"

.CODE

Start:

mov ax,@data

mov ds,ax

mov ah,09h

mov dx,offset message

int 21h

mov ax,4c00h

int 21h

END Start

# 

# Лекция 11. Система команд микропроцессора 8086. Команды пересылки данных

Удобнее для изучения команд представить их как совокупность групп команд, выполняющих похожие функции. С этой целью можно разделить систему команд на 7 функциональных групп:

1. Команды пересылки данных, осуществляющие обмен информацией.
   * общего назначения;
   * работы со стеком;
   * преобразования данных;
2. Арифметические команды, выполняющие арифметические операции над числами со знаком и без знака.
   * двоичной арифметики
   * десятичной арифметики
3. Команды манипулирования битами выполняющие логические операции и сдвиги содержимого регистров и ячеек памяти.
   * манипуляции битами;
   * сдвига;
4. Команды передачи управления, управляющие последовательностью исполнения команд программы.
   * безусловные;
   * условные;
5. Команды обработки строк для перемещения, сравнения и сканирования символьных данных.
6. Команды прерывания, вызывающие обработку микропроцессором некоторых специфических ситуаций.
7. Команды управления процессором, устанавливающие и сбрасывающие флаги состояния, а также изменяющие режим функционирования микропроцессора.

* работы с флагами;
* работы с системными регистрами.

## Команды пересылки данных

(а) **MOV** (от англ.  move  - "движение")

*MOV < Первый операнд >,< Второй операнд >*

*Приёмник     Источник*

Содержимое второго операнда становится содержимым первого операнда.

Пример.

MOV AX,BX ;  содержимое регистра BX копируется в регистр AX

Отметим *особенности*  *синтаксиса и семантики* команды (следуя [Юров,2002,с.135-136]):

*-*командой MOV нельзя осуществить пересылку из одной области памяти в другую. Если такая необходимость возникает, то нужно использовать в качестве промежуточного буфера любой доступный в данный момент регистр общего назначения.

- нельзя загрузить в сегментный регистр значение непосредственно  из памяти. Поэтому для выполнения такой загрузки нужно использовать промежуточный объект. Это может быть регистр общего назначения или стек;

- нельзя переслать содержимое одного сегментного регистра в другой сегментный регистр. Это объясняется тем, что в системе команд нет соответствующего кода операции. Но необходимость в таком действии часто возникает. Выполнить такую пересылку можно, используя в качестве промежуточных всё те же регистры общего назначения.

- нельзя использовать сегментный регистр cs в качестве первого  операнда. Причина здесь простая. Дело в том, что в архитектуре микропроцессора пара cs:ip всегда содержит адрес команды, которая должна выполняться следующей. Изменение командой MOV содержимого регистра cs фактически означало бы операцию перехода, а не пересылки, что недопустимо.

 (б**)  XCHG**

*XCHG < Первый операнд >,< Второй операнд >*

Команда производит двунаправленную пересылку данных. Естественно, что операнды должны иметь одинаковый тип. Не допускается (как и  для всех команд ассемблера) обменивать между собой содержимое двух ячеек памяти.

 Примеры

  xchg ax,bx ; Обменять содержимое регистров ax и bx

xchg ax,word ptr[si] ; Обменять содержимое регистра ax и слова

; в памяти по адресу в регистре SI

(в**) LEA** (от англ.  load effective address - "загрузка эффективного адреса")

*LEA < Назначение >,< Источник >*

Команда lea "похожа" на команду mov тем, что она также производит пересылку. Однако команда lea производит пересылку не данных, а  эффективного адреса данных (т.е. смещения данных относительно начала сегмента данных) в регистр, указанный операндом <Назначение>.

(г) **LDS** (" Загрузка указателя с использованием DS ")

*LDS < Назначение >,< Источник >*

Команда LDS загружает в два регистра 32-битный указатель, расположенный в памяти по адресу <Источник >. При этом старшее слово заносится в сегментный регистр DS, а младшее слово - в базовый регистр < Назначение >. В качестве операнда <Назначение> может выступать любой 16-битный регистр, кроме сегментных.

(д) **LES**  ("Загрузка указателя с использованием ES")

*LES <Назначение>,<Источник>*

Команда LES загружает в два регистра 32-битный указатель, расположенный в памяти по адресу <Источник >. При этом старшее слово заносится в сегментный регистр ES, а младшее слово - в базовый регистр <Назначение>. В качестве операнда <Назначение> может выступать любой 16-битный регистр, кроме сегментных.

## Команды преобразования типов

В системе команд микропроцессора есть несколько команд, которые могут облегчить программирование алгоритмов, производящих арифметические вычисления. В них могут возникать различные проблемы, для разрешения которых разработчики микропроцессора предусмотрели несколько команд.

Что делать, если размеры операндов, участвующих в арифметических операциях, разные? Например, предположим, что в операции сложения один операнд является словом, а другой занимает двойное слово. В этом случае можно на базе исходного операнда сформировать новый (формата двойного слова), старшие разряды которого просто заполнить нулями.

Для решения подобных проблем в системе команд микропроцессора есть  команды преобразования типов. Эти команды расширяют байты в слова, слова - в двойные слова и двойные слова - в учетверённые слова (64-разрядные значения). Команды преобразования типа особенно полезны при преобразовании целых со знаком, т.к. они автоматически заполняют старшие биты вновь формируемого операнда значениями знакового бита старого объекта. Эта операция приводит к целым значениям того же знака и той же величины, что и исходная, но уже в более длинном формате.

Перечислим команды преобразования типа [Юров,2002,с,171].

1. Команды без операндов  - эти команды работают с фиксированными регистрами:

  - **CBW** (convert byte to word ) - команда преобразования байта (в регистре AL) в слово (в регистре AX) путём распространения значения старшего бита AL на все биты регистра AH;

- **CWD** (convert word to double) - команда преобразования слова (в регистре AX) в двойное слово (в регистрах DX:AX) путём распространения значения старшего бита AX на все биты регистра DX;

- **CWDE** (convert word to double) - команда преобразования слова (в регистре AX) в двойное слово (в регистре EAX) путём распространения значения старшего бита AX на все биты регистра EAX;

- **CDQ**  (convert double word to quarter word) - команда преобразования двойного слова (в регистре EAX) в учетверённое слово (в регистрах EDX:EAX) путём распространения значения старшего бита EAX на все биты регистра EDX.

2. Команды с операндами

- **MOVZX** < Операнд\_1 >,< Операнд\_2 > - переслать с расширением нулём. Расширяет 8- или 16-разрядное значение  Операнд\_2  до 16- или 32-разрядного с очисткой (заполнением) нулями старших позиций  Операнд\_2. Данную команду удобно использовать для преобразования 8- или 16- разрядных чисел в числа большей разрядности.

mov bl,22h

movzx ax,bl ; ax=0022h

## Команды преобразования данных.

Из всей совокупности команд микропроцессора непосредственно к командам преобразования данных можно отнести только одну команду (Юров, 2002):

**xlat [адрес\_таблицы\_перекодировки]**

Ее действие заключается в том, что она замещает значение в регистре al другим байтом из таблицы в памяти, расположенной по адресу, указанному операндом *адрес\_таблицы\_перекодировки*.

Слово “таблица” весьма условно — по сути это просто строка байт. Адрес байта в строке, которым будет производиться замещение содержимого регистра al, определяется суммой (bx) + (al), то есть содержимое al выполняет роль индекса в байтовом массиве.

При работе с командой xlat обратите внимание на следующий тонкий момент. Несмотря на то, что в команде указывается адрес строки байт, из которой должно быть извлечено новое значение, этот адрес должен быть предварительно загружен (например, с помощью команды lea) в регистр bx. Таким образом, операнд *адрес\_таблицы\_перекодировки* на самом деле не нужен (необязательность операнда показана заключением его в квадратные скобки). Что касается строки байт (таблицы перекодировки), то она представляет собой область памяти размером от 1 до 255 байт (диапазон числа без знака в 8-битном регистре).

# Лекция 12. Целочисленные арифметические команды двоичной арифметики

Группа арифметических целочисленных команд работает с двумя типами чисел:

-  *целыми двоичными числами.* Числа могут иметь знаковый разряд

или не иметь такового, т.е. быть числами со знаком или без знака;

-  *целыми десятичными числами*.

 Для организации вычислений с целыми двоичными числами используются  целочисленные арифметические команды двоичной арифметики:

ADD, INC, ADC, SUB, DEC, SBB, MUL, IMUL, DIV, IDIV, NEG.

└────────┘ └────────┘  └────┘  └────┘  └─┘

 Сложение Вычитание Умножение Деление Изменение

  знака

Основными параметрами результата, которые необходимо контролировать при выполнении арифметических операций с целыми числами, являются признаки переполнения разрядной сетки и знака результата.

 (а) **ADD** (от англ. add - "сложение")

*ADD < Первый операнд >,<Второй операнд>*

 Приёмник   Источник

Команда производит сложение как восьмибитовых, так и шестнадцатибитовых операндов. *Результат помещается на место операнда-приёмника, второй операнд остаётся без изменения.*

 Пример.

(1) Исходные данные содержатся в регистрах.

    ... ...

mov ax,58h ax:=58h

mov bx,27h bx:=27h

add ax,bx ax:=ax+bx = 7FH

  (2) Исходные данные содержатся в оперативной памяти.

  ...

mov ax,Second

mov bx,ax bx:=4; { Второе число }

mov ax,First ax:=5; { Первое число }

add ax,bx ax:=ax+bx { Результат }

mov Rezult,ax

(б) **INC** (от англ.  increment - " увеличение ")

*INC <Операнд>*

Команда увеличивает содержимое операнда на 1.

(в) **ADC**(от англ.  add with carry - " сложение с переносом ")

*ADC < Первый операнд >,< Второй операнд >*

 Приёмник   Источник

Команда выполняет сложение двух указанных операндов и содержимого флага переноса CF. Результат помещается на место операнда-приёмника, второй операнд остаётся без изменений.

Перенос возникает если сумма не помещается в операнде-приёмнике. Например, команда ADC AL,CL складывает восьмибитовые значения регистров AL и CL и возвращает результат в регистр AL. Если результат имеет длину 9 битов, то 8 битов будут занесены в AL, а девятый бит - в флаг переноса CF.

Таким образом, команда ADC является средством микропроцессора для сложения длинных двоичных чисел, размерность которых превосходит поддерживаемые микропроцессором длины стандартных полей.

Пример. Найдём сумму чисел 240=F0h и 120=78h. Первое число F0h

положим в регистр AL, второе число 78h - в регистр CL, сумму чисел поместим в регистр AX.

  mov ah,0 ah:=0

mov ch,0 ch:=0

mov al,F0h al:=F0h

mov cl,78h cl:=78h

add al,cl al:=al+cl

(г) **SUB** (от англ.  substract - " вычитание ")

*SUB <Первый операнд>,<Второй операнд>*

Приёмник      Источник

Команда определяет разность уменьшаемого и вычитаемого. Результат помещается на место уменьшаемого, вычитаемое остаётся без изменений.

 Пример. Пусть регистр AX содержит число 58H, а регистр BX содержит число 20H. После выполнения команды SUB AX,BX в ячейку памяти AX будет записано число 38H.

mov ax,58h ax:=58h

mov bx,20h bx:=20h

sub ax,bx ax:=ax-bx = 38h

(д) **DEC** (от англ.  decrement─ " уменьшение")

*DEC < Операнд >*

Команда уменьшает содержимое операнда на 1.

(е) **SBB** (от англ.  substract with borrow  - " вычитание с заёмом ")

*SBB <Первый операнд>,<Второй операнд>*

 Приёмник   Источник

Команда определяет разность двух операндов, и, кроме того, из уменьшаемого вычитается содержимое флага переноса. Результат помещается на место первого операнда, предыдущее значение которого теряется. Содержимое второго операнда не изменяется.

Пример. Пусть регистр AX содержит число 4803H, а флаг переноса CF установлен в 1. После выполнения команды SBB AX,1500H в регистре AX будет записано 3302H.

mov ax,4803h ax:=4803h

sbb ax,1500h ax:=ax-1500h-CF = 3302h

(ж) **MUL** (от англ.  multiply - "умножение")

*MUL <Операнд>*

 Источник

Производит беззнаковое умножение содержимого регистра AX (при умножении слов) или содержимого регистра AL (при умножении байтов) на указанный операнд.

Если операнд является байтом, то он умножается на содержимое регистра AL и результат двойной длины (слово) записывается в регистр AX (в AH - старший байт, в AL - младший байт).

Если операнд является словом, то его значение умножается на содержимое регистра AX и результат (двойное слово) записывается в пару регистров DX и AX, причём регистр DX содержит старшие разряды результата.

Операнд может быть задан в регистре общего назначения или ячейке памяти и интерпретируется как число без знака.

Если содержимое регистра AH после однобайтового умножения или содержимое регистра DX после двухбайтового умножения не равны нулю, то флаг CF (флаг переноса) и OF (флаг переполнения) устанавливаются в 1. В противном случае они сбрасываются в 0.

 Примеры.

(1) Пусть регистр AL содержит число 30H, а регистр CL - 12H и выполняется команда MUL CL. Результат умножения (12H\*30H=360H) записывается в регистр AX. Поскольку содержимое регистра AH отлично от 0, флаги CF и OF установлены в 1.

  mov al,30h al:=30h

mov cl,12h cl:=12h

mul cl ax:=al\*cl =360h

(2) Пусть теперь регистр AX содержит число 1000H, регистр BX - число 550H. Тогда после выполнения команды MUL BX результат умножения (1000H\*550H=550000H) будет записан в пару регистров DX и AX.

Так как новое значение регистра DX (0055H) отлично от нуля, то флаги CF и OF устанавливаются в 1.

mov ax,1000h ax:=1000h

mov bx,550h bx:=550h

mul bx dx\_ax:=ax\*bx = 0055h

(з) **IMUL** (от англ.  integer multiply - "умножение целых чисел")

*IMUL < Операнд >*

Источник

Производит умножение восьмибитовых и шестнадцатибитовых сомножителей со знаком. Если не принимать во внимание, что перемножаются значения со знаком, то операционная семантика команды IMUL совпадает с операционной семантикой команды MUL.

Результатом байтового умножения является 16-битовое число, хранящееся в регистре AX. Результат умножения слов - 32-битовое число в паре регистров DX и AX, причём в DX - старшие 16 бит результата. Если старшая часть (AX при байтовом умножении или DX при длинах операндов, равных слову) содержит не знак младшей части результата, а его значащие цифры, то флаги CF и OF переводятся в единичное состояние. В противном случае оба флага обнуляются.

 Пример.

mov al,0FEh al:=-2

mov ah,10h ah:=10

imul ah ax:=al\*ah = -20=FFE0h

... ...

(и**) DIV** (от англ.  divide - "деление")

*DIV <Операнд>*

Источник

Осуществляет деление содержимого регистра AX (при делении байтов) или пары регистров DX и AX (при делении слов) на указанный операнд без учёта знаков.

При делении байтов целая часть отношения помещается в регистр AL, а остаток - в регистр AH:

(AL) → частное

(AH) →остаток

Если при этом (AX)/(Операнд)>FFH, то генерируется прерывание типа 0 (ошибка при делении).

Для двухбайтовых операций выполняется деление содержимого регистровой пары DX,AX на указанный операнд. Если результат деления больше FFFFH, то генерируется прерывание типа 0 (ошибка при делении). При 16-битовом делении целая часть отношения размещается в регистре AX, а остаток - в регистре DX (другими словами, (AX) →частное, (DX) →остаток).

Операнд может быть задан в регистре общего назначения или ячейке памяти, при этом операнд рассматривается как беззнаковый 8- или 16-битовый делитель соответственно при однобайтовом и двухбайтовом делении.

 Примеры.

(1) Пусть регистр AX содержит число 0047H, регистр CL - число 12H и выполняется команда DIV CL. Частное от деления (03H=47H/12H) записывается в регистр AL, а остаток 11H - в регистр AH.

mov ax,0047 ax:=0047h

mov cl,12h cl:=12h

div cl ax:=al/cl =11h

(2) Рассмотрим случай, когда регистры DX и AX содержат соответственно коды 0012H и F043H, представляющие 32-разрядное шестнадцатеричное число 0012F043H, регистр BP содержит число 4320H и выполняется команда DIV BP. Частное от деления (0048H=12F043H/4320H) записывается в регистр AX, а остаток (0F43H)-в регистр DX.

mov ax,F043h dxax:=0012F043h

mov dx,0012h bp:=4320h

mov bp,4320h

div bp dx\_ax:=dxax частное от деления 0048h→ax,

остаток от деления0А43h→dx.

(к) **IDIV** (от англ.  integer divide - "деление целых чисел").

*IDIV <Операнд>*

Источник

Выполняет деление содержимого регистра AX (при делении байтовых чисел) или пары регистров DX и AX (при делении слов) на указанный операнд с учетом знаков.

При 8-битовом делении целая часть отношения размещается в регистре AL, а остаток - регистре AH. При этом делении максимальное положительное частное не превышает 127 (7FH), а минимальное отрицательное не может быть меньше -127 (81H).

При делении чисел длиной в слово регистр AX содержит частное, а регистр DX - остаток. При этом максимальное положительное частное равно 32767 (7FFFH), а минимальное отрицательное частное равно -32767 (8001H).

Если отношение меньше допустимого минимума или больше максимума, а также, если произведена попытка деления на 0, то микропроцессор автоматически генерирует прерывание типа 0 (прерывание из-за ошибки  деления). Отношение всегда "усекается" до целой части.

(л) **NEG** (от англ.  negative - "отрицательный")

*NEG <Операнд>*

Источник

Команда выполняет инвертирование значения операнда. Физически команда выполняет одно действие (Операнд) = - (Операнд), т.е. вычитает операнд из 0.

Команду NEG можно применять для [Юров,2002,с.173]:

(1) смены знака у операнда;

(2) выполнения вычитания из константы. Дело в том, что команды SUB и SBB не позволяют вычесть что-либо из константы, т.к. константа не может служить операндом-приёмником в этих операциях. Поэтому данную операцию можно выполнить с помощью двух команд:

neg ax; Смена знака содержимого ax

...

add ax,340; Вычитание (ax)=340-(ax)

## Команда сравнения как команда с арифметическим принципом действия

**CMP**  (от англ.  compare  - " сравнение ").

*CMP <Первый операнд>,<Второй операнд>*

  Приёмник     Источник

  Не допускается использовать для записи операндов сегментный регистр или записывать оба операнда одновременно в ячейки оперативной памяти.

 Команда CMP вычитает из содержимого первого операнда содержимое второго операнда. Результат операции изменяет флаги. *Сами операнды не изменяются.* Первый операнд может быть записан в регистре, либо в ячейке памяти. Второй операнд может быть задан в регистре, ячейке памяти, либо непосредственным операндом.

Команда осуществляет сравнение операндов.

Укажем  значение флагов после выполнения команды  CMP:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Условие | CF | ZF | SF | OF | PF | AF |
| Операнды без знака; | | | | | | |
| приёмник < источник | 1 | 0 | 1 | 0 | 1 | 1 |
| приёмник = источник | 0 | 1 | 0 | 0 | 1 | 0 |
| приёмник > источник | 0 | 0 | 0 | 0 | 0 | 0 |
| Операнды со знаком; | | | | | | |
| приёмник < источник | 1 | 0 | 1 | 0 | 1 | 1 |
| приёмник = источник | 0 | 1 | 0 | 0 | 1 | 0 |
| приёмник > источник | 0 | 0 | 0 | 0 | 0 | 0 |

 Пример

mov si,100

cmp si,0; ZF=0 and SF=0

# Лекция 13. Команды манипулирования битами

Наряду со средствами арифметических вычислений, система команд микропроцессора имеет также средства логического преобразования данных. К средствам логического преобразования данных относятся логические команды и логические операции.

В языке ассемблера имеется следующий набор логических операций:

and (операция "И");

or (операция "ИЛИ");

xor (операция "исключающее ИЛИ");

not (операция "НЕ" или 1  0"отрицание").

Приведём также таблицу истинности для логических операций над битами:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Бит источника | Бит приемника | AND | OR | XOR | NOT | TEST |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 |

## Логические команды

Система команд микропроцессора содержит пять команд, поддерживающих данные операции. Эти команды выполняют логические операции над битами операндов. Размерность операндов, естественно, должна быть одинакова. Например, если размерность операндов равна слову (16 бит), то логическая операция выполняется сначала над нулевыми битами операндов и ее результат записывается на место бита 0 результата. Далее команда последовательно повторяет эти действия над всеми битами с первого до пятнадцатого.

**and операнд\_1,операнд\_2** — операция логического умножения.

Команда выполняет поразрядно логическую операцию И (конъюнкцию) над битами операндов операнд\_1 и операнд\_2. Результат записывается на место операнд\_1.

Пример. Пусть регистр CX содержит слово 02F7H. Выполним команду AND CX,F300H.

    Значения операндов

CX (Первый операнд) 02F7H 0000 0010 1111 0111

 Второй операнд      F300H 1111 0011 0000 0000

AND CX,F300H   0200H 0000 0010 0000 0000 ───> CX

(Первый операнд)

**or операнд\_1,операнд\_2** — операция логического сложения.

Команда выполняет поразрядно логическую операцию ИЛИ (дизъюнкцию) над битами операндов операнд\_1 и операнд\_2. Результат записывается на место операнд\_1.

Пример. Пусть регистр CX содержит слово 02F7H. Выполним команду OR CX,F300H.

    Значения операндов

CX (Первый операнд ) 02F7H 0000 0010 1111 0111

Второй операнд    F300H 1111 0011 0000 0000

OR CX,F300H     F3F7H 1111 0011 1111 0111 ───> CX

(Первый операнд)

**xor операнд\_1,операнд\_2** — операция логического исключающего сложения.

Команда выполняет поразрядно логическую операцию исключающего ИЛИ над битами операндов операнд\_1 и операнд\_2. Результат записывается на место операнд\_1.

Пример. Пусть регистр CX содержит слово 02F7H. Выполним команду XOR CX,F300H.

   Значения операндов

CX (Первый операнд)  02F7H 0000 0010 1111 0111

 Второй операнд    F300H 1111 0011 0000 0000

XOR CX,F300H     F1F7H 1111 0001 1111 0111 ───> CX

(Первый операнд)

**test операнд\_1,операнд\_2** — операция “проверить” (способом логического умножения).

Команда выполняет поразрядно логическую операцию И над битами операндов операнд\_1 и операнд\_2. Состояние операндов остается прежним, изменяются только флаги zf, sf, и pf, что дает возможность анализировать состояние отдельных битов операнда без изменения их состояния.

**not операнд** — операция логического отрицания.

Команда выполняет поразрядное инвертирование (замену значения на обратное) каждого бита операнда. Результат записывается на место операнда.

Логические операции, кроме NOT, устанавливают флаги SF (знака), ZF (нуля) и PF(четности), флаг переноса устанавливается в 0. Все команды, за исключением TEST, сохраняют результат логической операции в приемнике. Команда TEST выполняет такую же операцию, что и AND. Различие только в том, что результат операции TEST не сохраняется в приемнике, а используется для следующей команды перехода по нулю.

Логические команды хороши тем, что выполняются быстро, быстрее “коротких” команд сложения и вычитания. Приведем некоторые полезные приемы применения этих команд.

* Часто встречающийся в программах прием – использование логической команды “исключающее или” (XOR) для обнуления регистра:

*XOR AX,AX.*

* Поскольку логические команды устанавливают флаги, их можно применять для проверки значений регистров.

*OR DX,DX* ;проверить значение в регистре.

Следует отметить, что эта команда не изменит значения в регистре DX, но при этом, если в регистре хранится нулевое значение, будет установлен в 1 флаг ZF; если старший бит в регистре единичный (что свидетельствует об отрицательном значении при представлении его как числа со знаком), то будет установлен в 1 флаг знака SF.

* При проверки четности или нечетности значения в регистре, можно воспользоваться командой

*AND DX,1H* ;выделить младший бит, учитывая, что двоичное представление четных чисел имеет нулевой последний бит. Следует отметить, что при использовании команды AND значение в регистре будет этой командой изменено. Если необходимо оставить значение в регистре DX без изменений, то следует воспользоваться командой TEST, например, таким образом:

*TEST DX,00000001B* ;проверить младший бит.

Следующие две команды позволяют осуществить поиск первого установленного в 1 бита операнда. Поиск можно произвести как с начала так и от конца операнда:

**bsf операнд\_1,операнд\_2** (Bit Scaning Forward) - сканирование битов вперед.

Команда просматривает (сканирует) биты операнд\_2 от младшего к старшему (от бита 0 до старшего бита) в поисках первого бита, установленного в 1. Если таковой обнаруживается, в операнд\_1 заносится номер этого бита в виде целочисленного значения. Если все биты операнд\_2 равны 0, то флаг нуля zf устанавливается в 1, в противном случае флаг zf сбрасывается в 0.

mov al,02h

bsf bx,al ;bx=1

jz m1 ;переход, если al=00h

...

**bsr операнд\_1,операнд\_2** (Bit Scaning Reset) — сканирование битов в обратном порядке.

Команда просматривает (сканирует) биты операнд\_2 от старшего к младшему (от старшего бита к биту 0) в поисках первого бита, установленного в 1. Если таковой обнаруживается, в операнд\_1 заносится номер этого бита в виде целочисленного значения.

При этом важно, что позиция первого единичного бита слева отсчитывается все равно относительно бита 0. Если все биты операнд\_2 равны 0, то флаг нуля zf устанавливается в 1, в противном случае флаг zf сбрасывается в 0.

## Команды сдвига

Команды этой группы также обеспечивают манипуляции над отдельными битами операндов, но иным способом, чем логические команды, рассмотренные выше. Все команды сдвига перемещают биты в поле операнда влево или вправо в зависимости от кода операции.

Все команды сдвига имеют одинаковую структуру:

**КОП операнд, счетчик\_сдвигов**

Как видите на месте второго операнда располагается значение счетчика сдвигов. Количество сдвигаемых разрядов - счетчик\_сдвигов может задаваться двумя способами:

* статически, что предполагает задание фиксированного значения с помощью непосредственного операнда;
* динамически, что означает занесение значения счетчика сдвигов в регистр CL перед выполнением команды сдвига.

Значение счетчика сдвигов должно быть целым числом без знака, да к тому же и лежать в разумных пределах (в последних публикациях указывается ограничение, равное 31. Правда, говорится, что оно может быть и большим, до 255, но команды сдвига воспринимают из этого значения- источника только младшие 5 двоичных разрядов).

Все команды сдвига устанавливают флаг переноса **cf**.

По мере сдвига битов за пределы операнда они сначала попадают на флаг переноса, устанавливая его равным значению очередного бита, оказавшегося за пределами операнда. Куда этот бит попадет дальше, зависит от типа команды сдвига и алгоритма программы.

По принципу действия команды сдвига можно разделить на два типа:

**команды линейного сдвига;**

**команды циклического сдвига.**

## Команды линейного сдвига

К командам этого типа относятся команды, осуществляющие сдвиг по следующему алгоритму:

* очередной “выдвигаемый” бит устанавливает флаг cf;
* бит, вводимый в операнд с другого конца, имеет значение 0;
* при сдвиге очередного бита он переходит во флаг cf, при этом значение предыдущего сдвинутого бита теряется!

Команды линейного сдвига делятся на два подтипа:

* команды логического линейного сдвига;
* команды арифметического линейного сдвига.

**SHL операнд, счетчик\_сдвигов** -**логический сдвиг влево**

**–** каждый бит операнда сдвигается влево на количество битов, определяемое значением счетчик\_сдвигов. Освобождаемые позиции в младших разрядах заполняются нулями. Если представить эту операцию как серию сдвигов на 1 позицию, то старший разряд перед последним сдвигом попадает во флаг CF.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

0

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

CF

**SHR** **операнд, счетчик\_сдвигов** - **логический сдвиг вправо**

**–** каждый бит операнда сдвигается вправо на количество битов, определяемое значением счетчик\_сдвигов. Освобождаемые позиции в старших разрядах заполняются нулями. Если представить эту операцию как серию сдвигов на 1 позицию, то младший разряд перед последним сдвигом попадает во флаг CF.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

0

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

CF

**SАL** **операнд, счетчик\_сдвигов** **арифметический сдвиг влево**

**–** каждый бит операнда сдвигается влево на количество битов, определяемое значением счетчик\_сдвигов. Освобождаемые позиции в младших разрядах заполняются нулями. Знак операция не сохраняет, но устанавливает в 1 флаг CF, если изменяется знаковый разряд очередным сдвигаемым битом, поэтому на приведенной ниже иллюстрации команды связь с флагом обозначена прерывистой линией.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

0

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

CF

**SАR операнд, счетчик\_сдвигов** **арифметический сдвиг вправо**

**–** каждый бит приемника сдвигается влево на количество битов, определяемое значением счетчик\_сдвигов. Команда сохраняет знак операнда-приемника, восстанавливая его после сдвига каждого очередного бита.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

CF

Все описанные команды позволяют умножать (при сдвиге влево) или делить (при сдвиге вправо) значение источника на степени числа 2. При этом скорость выполнения команд сдвига намного больше, чем скорость операций умножения или деления.

## Команды циклического сдвига

К командам циклического сдвига относятся команды, сохраняющие значения сдвигаемых бит. Есть два типа команд циклического сдвига:

**команды простого циклического сдвига;**

*команды циклического сдвига через флаг переноса cf.*

К командам простого циклического сдвига относятся:

**ROL операнд,счетчик\_сдвигов** (Rotate Left) — циклический сдвиг влево.

**ROR операнд,счетчик\_сдвигов (**Rotate Right**) —** циклический сдвиг вправо.

Циклический сдвиг ROL и ROR отличается от логического тем, что выдавливаемые за пределы разрядной сетки биты операнда-приемника записываются в него же на освобождаемые справа или слева позиции. Последний выдавливаемый справа или слева разряд записывается во флаг CF.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

CF

Команды циклического сдвига через флаг переноса отличаются от предыдущих тем, что выдавливаемые биты становятся значением флага CF и только на следующем выполнении этих команд (при условии, что они выполняются в цикле) бит из флага CF заносится в младший или старший биты операнда-приемника.

**RCL операнд, счетчик\_сдвигов** -циклический сдвиг влево через перенос.

**RCR операнд, счетчик\_сдвигов** -циклический сдвиг вправо через перенос.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

CF

Занесение бита после предыдущего сдвига

Можно использовать команды циклических и обычных сдвигов для умножения и деления чисел в формате двойного слова на степени двойки. Рассмотрим 32-битовое число, в котором старшие 16 бит хранятся в DX, а младшие – в AX.

Инструкции для **умножения** этого числа таковы:

SHL AX,1 ;сдвигаем все биты влево, а старший – в CF

RCL DX,1 ;помещаем бит из CF в DX, предварительно сдвинув все биты регистра влево.

Для **деления** на 2 этого числа нужно применить команды:

SAR DX,1 ;сдвигаем биты вправо, а младший – в CF

RCR AX,1 ;все биты АХ сдвигаем вправо, а на освободившееся место записываем выдвинутый на предыдущем шаге бит из DX.

## Дополнительные команды сдвига

Система команд последних моделей микропроцессоров Intel, начиная с i80386, содержит дополнительные команды сдвига, расширяющие возможности, рассмотренные нами ранее.

Это — команды сдвигов двойной точности:

**shld операнд\_1,операнд\_2,счетчик\_сдвигов** — сдвиг влево двойной точности.

Команда shld производит замену путем сдвига битов операнда операнд\_1 влево, заполняя его биты справа значениями битов, вытесняемых из операнд\_2. Количество сдвигаемых бит определяется значением счетчик\_сдвигов, которое может лежать в диапазоне 0...31. Это значение может задаваться непосредственным операндом или содержаться в регистре cl. Значение операнд\_2 не изменяется.

**shrd операнд\_1,операнд\_2,счетчик\_сдвигов** — сдвиг вправо двойной точности.

Команда производит замену путем сдвига битов операнда операнд\_1 вправо, заполняя его биты слева значениями битов, вытесняемых из операнд\_2. Количество сдвигаемых бит определяется значением счетчик\_сдвигов, которое может лежать в диапазоне 0...31. Это значение может задаваться непосредственным операндом или содержаться в регистре cl. Значение операнд\_2 не изменяется.

Как мы отметили, команды shld и shrd осуществляют сдвиги до 32 разрядов, но за счет особенностей задания операндов и алгоритма работы эти команды можно использовать для работы с полями длиной до 64 бит.

# Лекция 14. Команды передачи управления

Команды программы хранятся в последовательно расположенных ячейках памяти, но редко исполняются по порядку. Многие программы содержат переходы, заставляющие микропроцессор изменять путь исполнения программы. То, какая команда программы будет выполняться следующей, микропроцессор узнает по содержимому пары регистров CS:IP.

Классификация команд переходов в персональной ЭВМ представлена на рис.



Команды микропроцессора, обеспечивающие организацию переходов в программе, можно разделить на три группы:

1. Команды безусловной передачи управления:

* команда безусловного перехода;
* вызов процедуры и возврата из процедуры;
* вызов программных прерываний и возврат из программных прерываний.

2. Команды условной передачи управления:

* команды перехода по результату команды сравнения;
* команды перехода по состоянию определённого флага;
* команды перехода по содержимому регистра CX.

3. Команды управления циклом:

* команда организации цикла со счётчиком CX;
* команда организации цикла со счётчиком CX, с возможностью досрочного выхода из цикла по дополнительному условию.

## Команда безусловного перехода

Команды безусловных переходов производят модификацию регистра IP или регистров IP и CS без предварительного анализа каких-либо условий. Все они имеют одинаковую мнемонику JMP и содержат один операнд.

**JMP [модификатор] *адрес\_перехода*.**

*Адрес перехода* может быть либо меткой, либо адресом области памяти, в которую предварительно помещен указатель перехода.

В системе команд микропроцессора существует пять команд безусловных переходов. Их различия определяются дальностью перехода и способом задания целевого адреса. Дальность перехода определяется местоположением операнда *адрес\_перехода*. Этот адрес может находиться в текущем сегменте кода или в некотором другом сегменте. В первом случае переход называется внутрисегментным или близким, а во втором случае – межсегментным или дальним.

Внутрисегментный переход предполагает, что изменяется только содержимое регистра IP. Можно выделить три варианта внутрисегментного перехода:

* прямой короткий;
* прямой;
* косвенный.

Прямой короткий внутрисегментный переход применяется, когда расстояние от команды Jmp до адреса перехода не более, чем -128 или +127 байтов. В этом случае транслятор языка формирует машинную команду безусловного перехода длиной 2 байта, 1 байт – код операции, 2-ой байт – смещение. Здесь нужно отметить одну особенность транслятора Ассемблера – он является однопроходным, иными словами, машинный код программы получается за один просмотр мнемокоманд от начала до конца. В связи с этим обстоятельством, если безусловный переход должен происходить на адрес до команды JMP, то транслятор может легко вычислить смещение. Если же переход короткий, но на метку после команды JMP, то транслятору нужно подсказать, что он должен сформировать команду безусловного короткого перехода. С этой целью в команде JMP используется модификатор SHORT PTR (полностью - SHORT POINTER или короткий указатель):

JMP SHORT PTR M1

. . . . . . не более 35-40 команд

M1:

MOV AL, 34H

Прямой внутрисегментный переход  отличается от прямого короткого тем, что длина машинной команды Jmp составляет 3 байта, в которой 2 последних байта интерпретируются как смещение. Нетрудно определить, что в этом варианте можно осуществлять переход в пределах 64 Кбайт памяти относительно следующей за JMP команды.

  ...

M1:

... ; Расстояние более 128 байт и менее 64 Кбайт

jmp M1

...

Косвенный внутрисегментный переход. В этом случае в команде перехода указывается не сам адрес перехода, а то место, где находится этот адрес. Это может быть либо регистр общего назначения, либо область памяти (слово памяти).

LEA BX, M1

JMP BX

. . . . . .

M1: MOV AL, 34H

Приведём ещё два примера косвенного внутрисегментного перехода, ориентированных на использование языка ассемблера.

 Пример [Юров,2002,с.210].

|  |
| --- |
| ...  .DATA  addr DW m1  DW m2  .CODE  ...  Cycl:  mov si,0  jmp addr[si]; Адрес перехода в слове памяти addr+(SI)  ...  mov si,2  jmp Cycl  m1:  ...  m2:  ... |

Операнд команды jmp определяет адрес перехода косвенно, после вычисления выражения addr+(SI).

## Команды условного перехода

Данные команды позволяют процессору "принять решение" о ходе выполнения программы в зависимости от определённых условий.

**Jx   <Метка перехода>**

 х  - модификатор, состоящий из одной, двух или трёх букв.

Мнемонический код этой команды всегда начинаются с буквы J от первой буквы английского слова  Jump ("перейти"). Остальные буквы могут быть следующими (они определяются в зависимости от условия):

|  |  |  |
| --- | --- | --- |
| Буква мнемокода | Условие | Тип операндов |
| Е | равно | любые |
| N | не равно | любые |
| G | больше | числа со знаком |
| L | меньше | числа со знаком |
| А | выше, в смысле “больше” | числа без знака |
| В | ниже, в смысле “меньше” | числа без знака |

Что касается операнда <Метка перехода>, то эта метка может находиться только в пределах текущего сегмента кода; межсегментной передачи управления в условных переходах не допускается. В связи с этим отпадает вопрос о модификаторе, который присутствовал в синтаксисе команд безусловного перехода. В ранних моделях микропроцессора (i8086, i80286) команды условного перехода могли осуществлять только короткие переходы - на расстояние от -128 до +127 байтов от команды условного перехода. Начиная с модели i80386, это ограничение снято, но только в пределах текущего сегмента кода.

Для того чтобы принять решение о том, куда будет передано управление командой условного перехода, предварительно должно быть сформировано условие, на основании которого и будет приниматься решение о передаче управления. Источниками такого условия могут быть:

  - команда сравнения CMP, сравнивающая значения двух операндов;

- любая команда, изменяющая состояние арифметических флагов;

- состояние регистра ECX/CX.

Формат команды CMP:

**CMP операнд1, операнд2**

Эта команда осуществляет, по сути, вычитание операнд1 - операнд2, однако результат никуда не записывает, а только устанавливает флаги в соответствие с нижеприведенной таблицей.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Сравниваемые операнды | Флаги | | | |
|  | OF | SF | ZF | CF |
| Операнды без знака  Источник < приемник | - | - | 0 | 1 |
| Источник = приемник | - | - | 1 | 0 |
| Источник > приемник | - | - | 0 | 0 |
| Операнды со знаком |  |  |  |  |
| Источник < приемник | 0/1 | 1 | 0 | - |
| Источник = приемник | 0 | 0 | 1 | - |
| Источник > приемник | 0/1 | 0 | 0 | - |

Приведем еще одну таблицу, в которой отражается действие команд условного перехода по значениям анализируемых этими командами флагов:

|  |  |  |
| --- | --- | --- |
| Мнемокод команды | Критерий перехода | Значения флагов для перехода |
| Тип операндов: любые числа | | |
| JE | Операнд\_1=операнд\_2 | ZF=1 |
| JNE | Операнд\_1<>операнд\_2 | ZF=0 |
| Тип операндов: числа со знаком | | |
| JL/JNGE | Операнд\_1<операнд\_2 | SF<>OF |
| JLE/JNG | Операнд\_1<=операнд\_2 | SF<>OF или ZF=1 |
| JG/JNLE | Операнд\_1>операнд\_2 | SF=OF и ZF=0 |
| JGE/JNL | Операнд\_1>=операнд\_2 | SF=OF |
| Тип операндов: числа без знака | | |
| JB/JNAE | Операнд\_1<операнд\_2 | CF=1 |
| JBE/JNA | Операнд\_1<=операнд\_2 | CF=1 или ZF=1 |
| JA/JNBE | Операнд\_1>операнд\_2 | CF=0 и ZF=0 |
| JAE/JNB | Операнд\_1=>операнд\_2 | CF=0 |

***Команды условного перехода без использования команды сравнения***

|  |  |  |
| --- | --- | --- |
| мнемокод | Описание | Проверяемые флаги |
| JC | Перейти при переносе (аналогично JB) | CF=1 |
| JNC | Перейти при отсутствии переноса | CF=0 |
| JO | Перейти при переполнении | OF=1 |
| JNO | Перейти при отсутствии переполнения | OF=0 |
| JP/JPE | Перейти при четности | PF=1 |
| JNP/JPO | Перейти при нечетности | PF=0 |
| JS | Перейти при наличии отрицательного знака | SF=1 |
| JNS | Перейти при отсутствии отрицательного знака | SF=0 |

***Команды, условием перехода которой является состояние регистра CX***

|  |  |  |
| --- | --- | --- |
| JCXZ | Перейти, если содержимое CX - 0 | CX=0 |

Важно отметить ограничения, свойственное команде JCXZ: в отличие от других команд условной передачи управления, эта команда может адресовать только короткие переходы - на -128 байтов или на +127 байтов от следующей за ней командой.

Если при программировании условия Вы воспользовались арифметической командой, то после неё уже не надо применять команду сравнения, поскольку дальнейшее сравнение результата будет осуществляться с 0 (если, конечно, Вы не будете сравнивать результат с другим операндом).

Рассмотрим некоторые арифметические команды, которые могут быть использованы в качестве условия: ADD, ADC, SUB, SBB, MUL, IMUL. Все эти команды вычитают операнд-источник из операнда-приёмника и в зависимости от результата устанавливают или обнуляют флаги.

Укажем  значения флагов после выполнения ADD, ADC, SUB и SBB:

CF=1, если результат не помещается в операнде-приёмнике;

PF=1, если результат имеет чётное число битов;

ZF=1, если результат равен 0;

SF=1, если результат отрицателен;

OF=1, если результат превышает диапазон допустимых значений приёмника в обратном коде.

Отметим, что содержимое SF и OF имеют смысл только при сложении и вычитании чисел со знаком.

## Организация циклов

Цикл представляет собой важную алгоритмическую структуру, без использования которой не обходится, наверное, ни одна программа.   
Организовать циклическое выполнение некоторого участка программы можно, к примеру, используя команды условной передачи управления или команду безусловного перехода jmp. При такой организации цикла все операции по его организации выполняются “вручную”. Но, учитывая важность такого алгоритмического элемента, как цикл, разработчики микропроцессора ввели в систему команд группу из трех команд, облегчающую программирование циклов. Дадим краткую характеристику этим командам: 

**LOOP метка\_перехода** - повторить цикл. Работа команды заключается в выполнении следующих действий:

* декремента регистра ecx/cx;
* сравнения регистра ecx/cx с нулем:

если (ecx/cx) > 0, то управление передается на метку перехода;

если (ecx/cx) = 0, то управление передается на следующую после loop команду.

**LOOPE/LOOPZ метка\_перехода** (Loop till cx <> 0 or Zero Flag = 0) - повторить цикл, пока cx <> 0 или **zf = 0.**

Команды loope и loopz — абсолютные синонимы, поэтому используйте ту команду, которая вам больше нравиться. Работа команд заключается в выполнении следующих действий:

* декремента регистра ecx/cx;
* сравнения регистра ecx/cx с нулем;
* анализа состояния флага нуля zf:

если (ecx/cx) > 0 и zf = 1, управление передается на метку перехода;

если (ecx/cx) = 0 или zf = 0, управление передается на следующую после loop команду.

**LOOPNE/LOOPNZ метка\_перехода** (Loop till cx <> 0 or Not Zero flag=0) - повторить цикл пока cx <> 0 или **zf = 1**.

Команды loopne и loopnz также абсолютные синонимы. Работа команд заключается в выполнении следующих действий:

* декремента регистра ecx/cx;
* сравнения регистра ecx/cx с нулем;
* анализа состояния флага нуля zf:
  + если (ecx/cx) > 0 и zf = 0, управление передается на метку перехода;
  + если (ecx/cx)=0 или zf=1, управление передается на следующую после loop команду.

Команды **loope/loopz** и **loopne/loopnz** по принципу своей работы являются взаимообратными. Они расширяют действие команды **loop** тем, что дополнительно анализируют флаг zf, что дает возможность организовать досрочный выход из цикла, используя этот флаг в качестве индикатора.

Общая особенность команд цикла в том, что они используют РОН **СХ** как счетчик числа циклов, поэтому при их использовании не забудьте до метки\_перехода послать в этот регистр нужное число!

Недостаток команд организации цикла **loop, loope/loopz** и **loopne/loopnz** в том, что они реализуют только короткие переходы (от –128 до +127 байт). Для работы с длинными циклами придется использовать команды условного перехода и команду jmp, поэтому постарайтесь освоить оба способа организации циклов.

# Лекция 15. Стек

**Стек** - это область памяти, специально выделяемая для временного хранения данных программы. Важность стека определяется тем, что для него в структуре программы предусмотрен отдельный сегмент.

Перечислим еще некоторые особенности работы со стеком:

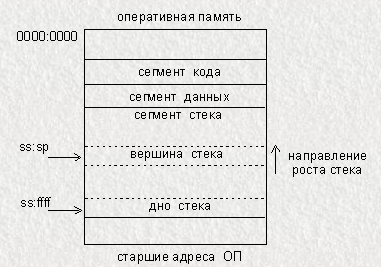
* запись и чтение данных в стеке осуществляется в соответствии с принципом LIFO (Last In First Out — “последним пришел, первым ушел”);
* по мере записи данных в стек последний растет в сторону младших адресов. Эта особенность заложена в алгоритм команд работы со стеком;
* при использовании регистров esp/sp и ebp/bp для адресации памяти ассемблер автоматически считает, что содержащиеся в нем значения представляют собой смещения относительно сегментного регистра ss.

Стек явно и неявно широко используется при программировании. Например, в стек помещается адрес возврата при вызове подпрограмм, в стеке удобно временно хранить данные из регистров, т.к. запись содержимого пары регистров в стек и извлечение их из стека осуществляется однобайтовыми (при работе с индексными регистрами двухбайтовыми) командами.

## Работа стека

Образно, стек напоминает стопку тарелок на кухне. Верхнюю тарелку взять легко, но чтобы добраться до нижней тарелки, необходимо снять все тарелки, расположенные над ней. Помещение новой порции на вершину стека называется  проталкиванием в стек  (push). Удаление порции из вершины стека, приводящее к поднятию на один уровень остальных порций, называется выталкиванием из стека (pop).

Для моделирования передвижения значений в стеке необходимо использовать регистры, которые определяют базовый адрес стека и смещение вершины стека, т.е. местоположение верхнего значения в стеке.



В памяти стек растет в направлении уменьшения адресов памяти, поэтому основание стека расположено в более высоких адресах памяти, чем  вершина стека. При этом сегментный регистр SS содержит адрес начала стекового сегмента, регистр-указатель стека SP всегда указывает на вершину стека, то есть содержит смещение, по которому в стек был занесен последний элемент. Команды работы со стеком неявно изменяют этот регистр так, чтобы он указывал всегда на последний записанный в стек элемент. Если стек пуст, то значение SP равно адресу последнего байта сегмента, выделенного под стек.

При занесении элемента в стек процессор уменьшает значение регистра SP, а затем записывает элемент по адресу новой вершины.

При извлечении данных из стека процессор копирует элемент, расположенный по адресу вершины, а затем увеличивает значение регистра указателя стека SP.

Для организации работы со стеком существуют специальные команды записи и чтения.

**PUSH источник** — запись значения источник в вершину стека.

Работа этой команды включает следующие действия:

* (sp) = (sp) – 2; значение sp уменьшается на 2;
* значение из источника записывается по адресу, указываемому парой ss:sp.

**POP назначение** — запись значения из вершины стека по месту, указанному операндом назначение. Значение при этом “снимается” с вершины стека.

Алгоритм работы команды pop противоположен алгоритму команды push:

* запись содержимого вершины стека по месту, указанному операндом назначение;
* (sp) = (sp) + 2; увеличение значения sp.

**PUSHA** — команда групповой записи в стек.

По этой команде в стек последовательно записываются регистры ax, cx, dx, bx, sp, bp, si, di. Заметим, что записывается оригинальное содержимое sp, то есть то, которое было до выдачи команды pusha.

**POPA** – команда, выполняющая обратное действие.

## Управление стеком

Принцип правильного управления стеком прост: каждая команда push в программе должна иметь соответствующую команду pop. Такое равенство обеспечивает правильность указателя стека, при этом программа может корректно записывать и восстанавливать значения. Несогласованность использования команд push и pop может привести к "зависанию" системы, т.е. привести либо к переполнению стека, либо к перекрытию областей памяти.

## *Доступ к элементам стека*

Команды PUSH и POP дают доступ только к вершине стека, но иногда необходим доступ к другим, более "низким" элементам стека. Пусть, к примеру, в стеке записано не менее трёх слов:

|  |
| --- |
| 1 |
| 2 |
| 3 |

Вершина стека→

+2

+4

Пусть далее мы хотим поместить в регистр AX копию третьего сверху элемента стека. Адрес третьего слова стека равен адресу вершины стека плюс 4. Учитывая это, устанавливаем регистр BP на вершину стека и используем выражение [BP+4] для доступа к третьему слову:

mov bp,sp

mov ax,[bp+4] или mov ax,ss:[bp+4]

# Лекция 16. Модульное программирование. Процедуры

## Концепция модульного программирования

 Концепцию модульного программирования  можно сформулировать в виде нескольких понятий и положений [Юров,2002,с.313-314]:

  (1)  функциональная декомпозиция задачи  - это разбиение большой задачи на ряд более мелких, функционально самостоятельных подзадач - модулей. Модули связаны между собой только по входным и выходным данным;

(2)  модуль - это основа концепции модульного программирования. Каждый модуль в функциональной декомпозиции представляет собой "чёрный ящик" с одним входом и одним выходом. Модульный подход позволяет безболезненно производить модернизацию программы в процессе её эксплуатации и облегчает её сопровождение. Дополнительно модульный подход позволяет разрабатывать части программ одного проекта на разных языках программирования, после чего с помощью компоновочных средств объединить их в единый загрузочный модуль;

-  реализуемые решения должны быть простыми и ясными  Если назначение модуля непонятно, то это говорит о том, что декомпозиция начальной или промежуточной задачи была проведена недостаточно качественно. В этом случае необходимо ещё раз проанализировать задачу и, возможно, провести дополнительное разбиение на подзадачи. При наличии сложных мест в проекте их нужно подробнее документировать с помощью продуманной системы комментариев. Этот процесс нужно продолжать до тех пор, пока Вы действительно не добьётесь ясного понимания назначения всех модулей задачи и их оптимального сочетания;

(3) назначение всех переменных модуля должно быть описано с помощью комментариев по мере их определения;

(4) исходный текст модуля должен иметь заголовок, в котором отражены как назначение этого модуля, так и все его внешние связи. Этот заголовок можно назвать  интерфейсной частью модуля . В этой части с использованием комментариев нужно поместить следующую информацию:

- назначение модуля;

- особенности функционирования;

- описание входных аргументов;

- описание выходных аргументов;

- использование внешних модулей и переменных;

- сведения о разработчике для защиты авторских прав;

(5) в ходе разработки программы следует предусматривать специальные блоки операций, учитывающие реакцию на возможные ошибки в данных или в действиях пользователя. Это очень важный момент, который означает то, что не должно быть тупиковых ветвей в алгоритме программы, в результате работы которых программа "виснет" и перестаёт отвечать на запросы пользователя. Любые непредусмотренные действия пользователя должны приводить к генерации ошибочной ситуации или к предупреждению о возможности возникновения такой ситуации.

Из этих положений видно, какое большое значение придаётся организации управляющих и информационных связей между структурными единицами программы (модулями), совместно решающими одну или несколько больших задач.

Применительно к языку ассемблера можно рассматривать несколько форм организации управляющих связей [Юров,2002,с.314]:

- использование  механизма макроподстановок, позволяющего изменять исходный текст программы в соответствии с некоторыми предварительно описанными параметризованными объектами;

- использование  механизма подпрограмм, написанных на языке ассемблера и структурно входящих в одну программу. В языке ассемблера такие подпрограммы называются  процедурами. В отличие от макрокоманд, взаимодействие процедур осуществляется на этапе выполнения программы;

- использование  механизма подпрограмм, написанных на разных языках программирования и соединяемых в единый модуль на этапе компоновки. Эта возможность реализуется благодаря унифицированному формату объектного модуля, однозначным соглашениям по передаче аргументов и единым схемам организации памяти на этапе выполнения.

В качестве  основных информационных связей можно выделить следующие [Юров,2002,с.314]:

  - использование общих областей памяти и общих программно-аппаратных ресурсов микропроцессора для связи модулей;

- унифицированную передачу аргументов при вызове модуля. Эту унификацию можно представлять двояко: на уровне пользователя и на уровне конкретного компилятора;

- унифицированную передачу аргументов при возврате управления из модуля.

В качестве некоторого итога приведённых выше общих рассуждений перечислим средства языка ассемблера по осуществлению функциональной декомпозиции программы:

- макросредства;

- процедуры;

- средства компилятора ассемблера в форме директив организации

оперативной памяти и её сегментации.

Наше дальнейшее обсуждение будет посвящено более глубокому изучению функциональной декомпозиции программ с использованием механизма процедур и связанных с ним средств компилятора.

## Понятие "процедура"

При написании программ обычно можно выделить одинаковые последовательности команд, часто встречающиеся в разных частях программы. Для того, чтобы многократно не переписывать такие последовательности команд, их объединяют определённым образом. В языке ассемблера имеется несколько средств, решающих проблему дублирования участков программного кода. К ним относятся [Юров,2002,с.212]:  механизм процедур,  макроассемблер,  механизм прерываний.

Сейчас мы рассмотрим механизм процедур.

**Процедура** (часто называемая  подпрограммой ) [Юров,2002,с.213] - это основная функциональная единица декомпозиции (разделения на несколько частей) некоторой задачи. Процедура представляет собой группу команд для решения конкретной подзадачи и обладает средствами получения управления из точки вызова задачи более высокого уровня и возврата управления в эту точку.

Другими словами,  процедуру можно определить как правильным образом оформленную совокупность команд, которая, будучи однократно описана, при необходимости может быть вызвана в любом месте программы.

## Описание процедуры

Для описания последовательности команд в виде процедуры в языке ассемблера используются две директивы: PROC и ENDP.

 Синтаксис описания процедуры  в программе таков:

**PROC  < Имя\_процедуры >** ←  *Заголовок процедуры*

< Команды\_и\_директивы\_языка\_ассемблера>   *Тело процедуры*

**ENDP  < Имя\_процедуры>**    ←*Конец процедуры*

Процедура может размещаться в любом месте программы, но так, чтобы на неё случайным образом не попало управление. Если процедуру просто вставить в общий поток команд, то микропроцессор будет воспринимать команды процедуры как часть этого потока и, соответственно, будет осуществлять выполнение команд процедуры. Учитывая это обстоятельство, можно использовать следующие варианты размещения процедуры в команде [Юров,2002,с.213]:

- в начале программы (до первой исполняемой команды);

- в конце (после команды, возвращающей управление операционной системе);

- промежуточный вариант - тело процедуры располагается внутри другой процедуры или основной программы. В этом случае необходимо предусмотреть обход процедуры с помощью команды безусловного перехода JMP;

- в отдельном сегменте кода. Этот вариант предполагает, что часто используемые процедуры выносятся в отдельный файл. Файл с процедурами должен быть оформлен как обычный исходный файл и подвергнут компиляции для получения объектного кода. Впоследствии этот объектный файл утилитой TLINK можно объединить с файлом, где эти процедуры используются.

## Команда вызова процедуры. Команда возврата управления

Как обратиться к процедуре? Как правило, процедуру размещают в памяти с некоторого адреса, которому в соответствие ставят некоторое символическое имя, обычно обозначающее имя процедуры, и затем с помощью команды **CALL** обращаются к этому символическому адресу, вызывая написанную процедуру.

 (1) **CALL [< Модификатор >] < Имя\_процедуры >**

Алгоритм выполнения команды вызова процедуры складывается из следующих команд микропроцессора:

- записывается побайтно в стек адрес команды, следующей за командой вызова процедуры;

- выполняется процедура с адреса, указанного в операнде команды;

- в счетчик команд записывается адрес возврата;

- выполняется безусловный переход по адресу возврата.

(2) команда возврата управления:

**RET  [< Число >]**

Команда RET считывает адрес возврата из стека и загружает его в регистры CS и IP/EIP, тем самым возвращая управление на команду, следующую в программе за командой CALL. <Число> - это необязательный параметр, обозначающий количество элементов (в байтах), удаляемых из стека при возврате из процедуры.

Для команды CALL актуальна проблема организации ближних и дальних переходов. Это видно из формата команд, где присутствует < Модификатор>. Как и в случае команды JMP, вызов процедуры командой CALL может быть [Юров,2002,с.216]:

-  внутрисегментным - процедура находится в текущем сегменте кода (имеет тип near), и в качестве возврата команда CALL сохраняет только содержимое регистра IP/EIP, что вполне достаточно для осуществления возврата;

- межсегментным - процедура находится в другом сегменте кода (имеет тип far), и для осуществления возврата команда CALL должна запомнить содержимое обоих регистров CS и IP/EIP. Очерёдность размещения их в стеке такова: сначала CS, затем IP/EIP. Если Вы используете оператор RET, то компилятор языка ассемблера "сам генерирует" правильный код.

## Организация интерфейса с процедурой

Последний и, наверное, самый важный вопрос, возникающий при работе с процедурами, - как правильно передать параметры процедуре и вернуть результат?

При программировании на языке высокого уровня программист ограничен в выборе способов передачи аргументов теми рамками, которые для него оставляет компилятор.  В языке ассемблера практически нет никаких ограничений на этот счёт, и фактически решение проблемы передачи аргументов предоставлено программисту. Поэтому существуют следующие варианты передачи аргументов в модуль (процедуру):

- через регистры;

- через общую область памяти;

- через стек;

- с помощью директив языка ассемблера EXTRN и PUBLIC.

*Передача аргументов через регистры.*

Это наиболее простой в реализации способ передачи данных, который очень популярен при небольшом объёме передаваемых данных.

Перечислим ограничения на способ передачи аргументов через регистры [Юров,2002,с.318]:

  - небольшое число доступных для пользователя регистров;

- необходимо постоянно помнить о том, какая информация и в каком регистре находится;

- ограничение размера передаваемых данных размерами регистра.

Если размер данных превышает 8, 16 или 32 бита, то передачу данных посредством регистров произвести нельзя. В этом случае передавать нужно не сами данные, а указатели на них.

*Передача аргументов через стек.*

Этот способ наиболее часто используется для передачи аргументов при вызове процедур. Суть его заключается в том, что вызывающая процедура самостоятельно заносит в стек передаваемые данные, после чего производит вызов вызываемой процедуры.

При передаче управления процедуре микропроцессор автоматически записывает в вершину стека два (для процедур типа near) или четыре (для процедур типа far) байта, которые являются адресом возврата в вызывающую программу. Если перед передачей управления процедуре командой CALL в стек были записаны переданные процедуре данные или указатели на них, то они окажутся под адресом возврата.

При рассмотрении архитектуры микропроцессора мы выяснили, что стек обслуживается тремя регистрами: SS, SP и BP. Микропроцессор автоматически работает с регистрами SS и SP в предположении, что они всегда указывают на вершину стека. По этой причине их содержимое изменять не рекомендуется.

Для осуществления произвольного доступа к данным в стеке микропроцессора имеет специальный регистр bp. Так же как и для регистра sp, использование bp автоматически предполагает работу с сегментом стека. Перед использованием этого регистра для доступа к данным стека его содержимое необходимо правильно инициализировать, что предполагает формирование в нём адреса, который бы указывал непосредственно на переданные данные. Для этого в начало процедуры рекомендуется включать дополнительный фрагмент кода. Он имеет своё название - **пролог процедуры** .

Типичный фрагмент программы, содержащий вызов процедуры с передачей аргументов через стек, может выглядеть так [Юров,2002,с.323]:

MASM

MODEL small

...

PROC Proc\_1 near ; "Близкая" процедура с n аргументами

; ***Начало пролога***

***push bp***

***mov bp,sp***

***; Конец пролога***

mov ax,[bp+4] ; Доступ к аргументу arg\_n для процедуры

mov ax,[bp+6] ; Доступ к аргументу arg\_{n-1} для процедуры

... ; Команды процедуры

; Подготовка к выходу из процедуры

***; Начало эпилога***

***mov sp,bp ;*** Восстановление SP

***pop bp ;*** Восстановление значения старого BP

; до входа в процедуру

***Ret ;*** Возврат в вызывающую программу

***; Конец эпилога***

ENDP Proc\_1

; ---------

.CODE

main Proc

mov ax,@data

mov ds,ax

...

push arg\_1 ; Запись в стек первого аргумента

push arg\_2 ; Запись в стек второго аргумента

...

push arg\_n ; Запись в стек n-го аргумента

call Proc\_1 ; Вызов процедуры Proc\_1

; Действия по очистке стека после возврата из процедуры

...

m1:

\_exit

main endp

END main

Код пролога состоит всего из двух команд. Первая команда push bp сохраняет содержимое bp в стеке с тем, чтобы исключить порчу находящегося в нём значения в вызываемой процедуре. Вторая команда пролога mov bp,sp настраивает bp на вершину стека. После этого мы можем не волноваться о том, что содержимое sp перестанет быть актуальным, и осуществлять прямой доступ к содержимому стека. Что мы и делаем. Для доступа к arg\_n достаточно сместиться от содержимого bp на 4, для arg\_{n-1} - на 6 и т.д. Но эти смещения подходят только для процедур типа near. Для far-процедур эти значения необходимо скорректировать ещё на 2, т.к. при вызове процедуры "дальнего" типа в стек записывается полный адрес - содержимое CS и IP. Поэтому для доступа к arg\_n команда будет выглядеть так: mov ax,[bp+6], а для arg\_{n-1}, соответственно, mov ax,[bp+8] и т.д.

Конец процедуры также должен быть оформлен особым образом и содержать действия, обеспечивающие корректный возврат из процедуры.

Фрагмент кода, выполняющего такие действия, имеет своё название - **эпилог процедуры**. Код эпилога должен восстановить контекст программы в точке вызова вызываемой процедуры из вызывающей программы. При этом, в частности, нужно откорректировать содержимое стека, убрав из него ставшие ненужными аргументы, передававшиеся в процедуру.

Это можно сделать несколькими способами:

- используя последовательность из n команд POP. Лучше всего это делать в вызывающей программе сразу после возврата управления из процедуры;

- откорректировать регистр указателя стека SP на величину 2n, например, командой ADD sp,2\*n, где n - количество аргументов. Это также лучше делать после возврата управления вызывающей процедуре;

- используя машинную команду RET n в качестве последней исполняемой команды в процедуре, где n - количество байтов, на которое нужно увеличить содержимое регистра ESP/SP после того, как со стека будут сняты составляющие адреса возврата. Этот способ аналогичен предыдущему, но выполняется автоматически микропроцессором.

*Передача аргументов через общую область памяти.*

Этот вариант передачи аргументов предполагает, что вызывающая и вызываемая программы "условились" использовать некоторую область памяти как общую. Компилятор предоставляет специальное средство для организации такой области: наличие атрибута комбинирования сегментов в директиве сегментации указывает компоновщику TLINK, как нужно комбинировать сегменты, имеющие одно имя. Значение COMMON означает, что все сегменты, имеющие одинаковое имя в объединяемых модулях, будут располагаться компоновщиком, начиная с одного адреса оперативной памяти. Это значит, что они будут просто перекрываться в памяти и, следовательно, совместно использовать выделенную память.

*Передача аргументов с помощью директив . EXTRN и PUBLIC.*

Приведём несколько правил оформления модулей:

(1) внешне модуль мало отличается от основной программы, только определение стекового сегмента не требуется - он определяется только в главной программе;

(2) в отдельных модулях отсутствует инициализация регистров сегмента данных и возврат управления в DOS по окончании программы;

(3) текст модуля содержит лишь набор процедур, помеченных директивами PROC и END;

(4) все процедуры, которые вы хотите экспортировать "во внешний мир", помечаются директивой PUBLIC, синтаксис которой имеет вид:

**PUBLIC <Имя\_процедуры>,[<Имя\_процедуры>]**

(5) модуль заканчивается директивой END, причём не указывается метка точки входа, она указывается только в основной программе.

Приведём макет модуля:

%TITLE "Пример модуля"

IDEAL

MODEL small

; Макроопределения

...

;------

DATASEG

...

;------

CODESEG

PUBLIC Proc\_1,Proc\_2,...,Proc\_n ; Объявлены имена модулей,

; видимых в других модулях

;----------

PROC Proc\_1

...

ENDP Proc\_1

;----------

PROC Proc\_2

...

ENDP Proc\_2

;----------

PROC Proc\_n

...

ENDP Proc\_n

END

Приведём  макет основной программы:

  %TITLE "Использование модулей"

IDEAL

MODEL small

STACK 256

DATASEG

exCode DB 0

CODESEG

EXTRN Proc\_1:PROC,Proc\_2:PROC,...,Proc\_n:PROC ; Внешние имена

Start:

mov ax,@data

mov ds,ax

call Proc\_1

call Proc\_2

...

call Proc\_n

Exit:

mov ah,04Ch

mov al,[exCode]

int 21h

END Start

Модули ассемблируются как и все остальные программы. После того, как вы получите несколько объектных файлов, их нужно скомпоновать:

**tlink  Главная\_программаМодуль 1   Модуль 2 ...  Модуль i**

В результате Вы получите исполняемый EXE файл с именем главной программы.

## Возврат результата из процедуры

В отличие от языков высокого уровня, в языке ассемблера нет отдельных понятий "процедура" и "функция". Организация возврата результата из процедуры полностью ложится на программиста.

В общем случае программист располагает следующими вариантами возврата значений из процедуры [Юров,2002,с.330]:

(1) с использованием регистров. Ограничения здесь те же, что и при передаче данных, - это небольшое количество доступных регистров и их фиксированный размер. Функции DOS используют именно этот способ. Из рассматриваемых здесь трёх вариантов данный способ является наиболее быстрым, поэтому его имеет смысл использовать для организации критичных по времени вызова процедур;

(2) с использованием общей области памяти. Этот способ удобен при возврате большого количества данных, но требует внимательности в определении областей данных и подробного документирования для устранения неоднозначностей;

(3) с использованием стека. Здесь, подобно передаче аргументов через стек, также нужно использовать регистр BP. При этом возможны следующие варианты:

- использование для возвращаемых аргументов тех же ячеек в стеке, которые применялись для передачи аргументов в процедуру. То есть предполагается замещение ставших ненужными входных аргументов выходными данными;

- предварительное помещение в стек наряду с передаваемыми аргументами фиктивных аргументов с целью резервирования места для возвращаемого значения. При использовании этого варианта процедура, конечно же, не должна пытаться очистить стек командой RET. Эту операцию придётся делать в вызывающей программе, например, командой POP.

# Лекция 18. Цепочки

Цепочечные команды также называют командами обработки строк символов. Отметим, что трактовка терминов " строка символов " и "цепочка" несколько отличаются.

**Строка символов**  (в языке ассемблера) [Юров,2002,с.230] – это последовательность байтов.

**Цепочка**  (в языке ассемблера) [Юров,2002,с.230] - это более общее название строки для случаев, когда элементы последовательности имеют размер больше байта, т.е. слово или двойное слово.

**Цепочечные команды** (по [Юров,2002,с.230]) - это команды, которые:

(1) позволяют проводить действия над блоками памяти, представляющими собой  последовательности элементов  следующего размера: байт, слово, двойное слово;

(2) кроме обработки текущего элемента цепочки, осуществляют ещё и автоматическое продвижение к следующему элементу данной цепочки.

Содержимое блоков памяти для микропроцессора не имеет никакого значения (это могут быть символы, числа и т.п.). Главное, чтобы размерность элементов совпадала с одной из вышеперечисленных, и эти элементы находились в соседних ячейках памяти.

## Синтаксис цепочечных команд

Несмотря на большое количество мнемонических названий, существует только пять цепочечных команд: MOVS, CMPS, SCAS, LODS, STOS. Остальные являются сокращенными мнемоническими названиями для этих же самых команд. Итак, команды делятся на следующие пять групп:

**1. Пересылка цепочки:**

MOVS < Адрес\_приёмника >,< Адрес\_источника>

MOVSB

MOVSW

MOVSD

**2. Сравнение цепочек:**

CMPS <Адрес\_приёмника>, <Адрес\_источника>

CMPSB

CMPSW

CMPSD

**3. Сканирование цепочки:**

SCAS  < Адрес\_приёмника >

SCASB

SCASW

SCASD

**4. Загрузка элемента из цепочки:**

LODS Адрес\_источника>

LODSB

LODSW

LODSD

**5. Сохранение элемента в цепочке .**

STOS  <Адрес\_приёмника >

STOSB

STOSW

STOSD

Для описания семантики цепочечных команд необходимо ещё остановиться на следующих важных моментах:  префиксах повторения, цепочечных индексных регистрах и направлении обработки цепочек.

## Префиксы повторения

 Префиксы повторения  предназначены для использования цепочечными командами. Без префикса повторения цепочечная команда выполняется один раз. Размещение префикса перед цепочечной командой заставляет её выполнятся в цикле. Отличия приведённых префиксов в том, на каком основании принимается решение о циклическом выполнении цепочечной команды: по содержимому регистра ECX/CX или по состоянию флага нуля ZF. Префиксы повторения имеют следующие  мнемонические обозначения  [Юров,2002,с.231-232]:

- **REP**  (от англ.  REPeat  - "повторить"). Этот префикс используется с командами пересылки и сохранения элементов цепочек (MOVS и STOS).

Префикс заставляет данные команды выполняться, пока содержимое регистра ECX/CX не станет равным 0. При этом цепочечная команда, перед которой стоит префикс, автоматически уменьшает содержимое регистра ECX/CX на единицу. Та же команда, но без префикса, этого не делает;

**- REPE  или  REPZ**  (от англ.  REPeat while Equal or Zero  - "повторять, пока равно или флаг - нуль"). Эти префиксы являются абсолютными синонимами. Они заставляют цепочечную команду выполняться до тех пор, пока содержимое регистра ECX/CX не равно 0 или флаг ZF равен 1. Благодаря возможности анализа флага ZF наиболее эффективно эти префиксы можно использовать с командами CMPS и SCAS для поиска отличающихся элементов цепочек;

- **REPNE или  REPNZ**  (от англ.  REPeat while Not Equal or Zero  -

"повторять, пока не нуль или флаг - нуль"). Эти префиксы также являются абсолютными синонимами. Они заставляют цепочечную команду выполняться до тех пор, пока содержимое регистра ECX/CX не равно 0 или флаг ZF равен 0. При невыполнении одного из этих условий работа команды прекращается. Данные префиксы можно использовать с командами CMPS и SCAS, но для поиска совпадающих элементов цепочек.

## Цепочечные  индексные регистры

Все цепочечные команды используют для выполнения своих функций определённые регистры. В отличие от команд, для которых Вы сами выбираете используемые регистры, цепочечные команды требовательнее и всегда работают с одними и теми же комбинациями регистров:

* DS:SI (индексными регистрами цепочки-источника);
* ES:DI (индексными регистрами цепочки-назначения), которые определяют смещения в данных и сегментах.

Если DS и ES адресуют один и тот же сегмент данных, то при выполнении цепочечных команд Вам не стоит беспокоится о корректной адресации сегментов памяти.

Если DS и ES адресуют различные сегменты, то Вы должны быть внимательными, указывая нужные сегменты для выполняемых операций. Кроме того, индексный регистр назначения DI определяется всегда относительно сегмента, адресуемого ES, а индексный регистр источника SI - относительно DS.

## Направление обработки цепочки

Пять цепочечных команд загружают, записывают, пересылают, сравнивают и выполняют поиск байтов и слов. Выполняя это, каждая команда также увеличивает или уменьшает значения в используемых ею регистрах:

- при операциях с байтами происходит вычитание или добавление 1 к регистру SI или DI (или обоим);

- при операциях со словами добавляется или вычитается 2.

Флаг направления DF определяет, должна ли цепочечная команда приводить к увеличению или уменьшению SI и DI:

- если DF=0, то значения индексных регистров ESI/SI и EDI/DI будут автоматически увеличиваться (операция инкремента) цепочечными командами, т.е. обработка будет осуществляться в направлении возрастания адресов;

- если DF=1, то значения индексных регистров ESI/SI и EDI/DI будут автоматически уменьшаться (операция декремента) цепочечными командами, т.е. обработка будет идти в направлении убывания адресов.

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

-  CLD  (от  Clear Direction Flag  - "флаг очистки"). Команда сбрасывает флаг направления DF в 0;

-  STD  (от  Set Direction Flag  - " флаг установки "). Команда устанавливает флаг направления DF в 1.

## Пересылка цепочек

**MOVS < адрес\_приёмника >,< адрес\_источника>**

Команда копирует байт, слово или двойное слово из цепочки, адресуемой операндом адрес\_источника, в цепочку, адресуемую операндом адрес\_приемника.

Сама по себе команда movs пересылает только один элемент, исходя из его типа, и модифицирует значения регистров esi/si и edi/di. Если перед командой написать префикс rep, то одной командой можно переслать до 64 Кбайт данных (если размер адреса в сегменте 16 бит — use16) или до 4 Гбайт данных (если размер адреса в сегменте 32 бит - use32).

 Пример.

cld ; Автоувеличение si, di

mov cx,100 ; Использование cx в качестве счетчика

repmovsb ; Переслать 100 байт

Эти три короткие команды пересылают 100 байт памяти, начиная с адреса ds:si, в область памяти, начиная с адреса es:di. Префикс повторения REP циклически выполняет команду movsb, каждый раз вычитая 1 из содержимого CX, до тех пор, пока в cx не будет 0.

## Сравнение цепочек

Для сравнения двух цепочек используется команда **CMPS (CMPSB, CMPSW)**.

**CMPS <адрес\_приёмника>, <адрес\_источника>**

* адрес\_источника определяет цепочку-источник в сегменте данных. Адрес цепочки должен быть заранее загружен в пару **ds:si*;***
* адрес\_приемника определяет цепочку-приемник. Цепочка должна находиться в дополнительном сегменте, и ее адрес должен быть заранее загружен в пару **es:di.**

Команда cmps вычитает байт или слово, расположенное по адресу es:di, из байта или слова в ds:si, сохраняя не результат, а флаги - аналогично работе cmp.

После сравнения оба регистра si и di увеличиваются или уменьшаются на 1 или 2.

Пример.

cld ; Автоувеличение di, si (сброс флага DF)

mov si,offset s1 ; ds:si указывает на первую цепочку

mov di,offset s2 ; es:di указывает на вторую цепочку

mov cx,strlength ; Запись в CX длины цепочки

repe cmpsb ; Сравнение двух цепочек

jb Less ; Переход, если s1<s2

ja Greater ; Переход, если s1>s2

je Equal ; Переход, если s1=s2

Less: ...

Greater: ...

Equal: ...

В этой последовательности подразумевается, что цепочка s1 содержится в сегменте, адресуемом ds, а цепочка s2 - в сегменте, адресуемом es. Команда repe сравнивает до тех пор, пока встретится первое несовпадение.

## Поиск в цепочках

Для поиска некоторого значения в области памяти используется команда

**SCAS  < адрес\_приёмника >**

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

При каждом повторе scas сравнивает значение байта в al или значение слова в регистре ax с данными, находящимися по адресу es:di. Затем регистр di уменьшается или увеличивается на 1 или 2.

 Пример. Поиск нуля среди 265 байт памяти.

...

cld ; Автоувеличение di (сброс флага DF)

mov di,offset Start ; Начальная точка поиска

mov cx,250 ; Запись в cx максимального значения счетчика

xor al,al ; al ← 0, то есть искомое значение

repne scasb ; Найти в памяти значения, совпадающие с al

je Yes ; Переход, если по адресу es:di-1 был найден 0

После очистки флага направления scasb будет автоматически увеличивать di на 1, в которой записывается адрес метки Start. Команда repne scasb проверяет 250 байт, до тех пор, пока значение cx не станет равным 0 и пока ZF (флаг нуля) показывает, что не было найдено совпадения.

## 

## Загрузка цепочек

**LODS Адрес\_источника>**

Команда lods загружает данные, адресуемые ds:si или es:di в регистр al для байтовых операций или в ax для операций со словами. После этого, в зависимости от установки флага направления df, увеличивается или уменьшается значение si или di. С помощью этих команд вы можете создать простой цикл для поиска определенного значения байта.

 Пример. Поиск нуля среди 265 байт памяти.

...

cld ; Автоувеличение si (сброс флага DF)

Repeat:

lods [byte ptr ds:si] ; al ← [ds:si]; si ← si+1

cmp al,0 ; al=0?

jne Repeat ; Повторять, если не равно 0

Эти команды осуществляют поиск в цепочке нулевого байта. Если такого не будет, то цикл будет повторяться бесконечно. Будьте внимательны, чтобы не внести в свои программы подобные ошибки.

 Замечание.

Автоувеличение либо уменьшение si или di на "краях" сегмента приводит к переходу на другой конец сегмента. Другими словами, если si или di равно 0FFFFh, добавление 1 "увеличивает" значение регистра до 0. И наоборот, если регистр равен 0000h, то вычитание 1 "уменьшает" его значение до 0FFFFh.

## Запись данных в цепочку. Заполнение памяти

**STOS  <Адрес\_приёмника >**

Команда stos записывает байт или слово из регистра al или ax соответственно в память по адресу es:di. Так же как и lods, в зависимости от установки флага DF и того, состоят данные из байтов или слов, stos увеличивает или уменьшает регистр di на 1 или 2. Использование в цикле команд lods и stos позволяет переместить цепочку из одного места памяти в другое.

 Пример.

...

Cld ; Автоувеличение si и di

Repeat:

lodsw ; ax ← [ds:si]; si ← si+2

cmp ax,0FFFFh ; ax=FFFFh?

je Exit ; Переход, если ax=FFFFh

stosw ; [es:di] ←ax; di ← di+2

jmp Repeat ; Повторять до окончания

Exit:

Цикл будет выполняться бесконечно, если в цепочке не будет значения FFFFh. Подумайте, почему?

Команда STOS обеспечивает простой способ заполнения памяти значениями байтов и слов. Однако будьте внимательны с ней, т.к. она может стереть целый фрагмент памяти.

# Лекция 19. Массивы

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

## Описание и инициализация массива в программе

Специальных средств описания массивов в программах ассемблера, конечно, нет. При необходимости использовать массив в программе его нужно моделировать одним из следующих способов:

1. Перечислением элементов массива в поле операндов одной из директив описания данных. При перечислении элементы разделяются запятыми.

Array DB 0,1,2,3,4,5,6,7,8,9

1. Используя оператор повторения **dup**. Такой способ определения используется для резервирования памяти с целью размещения и инициализации элементов массива.

Массив из десяти целых элементов, размер каждого из которых 1 байт

mas DB 10 DUP(?)

-----------------------------------

Массив из пяти нулевых элементов, размер каждого из которых 2 байта

mas\_0 DW 5 DUP(0)

;---------------------

Массив из пятнадцати элементов, имеющих значение 51. Размер каждого из которых 4 байта

mas\_dwords DD 15 DUP(51)

1. Используя директивы **label** и **rept**. Пара этих директив может облегчить описание больших массивов в памяти и повысить наглядность такого описания. Директива **rept** относится к макросредствам языка ассемблера и вызывает повторение указанное число раз строк, заключенных между директивой и строкой endm. Достоинство директивы **label** в том, что она не резервирует память, а лишь определяет характеристики объекта. В данном случае объект — это ячейка памяти. Используя несколько директив **label**, записанных одна за другой, можно присвоить одной и той же области памяти разные имена и разный тип, что и сделано в следующем фрагменте:

...

mas\_b label byte

mas\_w label word

rept 4

dw 0f1f0h

В результате в памяти будет создана последовательность из четырех слов **f1f0**. Эту последовательность можно трактовать как массив байт или слов в зависимости от того, какое имя области мы будем использовать в программе — mas\_b или mas\_w.

1. Использование цикла для инициализации значениями области памяти, которую можно будет впоследствии трактовать как массив.

## Доступ к элементам массива

Теперь, когда Вы знаете, как описывать массивы, следующим шагом будет изучение способов чтения и записи их значений. Например, как обратиться к пятому элементу массива? Ключом к ответу будет понимание того, что в языке ассемблера индексы массивов - обычные адреса, поэтому, независимо от типа данных, записанных в массив, индексация отдельных элементов сводится к следующим двум шагам:

1) умножить размер элемента массива на индекс массива, что составляет смещение относительно начала массива;

2) прибавить результат к базовому адресу массива.

**база+(размер элемента \* индекс)**

Этот механизм адресации элементов массива получил название  принцип индексирования . Необходимо помнить, что при реализации на машинном языке принципа индексирования, должны быть предусмотрены определенные средства запоминания и хранения обеих частей получаемого адреса: неизменного базового адреса массива и динамически меняющегося значения индекса или смещения. Базовый адрес массива обычно храниться так же, как и при использовании других способов адресации. Для хранения смещения используется один из индексных регистров bx, di, si - все они используют один сегментный регистр ds.

Давайте еще раз обратимся к описанию массива. К примеру, в программе статически определена последовательность данных:

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

Пусть эта последовательность чисел трактуется как одномерный массив. Размерность каждого элемента определяется директивой **dw**, то есть она равна **2** байта. Чтобы получить доступ к третьему элементу, нужно к адресу массива прибавить **6**. Нумерация элементов массива в ассемблере начинается с нуля. То есть в нашем случае речь, фактически, идет о **4**-м элементе массива — 3, но об этом знает только программист; микропроцессору в данном случае все равно — ему нужен только адрес.

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

* индексная адресация со смещением — режим адресации, при котором эффективный адрес формируется из двух компонентов:
  + постоянного (базового) — указанием прямого адреса массива в виде имени идентификатора, обозначающего начало массива;
  + переменного (индексного) — указанием имени индексного регистра.   
    К примеру:

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

...

mov si,4 ;поместить 3-й элемент массива mas в регистр ax:

mov ax, mas[si]

* базовая индексная адресация со смещением — режим адресации, при котором эффективный адрес формируется максимум из трех компонентов:
  + постоянного (необязательный компонент), в качестве которой может выступать прямой адрес массива в виде имени идентификатора, обозначающего начало массива, или непосредственное значение;
  + переменного (базового) — указанием имени базового регистра;
  + переменного (индексного) — указанием имени индексного регистра.

Этот вид адресации удобно использовать при обработке двухмерных массивов.

## Двухмерные массивы

С представлением одномерных массивов в программе на ассемблере и организацией их обработки все достаточно просто. А как быть если программа должна обрабатывать двухмерный массив? Все проблемы возникают по-прежнему из-за того, что специальных средств для описания такого типа данных в ассемблере нет. Двухмерный массив нужно моделировать. Непосредственно моделирование обработки массива производится в сегменте кода, где программист, описывая алгоритм обработки ассемблеру, определяет, что некоторую область памяти необходимо трактовать как двухмерный массив.   
При этом вы вольны в выборе того, как понимать расположение элементов двухмерного массива в памяти: по строкам или по столбцам.

Если последовательность однотипных элементов в памяти трактуется как двухмерный массив, расположенный по строкам, то адрес элемента **(i, j)** вычисляется по формуле

**(база + количество\_элементов\_в\_строке \* размер\_элемента \* i+j)**

Здесь **i = 0...n–1** указывает номер строки, а **j = 0...m–1** указывает номер столбца.

Например, пусть имеется массив чисел (размером в 1 байт) mas(i, j) с размерностью 4 на 4 **(i= 0...3, j = 0...3)**:

23 04 05 67

05 06 07 99

67 08 09 23

87 09 00 08

В памяти элементы этого массива будут расположены в следующей последовательности:

**23 04 05 67 05 06 07 99 67 08 09 23 87 09 00 08**

Если мы хотим трактовать эту последовательность как двухмерный массив, приведенный выше, и извлечь, например, элемент   
**mas(2, 3) = 23**, то проведя нехитрый подсчет, убедимся в правильности наших рассуждений:

**Эффективный адрес mas(2, 3) = mas + 4 \* 1 \* 2 + 3 = mas + 11**

Организовать адресацию двухмерного массива логично, используя рассмотренную нами ранее базово-индексную адресацию. При этом возможны два основных варианта выбора компонентов для формирования эффективного адреса:

* сочетание прямого адреса, как базового компонента адреса, и двух индексных регистров для хранения индексов:

mov ax,mas[bx][si]

* сочетание двух индексных регистров, один из которых является и базовым и индексным одновременно, а другой — только индексным:

mov ax,[bx][si]

Вместо регистра bx возможно использовать регистр bp, соответственно, вместо регистра si возможно использовать регистр di.

Пример. Пусть у нас есть двухмерный байтовый массив A размером 10x20:

label a byte

rept 10

db 20 dup(?)

endm ; a[0..9,0..19]

 Нам требуется записать в регистр al количество таких строк этой матрицы, в которых начальный элемент строки встречается ещё раз.

 Решение.

При расположении элементов матрицы в памяти по строкам (первые 20 байтов - начальная строка матрицы, следующие 20 байтов – вторая строка и т. д.) адрес элемента A[i,j] равен A+20\*i+j. Для хранения величины 20\*i отведём регистр bx, а для хранения j - регистр si. Тогда A[bx] - это начальный адрес i-ой строки матрицы, а A[bx][si] - адрес j-ого элемента этой строки.

...

mov al,0 ; Количество искомых строк

;Внешний цикл (по строкам)

mov cx,10 ; Счётчик внешнего цикла

mov bx,0 ; Смещение от A до начала строки

Repeat1:

mov ah,[A+bx] ; ah - начальный элемент строки

mov dx,cx ; Сохранить счётчик cx внешнего цикла

;Внутренний цикл (по столбцам)

mov cx,19 ; Счётчик внутреннего цикла

mov si,0 ; Индекс элемента внутри строки

Repeat2:

inc si ; j:=j+1

cmp [A+bx+si],ah ; A[i,j]=ah?

loopne Repeat2 ; Цикл, пока A[i,j]<>ah

jne No ; Переход, если ah не повторился

inc al ; Учёт строки, если ah повторился

;Конец внутреннего цикла

No:

mov cx,dx ; Восстановить счётчик внешнего цикла

add bx,20 ; На начало следующей строки

loop Repeat1 ; Выполнение цикла 10 раз

# Лекция 20. Структуры

Рассмотренные нами выше массивы представляют собой совокупность однотипных элементов. Но часто в приложениях возникает необходимость рассматривать некоторую совокупность данных разного типа как некоторый единый тип.

Это очень актуально, например, для программ баз данных, где необходимо связывать совокупность данных разного типа с одним объектом.   
Такой объект обычно описывается с помощью специального типа данных — **структуры.**   
С целью повысить удобство использования языка ассемблера в него также был введен такой тип данных.

Структура — это тип данных, состоящий из фиксированного числа элементов разного типа.

Для использования структур в программе необходимо выполнить три действия:

[Задать](D:\\Мамино\\Асемблер\\Структуры\\Юров\\guide\\Text\\Data.htm" \l "ЗадатьШаблон#ЗадатьШаблон) **[шаблон структуры](D:\\Мамино\\Асемблер\\Структуры\\Юров\\guide\\Text\\Data.htm" \l "ЗадатьШаблон#ЗадатьШаблон)**.

По смыслу это означает определение нового типа данных, который впоследствии можно использовать для определения переменных этого типа.

[Определить **экземпляр структуры**](file:///D:\Мамино\Асемблер\Структуры\Юров\guide\Text\Data.htm#ОпределитьЭкземпляр#ОпределитьЭкземпляр)**.**

Этот этап подразумевает инициализацию конкретной переменной заранее определенной (с помощью шаблона) структурой.

Организовать [**обращение к элементам структуры**](file:///D:\Мамино\Асемблер\Структуры\Юров\guide\Text\Data.htm#РаботаСоСтруктурой#РаботаСоСтруктурой)**.**

Описать структуру в программе означает лишь указать ее схему или шаблон; память при этом не выделяется.

Определить структуру — значит, дать указание транслятору выделить память и присвоить этой области памяти символическое имя.

Описать структуру в программе можно только один раз, а определить — любое количество раз.

## Описание шаблона структуры

Описание шаблона структуры имеет следующий синтаксис:

**STRUC** *имя\_структуры*

*<описание полей>*

**ENDS** *имя\_структуры*

Пример.

...

STRUC Date

day db 1 ; Поле 'День' - начально установлено в 1

month db ? ; Поле 'Месяц' - значение не установлено

year dw 1991 ; Поле 'Год' - начально установлено в 1991

ENDS Date

Местоположение шаблона в программе может быть произвольным, но, следуя логике работы однопроходного транслятора, он должен быть расположен до того места, где определяется переменная с типом данной структуры. То есть при описании в сегменте данных переменной с типом некоторой структуры ее шаблон необходимо поместить в

## Определение данных с типом структуры

Для использования описанной с помощью шаблона структуры в программе необходимо определить переменную с типом данной структуры. Для этого используется следующая синтаксическая конструкция:

**[имя переменной] имя\_структуры <[список значений]>**

Здесь:

* имя переменной — идентификатор переменной данного структурного типа.   
  Задание имени переменной необязательно. Если его не указать, будет просто выделена область памяти размером в сумму длин всех элементов структуры.
* список значений — заключенный в угловые скобки список начальных значений элементов структуры, разделенных запятыми.   
  Его задание также необязательно.   
  Если список указан не полностью, то все поля структуры для данной переменной инициализируются значениями из шаблона, если таковые заданы.   
  Допускается инициализация отдельных полей, но в этом случае пропущенные поля должны отделяться запятыми. Пропущенные поля будут инициализированы значениями из шаблона структуры. Если при определении новой переменной с типом данной структуры мы согласны со всеми значениями полей в ее шаблоне (то есть заданными по умолчанию), то нужно просто написать угловые скобки.   
  К примеру:

Dt1 Date <6,6,?> ; 6 6 1991

Dt2 Date <,,1998> ; 1 ? 1998

Dt3 Date <,,> ; 1 ? 1991

## Ссылки на поля структур

Описав тип структуры и переменные этого типа, мы получаем право работать с ними. Как единое целое структуры используются редко, обычно они обрабатываются по полям. Чтобы сослаться на поле структуры, надо использовать конструкцию вида:

***Имя\_переменной. имя\_поля***

Такая конструкция обозначает ту ячейку памяти, которую занимает указанное поле указанной переменной. Встречая эту конструкцию, ассемблер заменяет её на адрес данной ячейки. При этом тип (размер) этого адреса считается равный типу (размеру) данного поля, например, тип Dt1.year - это слово.

## Использование структурированных переменных

Использовать поля структур не намного сложнее, чем простые переменные. Допускаются всё те же режимы адресации. Для задания нового значения поля day можно использовать такую команду:

mov [Dt1.day], 5

Можно также загружать значения полей в регистры:

mov ax, [Dt2.year]

Вы можете складывать, вычитать, считывать, записывать, логически объединять поля и регистры.

 Пример.

  inc [Dt1.day] ; Прибавить 1 к полю 'День'

add [Dt1.year], cx ; Прибавить cx к полю 'Год'

cmp [Dt2.month], 8 ; 'Месяц'=8?

# Лекция 21. Расширенные возможности современных микропроцессоров

## Архитектурные особенности

Операционная система MS-DOS, язык ассемблера МП 86 и методы программирования микропроцессоров корпорации Intel разрабатывались применительно к 16-разрядному процессору 8086 и тому режиму, который впоследствии получил название реального. Появление процессора 80386 знаменовало собой начато нового этапа в развитии операционных систем и прикладного программирования - этапа многозадачных графических операционных систем защищенного режима типа Windows и 32-разрядных прикладных программ. При этом все архитектурные средства 86-го процессора входят в состав любого современного процессора, который, таким образом, можно условно разделить на две части - МП 86 и дополнительные средства, обеспечивающие защищенный режим, 32-разрядную адресацию и прочее.

Из этих дополнительных средств можно выделить те, которые обеспечивают защищенный режим, и в реальном режиме не используются (во всяком случае, явным образом; в действительности, процессор, даже работая в реальном режиме, использует по крайней мере некоторые из этих средств). Сюда, например, относятся регистры таблиц дескрипторов, регистры тестирования и отладки, привилегированные команды защищенного режима, система страничного отображения адресов и др.

С другой стороны, часть новых свойств современных процессоров можно использовать и в реальном режиме, выполняя программы под управлением MS-DOS. Сюда, прежде всего, относится использование 32-битовых операндов, некоторых новых команд процессора и расширенных возможностей старых команд.

**Регистры общего назначения и регистры-указатели** отличаются от аналогичных регистров МП 86 тем, что они являются 32-разрядными. К обозначениям регистров добавлена приставка E (Extended - расширенный): EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP. Возможно обращение к младшим 16 разрядам расширенных регистров по именам без приставки, например, AX, BX, а также отдельно к младшим и старшим байтам, например, AL, AH.

Однако старшие половины 32-разрядных регистров не имеют мнемонических обозначений и непосредственно недоступны. Для того, чтобы прочитать, например, содержимое старшей половины регистра ЕАХ (биты 31...16) придется сдвинуть все содержимое ЕАХ на 16 бит вправо (в регистр АХ) и прочитать затем содержимое АХ.

Все регистры общего назначения и указатели программист может использовать по своему усмотрению для временного хранения адресов и данных размером от байта до двойного слова. Так, например, возможно использование следующих команд:

mov ЕАХ,0FFFFFFFFh ;Работа с двойным словом (32 бит)

mov AX,0FFFFh ;Работа со словом (16 бит)

mov AL, 0FFh ;Работа с байтом (8 бит)

Все **сегментные регистры**, как и в МП 86, являются 16-разрядными. В их состав включено еще два регистра - FS и GS, которые могут использоваться для хранения сегментных адресов двух дополнительных сегментов данных. Таким образом, при использовании расширенных возможностей современных процессоров программе одновременно доступны четыре сегмента данных, а не два, как в МП 86.

**Регистр указателя команд** также является 32-разрядным и обычно при описании процессора его называют EIP. Младшие шестнадцать разрядов этого регистра соответствуют регистру IP процессора МП 86. Весь регистр EIP используется только в 32-разрядных приложениях; в 16-разрядных программах адреса могут быть только 16-разрядными и, соответственно, для адресации в программном сегменте используется младшая половина регистра EIP.

**Регистр флагов** принято называть EFLAGS (от extended flags, расширенные флаги). Хотя он имеет длину 32 бит, только младшие 18 бит (да и то не все) содержат значащую информацию. Дополнительно к шести флагам состояния (CF, PF, AF, ZF, SF и OF) и трем флагам управления состоянием процессора (TF, IF и DF), назначение которых было описано ранее, он включает новые флаги задачи, рестарта и виртуального режима, а также двухбайтовое поле привилегий ввода-вывода. Все эти биты используются только в защищенном режиме и здесь рассматриваться не будут.

## Дополнительные режимы адресации

Режимы адресации 32-разрядных процессоров предназначены для 32-разрядных приложений, в которых сегменты данных или стека (как, впрочем, и сегменты команд) могут иметь размеры до 232 = 4 Гбайт. Однако в реальном режиме размер любого сегмента ограничивается величиной 216 = 64 Кбайт, и 32-битовые смещения не имеют смысла. С другой стороны, ничто не мешает нам использовать для образования 16-битового смещения 32-разрядные регистры (ЕВХ, ESI и проч.), если, конечно, их реальное содержимое не будет превышать величины FFFFh.

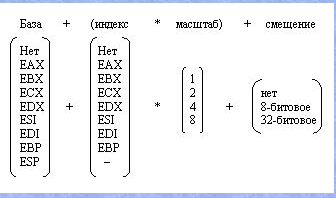
Указание в качестве операндов команд 32-разрядных регистров позволяет использовать дополнительные возможности 32-разрядных процессоров по части адресации памяти, что в некоторых случаях может оказаться полезным.

В отличие от МП 86, где базовыми регистрами могут быть только ВХ и ВР, а индексными только SI и DI, 32-разрядные процессоры допускают использование в качестве и базовых, и индексных практически всех регистров общего назначения. Таким образом, вполне законна команда вида

mov ЕАХ,[ЕСХ][EDX]

Второе отличие заключается в возможности масштабирования содержимого индексного регистра, т.е. умножения его на заданный в команде коэффициент, который может принимать значения 1, 2, 4 или 8. Пример такой адресации: ***inc word ptr [ЕАХ] [ЕСХ\*2]***

Режимы косвенной адресации памяти, предоставляемые 32-разрядными процессорами при использовании 32-разрядных регистров, изображены на рис.1   
Из рисунка видно, что в качестве базового можно использовать все регистры общего назначения, включая даже указатель стека ESP. При этом, если в качестве базового выступает один из регистров ESP или ЕВР, то, по умолчанию, адресация осуществляется через сегментный регистр SS, хотя возможна замена сегмента. Во всех остальных случаях адресация по умолчанию осуществляется через сегментный регистр DS. Использование регистра ЕВР в качестве индексного не адресует нас к стеку: адресация по прежнему осуществляется с помощью регистра DS.



Рис**.**1Режимы косвенной адресации с использованием 32-разрядных регистров.

Прочерк во второй колонке подчеркивает, что регистр ESP нельзя использовать в качестве индексного. Это не означает, что ESP нельзя указывать в качестве второго операнда:

mov ЕАХ,[ЕСХ][ESP]

Недопустима только конструкция, в которой содержимое ESP умножается на масштабирующий множитель:

mov ЕАХ,[ЕСХ][ESP\*8]

Таким образом, понятия базовой и индексной адресации в 32-разрядных процессорах несколько размываются.

Если регистр указывается с масштабирующим множителем, то это, конечно, индексная адресация. Если же множитель отсутствует, то адресацию и через ЕВХ, и через ESI с равным успехом можно отнести как к базовой, так и к индексной.

## Использование средств 32-разрядных процессоров в программировании

Как уже отмечалось, при разработке 16-разрядных программ реального режима, предназначенных для выполнения под управлением операционной системы MS-DOS, вполне допустимо использование ряда дополнительных возможностей 32-разрядных процессоров. В реальном режиме можно использовать:

* 32-разрядные операнды;
* дополнительные команды и расширенные возможности команд МП 86;
* дополнительные режимы адресации;
* четыре сегментных регистра для адресации данных вместо двух.

Для того, чтобы транслятор распознавал все эти средства, необходимо начать программу **с директивы .586 (или, при желании, .486 или .386)** и указать при этом для сегментов команд и данных описатель **use 16**, чтобы программа осталась 16-разрядной.

Следует заметить, что возможности использования в программах реального режима дополнительных средств 32-разрядных процессоров, хотя и кажутся привлекательными, в действительности весьма ограничены.

Новых команд не так уж много, и они не имеют принципиального характера;

32-разрядные данные используются в прикладных программах относительно редко (если не касаться вычислительных программ, содержащих действительные числа, но такие программы редко пишут на языке ассемблера);

расширенные возможности адресации в полной мере проявляются лишь в 32-разрядных программах, не работающих в DOS.

Тем не менее, в каких-то случаях привлечение средств 32-разрядных процессоров может оказаться полезным и в 16-разрядных программах, и мы приведем несколько примеров их использования.

Среди системных данных DOS и BIOS есть данные, требующие для своего размещения 2 слов.

Рассмотрим пример упорядочивания массива 32-разрядных чисел в убывающем порядке методом пузырьковой сортировки. В приведенном алгоритме используются расширенные возможности адресации 32-разрядных процессоров.

**Пример.** Пузырьковая сортировка

.586  
assume CS:code,DS:data  
code segment use16  
main proc  
mov AX, data ;Настроим DS  
mov DS,AX ;на сегмент данных  
mov ESI,offset list ;ESI-> начало массива  
mov ECX,1000 ;Число элементов в массиве  
start: mov EDX, 0 ;Индекс сравниваемой пары  
sort: cmp EDX,ECX ;Индекс пары дошел до  
jge stop ;индекса массива? К следующей паре  
mov EAX,[ESI+EDX\*4+4];Второй элемент пары  
cmp [ESI+EDX\*4],EAX ;Сравним с предыдущим  
jge noswap ;Если первый больше, то хорошо  
xchg [ESI+EDXM] , EAX ;Первый меньше. Обменять  
mov [ESI+EDXM + 4],EAX ;первый на второй  
noswap: inc EDX ;Увеличим индекс пары  
jmp sort ;И на сравнение  
stop: loop start ;Цикл по всем элементам  
mov AX,4C00h  
int 21h  
main endp  
code ends  
data segment  
list label ;Имя тестового массива  
nmb=0 ;Заполним массив на этапе  
rept 1000 /трансляции числами от 0  
ddnmb /до 990  
nmb=nmb+10 /через 10  
endm  
data ends  
stk segment stack  
dw 128 dup (0)  
stk ends  
end main

Алгоритм пузырьковой сортировки предусматривает выполнение двух вложенных циклов. Во внутреннем цикле сравниваются пары элементов. Первый элемент берется по адресу [ESI + EDX \* 4], второй - по следующему адресу [ESI + EDX \* 4 + 4]. Если второй элемент больше первого, происходит обмен значений этих элементов, и элемент с меньшим значением "всплывает" на одно место выше (т.е. перемещается по большему адресу). После этого увеличивается индекс пары и выполняется сравнение второго элемента со следующим. Если оказывается, что следующий элемент больше предыдущего, они меняются местами. В результате элемент с самым маленьким значением всплывает на самый верх списка.

Внутренний цикл, пройдясь по всем парам, повторяется сначала, обеспечивая всплывание следующего по величине элемента. Каждый следующий проход внутреннего цикла требует на 1 меньше шагов, чем предыдущий. После завершения упорядочивания элементы выстраиваются по возрастающим адресам в порядке уменьшения их значений.

В примере тестовый массив данных образован из возрастающих (на 10) чисел от 0 до 990. В результате упорядочивания они должны расположиться в обратном порядке, от больших к меньшим. В примере не предусмотрены средства вывода на экран элементов массива, поэтому его изучение следует проводить в отладчике, наблюдая всплывание каждого элемента.

Как уже отмечалось, в 32-разрядных процессорах увеличено до 4 число сегментных регистров данных. Это дает возможность совместной работы с четырьмя сегментами данных (общим объемом до 256 Кбайт) без перенастройки сегментных регистров. Структура такого рода программы может выглядеть следующим образом:

.586  
datal segment use16  
first dw 7000h dup(')  
datal ends  
data2 segment use16  
second dw 7000h dup (')  
data2 ends  
data3 segment use16  
third dw 7000h dup (')  
data3 ends  
data4 segment use16  
forth dw 7000h dup (')  
data4 ends  
 code segment use16  
 assume DS:datal,ES:data2,FS:data3,GS:data4  
main proc  
 ;Настроим все 4 сегментных регистра на базовые адреса  
 ; соответствующих сегментов   
mov AX,datal ;DS->datal  
mov word ptr[BX],1111h ;Обращение через DS по умолчанию   
 ;Обращение к разным сегментам с явным указанием   
 ;требуемого сегментного регистра (замена сегмента)  
mov word ptr ES:[BX],2222h  
mov word ptr FS:[BX],3333h  
mov word ptr GS:[BX],4444h  
 ;Обращение по именам полей данных разных сегментов

; с учетом действия директивы assume  
mov first,1 ;Запись в сегмент datal  
mov second,2 ;Запись в сегмент data2  
mov third,3 ;Запись в сегмент data3  
mov fourth,4 ;Запись в сегмент data4   
; Перенос данных из сегмента в сегмент  
push first  
pop second+2  
push third  
pop fourth+2  
...  
main endp  
code ends

В программе объявлены 4 сегмента данных с именами datal, data2, data3 и data4, содержащие массивы 16-разрядных данных с именами first, second, third и fourth. Длина каждого массива составляет 56 Кбайт, и, таким образом, общий объем данных, доступных программе в любой момент, составляет более 200 Кбайт. Сегменты данных описаны до сегмента команд, что в данном случае имеет значение. В сегменте команд с помощью директивы assume указано соответствие каждому из сегментов своего сегментного регистра (DS, ES, FS и GS). Это даст нам возможность обращаться по именам полей сегментов без явного указания соответствующих этим сегментам сегментных регистров.

Программа начинается с обычной практически для всех программ процедуры настройки всех сегментных регистров. Стоит еще раз повторить, что директива assume лишь обеспечивает правильную трансляцию программы, но не инициализирует сегментные регистры. "Правильная трансляция" в данном случае заключается в том, что при обработке команд, в которых упоминаются имена данных того или иного сегмента, ассемблер автоматически предваряет эти команды, префиксом замены сегмента, выбирая для замены сегментный регистр, указанный в директиве assume для данного сегмента. Так, команда

mov first, I

преобразуется в последовательность кодов (по листингу' трансляции)

С7 06 0000r 0001

где С7 06 - это код команды mov в случае прямой адресации памяти и использования непосредственного операнда, 0000г - смещение адресуемой ячейки, а 0001 - непосредственный операнд (все числа, разумеется, шестнадцатеричные). Здесь нет префикса замены сегмента, потому что адресуется сегмент, которому соответствует регистр DS, используемый процессором по умолчанию. Однако команды с обращением к другим сегментам транслируются с включением в их коды соответствующих пре фиксов, несмотря на то, что в исходных предложениях не указаны сегментные регистры, а содержатся только ссылки на (уникальные) имена ячеек тех или иных сегментов:

mov second, 2 ; Код команды 26: С7 06 0000r 0002

mov third, 3 ;Код команды 64: С7 06 0000r 0003

mov fourth, 4 ; Код команды 65: С7 06 0000r 0004

Настроив сегментные регистры, мы можем обращаться к полям данных всех четырех сегментов с использованием любых способов адресации. В приведенном фрагменте в регистр ВХ помещается смещение последней ячейки любого из массивов, после чего с помощью косвенной базовой адресации в последние слова всех четырех массивов записываются произвольные числа 1111h, 2222h, 3333h и 4444h. Во всех случаях требуется описатель word ptr, так как по виду команды ассемблер не может определить, хотим ли мы занести в память байт, слово или двойное слово. При обращении к сегментам, адресуемых не через DS, необходимо явное указание сегментного регистра (которое будет преобразовано в код префикса замены сегмента), потому что по виду команды с адресацией через регистры транслятор не может определить, к какому сегменту происходит обращение.

Проще обстоит дело, если в команде указаны имена ячеек сегментов. В этом случае, как уже говорилось выше, транслятор автоматически включает в код команды требуемый префикс замены сегмента.

Наконец, в конце программы приведены строки пересылки данных из сегмента в сегмент через стек. Они убедительны в том отношении, что в четырех последовательных командах производится обращение к четырем различным сегментам программы без перенастройки сегментных регистров, которую пришлось бы выполнить, если бы мы ограничились возможностями МП 86.

# Лабораторный практикум

# Лабораторная работа №1. Этапы создания программы на ассемблере. Отладчик Turbo Debugger

Разработка программы на языке ассемблера включает несколько этапов:

1. подготовка исходного текста программы;
2. создание объектного модуля;
3. создание загрузочного модуля;
4. отладка программы.
5. **Подготовка исходного текста** программы выполняется с помощью любого текстового редактора. Файл с исходным текстом программы должен иметь расширение \***.asm**
6. Следующий шаг – **трансляция программы**, т.е. преобразование строк исходного языка в коды машинных команд. Эта операция выполняется с помощью транслятора с языка ассемблера (программы пакета TASM):

**tasm.exe  Имя\_исходного\_файла, Имя\_файла\_листинга**

На этом шаге формируются:

  - объектный модуль (**файл с расширением .obj**), который включает в себя представление исходной программы в машинных кодах и некоторую другую информацию, необходимую для отладки и компоновки его с другими объектными модулями;

- файл листинга (**файл с расширением .lst**).  Листинг - очень важный документ, содержащий расширенную информацию о коде ассемблера исходной программы. Для каждой команды ассемблера указываются её машинный (объектный) код и смещение в кодовом сегменте. Кроме того, в конце листинга TASM формирует таблицы, которые содержат информацию о метках и сегментах, используемых в программе. Если есть ошибки или сомнительные участки кода, то TASM включает в конец листинга сообщения о них. Кроме того, что очень удобно, сообщения об ошибках включаются в текст листинга непосредственно после ошибочной строки.

Если Ваша программа содержит ошибки, то компилятор выдаст в листинге строки сообщений, начинающиеся словами "Error" и "Warning".

Наличие строки с "Error" говорит о том, что у Вас в программе есть недопустимые (с точки зрения синтаксиса) комбинации символов.

Логика работы программы для компилятора не имеет никакого значения. Вы можете написать абсолютную чушь, но если она будет синтаксически правильна, компилятор поспешит Вас обрадовать, сообщив, что всё хорошо. Наличие строки "Warning" означает, что конструкция синтаксически правильна, но не соответствует некоторым соглашениям языка, и это может послужить источником последующих ошибок.

Дальнейшие Ваши действия зависят от характера ошибки. По мере накопления опыта ошибки будут происходить, скорее всего, в результате простой описки. Пока же Ваши действия будут заключаться в выяснении того, насколько правильно написана та или иная синтаксическая конструкция. Исправив несколько первых ошибок, перекомпилируйте программу и приступайте к устранению следующих ошибок. Возможно, что этого делать не придётся, т.к. после исправления одной ошибки могут исчезнуть и последующие (так называемые наведённые ошибки).

О нормальном окончании процесса компиляции можно судить по сообщению TURBO Assembler, в котором отсутствуют строки с сообщениями об ошибках и предупреждениях.

1. После того как мы получили объектный модуль, можно приступать к следующему шагу - созданию исполняемого (загрузочного) модуля (другими словами, осуществить  **компоновку** программы) при помощи командной строки:

**tlink.exe /v  Имя\_объектного\_файла**

Опция /v указывает на необходимость сохранения отладочной информации в исполняемом файле.

Главная цель этого шага - преобразовать код и данные в объектных файлах в их  перемещаемое выполняемое отображение. Чтобы понять, в чём здесь суть, нужно разобраться, зачем вообще разделяют процесс создания исполняемого модуля на два шага - компиляцию и компоновку.

Это сделано намеренно для того, чтобы можно было объединять вместе несколько модулей (написанных на одном или нескольких языках). Формат объектного файла позволяет, при определённых условиях, объединять несколько отдельно откомпилированных исходных модулей в один модуль. Результатом работы компоновщика является создание загрузочного файла (с **расширением .exe**). После этого операционная система может загрузить такой файл в память и выполнить его.

1. следующим обязательным этапом процесса разработки программы является  процесс **отладки.**

**Отладка программы** (англ. debugging) - это систематический процесс испытания работы программы и исправления обнаруживаемых при этом ошибок. Отладка состоит из многократного исполнения программы для специально подобранных  тестовых наборов исходных данных ( тестов). Тестовые наборы составляются таким образом, чтобы поведение программы при исполнении на этих наборах, в том числе результат исполнения, было бы заранее известно, при этом процесс исполнения затрагивал бы все команды программы.

Вполне возможно, что результаты тестирования Вас не удовлетворят. В этом случае придётся вносить поправки в код программы, т.е. возвращаться к первому шагу процесса разработки.

Отладку программы будем осуществлять с помощью отладчика Turbo Debager.

Текстовый редактор

Трансляция программы tasm.exe

Компановка программы tlink.exe

Отладка программы td.exe

**Знакомство с программой-отладчиком Turbo Debugger. Выполнение**

**простейших команд микропроцессора в среде Turbo Debugg**

**Понятие отладки. Назначение программ-отладчиков**

*Отладка*(debugging) — один из важнейших этапов разработки программного обеспечения (английский термин bug означает "ошибка в программе"). В процессе отладки путем детального анализа в компьютерных программах выявляются и устраняются возможные логические ошибки, которые не обнаруживаются на стадии компиляции.

*Отладчики*(debugger) — это вспомогательные программы (утилиты), включаемые в набор инструментальных средств программиста для выполнения отладки других программ. Отладчики предоставляют программисту возможность выполнять программу по шагам, следить за изменениями данных и проверять выполнение условий. В зависимости от уровня языка, которым оперирует отладчик, можно выделить два их типа.

*Отладчики исходного кода* дают программисту возможность видеть текст программы на языке высокого уровня (например, Си), проверять значения отдельных переменных и агрегатов данных (таких, как массивы), используя их имена.

*Отладчики машинного уровня* отслеживают реально выполняемые машинные команды, отображаемые в виде команд ассемблера. Они позволяют также просматривать содержимое ячеек памяти и регистров микропроцессора.

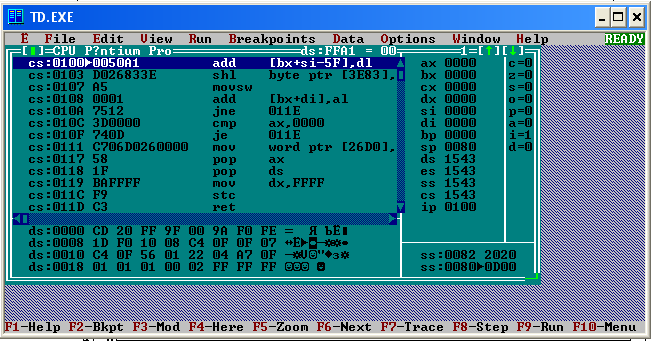
Отладчики, интегрированные в среду разработки пакетов программ, например, Borland C++, Borland Pascal, относится к первому типу.

Рассматриваемый здесь Turbo Debugger относится к отладчикам второго типа.

***Основные режимы работы отладчика Turbo Debugger***

Структура экрана программы Turbo Debugger

При запуске Turbo Debugger на экране появляется его основное меню и рабочее окно рис.1.Это окно представляет собой графическое изображение программной модели микропроцессора i8086.

****

Рабочее окно состоит из следующих четырёх окон:

(1)  окно листинга  или  дизассемблера  (центральное, самое большое окно). В нём отображается некоторый участок памяти компьютера с разбивкой на  числовые  и  мнемонические машинные коды  - команды процессора. Причём слева показывается логический адрес, по которому располагается эта команда в памяти компьютера.

 (2)  окно дампа  (располагается в нижней части экрана). Это окно составлено из:

- окна, в котором изображается участок памяти в шестнадцатеричном формате (данные из памяти);

- окна, в котором изображается информация в том виде, в котором мы можем видеть её на экране (данные из видеопамяти);

(3)  окно регистров  (располагается справа от окна листинга). Здесь показаны значения 16-разрядных  регистров  и состояния  флагов. Значения регистров представляются в шестнадцатеричном виде.

(4)  окно стека  (располагается справа от окна дампа под окном регистров), в котором отображается участок памяти  стек  в шестнадцатеричном формате.

Команды работы с Turbo Debugger

|  |  |  |
| --- | --- | --- |
| Команды | Горячие клавиши | Описание |
| Переключение между окнами | Tab  Shift+Tab | По часовой стрелке  Против часовой стрелки |
| Размер окна | F5 | Увеличение/уменьшение  размеров окна |
| Закрыть окно | Alt+F3 |  |
| Загрузка программы | - | Меню File/Open… |
| Запуск программы | F9 | Программа выполняется до конца или до точки останова |
| Установка точки останова | Alt+F2 | Установить маркер на требуемую команду, а затем Alt+F2. Команда выделяется красным цветом. |
| Снятие точки останова | F2 | Установить маркер на требуемую команду, а затем F2. Снимается выделение красным цветом. |
| Пошаговое выполнение | F7 |  |
| Запуск после точки останова | F9 |  |
| Выбор исполняемой команды | Ctrl+N | Позволяет сделать очередной любую команду |
| Изменение данных | Ctrl+C | Установить маркер на редактируемую ячейку или регистр, а затем Ctrl+C |
| Установка типа изменяемых данных | Ctrl+D | При редактировании ячейки памяти можно установить: byte, word и т.д. |
| Просмотр памяти | Ctrl+G | В диалоговом окне ввести адрес начала области просмотра |
| Поиск | Ctrl+S | Поиск необходимой инструкции или содержимого ячейки памяти |

* Загрузка среды TURBO Debugger и выход из неё

Находясь в каталоге, имеющем загрузочный файл среды TURBO Debugger, требуется в командной строке ввести команду:

C:\TAsm\>td.exe

Выход из среды TURBO Debugger: {Alt+X}.

* Загрузка исполняемого модуля программы в отладчик TDBG:

C:\TAsm\>td.exe < Имя\_исполняемого\_модуля >

Для набора и корректировки программ используется режим дизассемблера, который устанавливается автоматически при входе в окно листинга.

Переход из окна в окно осуществляется клавишей {TAB} (активным считается то окно, в котором находится курсор). В пределах одного окна курсор перемещается с помощью стрелок. Вносить изменения можно только в поле, в котором находится курсор.

Объём текста, записанного в одном из окон, может быть больше, чем мы видим, поэтому пролистывание текста можно осуществлять с помощью клавиш {PageUp} и {PageDown}.

Отладчик имеет  меню  (расположенное в верхней части экрана), в котором находятся команды, необходимые для поддержки работы, и строку-подсказку с функциональными клавишами, расположенную внизу экрана.

* Ввод мнемонических кодов (команд ассемблера).

Находясь в центральном окне дизассемблера, содержащем мнемонические машинные коды, требуется подвести курсор к нужной строке и начать ввод инструкции. При вводе первого символа появится окно:

Enter instruction to assemble

OK CANCEL HELP

Ввод команды, отображаемой в появившемся окне, завершается нажатием клавиши {ENTER}. Если команда введена неправильно, то появится окно с сообщением об ошибке, в противном случае - команда отобразится в предполагаемой строке окна дизассемблера.

* Ввод значений регистров и флагов

Находясь в окне регистров и выбрав регистр или флаг для изменения его значения, требуется начать ввод числа. При вводе первой цифры появится окно с отображением вводимого числа:

Enter new value

OK CANCEL HELP

Если значение введено неправильно, то появится сообщение об ошибке, в противном случае - намеченный регистр или флаг приобретёт новое значение.

* Ввод значений байтов памяти

Находясь в окне дампа, отображающего участок памяти, можно изменить значение какого-либо байта. Подведя курсор к нужному байту, начать ввод числа, учитывая его систему счисления:

Enter new data bytes

OK CANCEL HELP

* Перемещение в определённый участок памяти

Перемещение по памяти на нужный участок осуществляется гораздо быстрее, зная адрес. В окне дизассемблера, дампа или стека (в зависимости от того, в каком сегменте требуется перемещаться), нужно нажать комбинацию клавиш {CTRL+G}. В появившемся окне останется только набрать логический адрес ячейки памяти (сегментный адрес:смещение, например, DS:0000).

* Выполнение программы в TURBO Debugger

Запустить программу на выполнение (написанную ранее или исправленную в отладчике) можно в одном из четырёх режимов:

(1) режим безусловного выполнения;

(2) выполнение по шагам;

(3) выполнение до текущего положения курсора;

(4) выполнение с установкой точек прерывания.

(1)  Режим безусловного выполнения  целесообразно применять, когда требуется посмотреть на общее выполнение программы.

Запуск программы: {F9}.

В точках программы, где предусмотрен ввод данных с клавиатуры и вывод их на экран, отладчик, в соответствии с логикой работы применяемого средства ввода, будет осуществлять определённые действия.

Просмотр результатов работы программы на экране дисплея можно осуществить с помощью комбинации клавиш {Alt+F5}.

(2)  Выполнение по шагам применяется для детального изучения работы программы по командам. При этом можно наблюдать состояние регистров, флагов и памяти после каждой выполненной команды.

Просмотр выполнения команд программы с просмотром команд процедур или прерываний, встречаемых в потоке команд: {F7}.

Просмотр выполнения команд программы без просмотра команд процедур или прерываний, встречаемых в потоке команд (т.е. вызов процедуры или прерывания отрабатывается как одна обычная команда): {F8}.

(3) Выполнение до текущего положения курсора позволяет выполнить программу по шагам, начиная с произвольного места программы. Этот режим целесообразно использовать в том случае, если Вас интересует только правильность функционирования некоторого участка программы.

Для активизации этого режима необходимо установить курсор на нужную строку программы и нажать клавишу {F4}.

Программа начнёт выполнение и остановится на отмеченной команде, не выполнив её. Далее Вы можете использовать при необходимости пошаговый режим.

(4) Выполнение с установкой точек прерывания позволяет выполнить программу с остановкой её в строго определённых  точках прерывания.

Перед выполнением программы необходимо установить эти точки в программе, для чего следует перейти в нужную строку и нажать клавишу {F2}. Выбранные строки будут подсвечиваться другим цветом.

После установки точек прерывания программа запускается на выполнение клавишей {F9}. На первой точке прерывания программа остановится. Теперь можно посмотреть состояние микропроцессора и памяти, а затем продолжить выполнение программы. Сделать это можно или в пошаговом режиме, или выполнив программу до следующей точки прерывания.

Установленные ранее точки прерывания можно убрать - для этого нужно повторно выбрать нужные строки и нажать клавишу {F2}.

Прервать выполнение программы в любом из этих режимов можно, нажав комбинацию клавиш {CTRL+F2}.

 Замечания

Во всех случаях программа начинает выполнение с адреса, находящегося в счетчике команд IP (в окне регистров). Поэтому, прежде чем выполнить программу, поместите в IP адрес начала программы.

 Внесённые в программу изменения, непосредственно в отладчике, не отразятся на исходной программе.

Выполнение программы с самого начала можно осуществить с помощью комбинации клавиш {CTRL+F2}.

# Лабораторная работа №2. Команды пересылки данных. Арифметические команды

Вопросы, необходимые рассмотреть при подготовке к лабораторной работе:

1. Команды пересылки данных, осуществляющие обмен информацией.
2. Арифметические команды, выполняющие арифметические операции над числами со знаком и без знака.
3. Команда сравнения как команда с арифметическим принципом действия

**Команды пересылки** таблица 1.

|  |  |
| --- | --- |
| **Мнемокод** | **Формат** |
| Команды общего назначения  MOV | MOV приемник, источник |
| XCHG | XCHG приемник, источник |
| XLAT | XLAT таблица\_источник |
| Команды пересылки адреса  LEA | LEA регистр (16), память (16) |
| LDS | LDS регистр (16), память (32) |
| LES | LES регистр (16), память (32) |
| Команды пересылки флагов  LAHF | LAHF |
| SAHF | SAHF |

**Арифметические команды**

|  |  |
| --- | --- |
| **Мнемокод** | **Формат** |
| Команды сложения  ADD | ADD приемник, источник |
| ADC | ADC приемник, источник |
| AAA | AAA |
| DAA | DAA |
| INC | INC приемник |
| Команды вычитания  SUB | SUB приемник, источник |
| SBB | SBB приемник, источник |
| AAS | AAS |
| DAS | DAS |
| DEC | DEC приемник |
| NEG | NEG приемник |
| CMP | CMP приемник, источник |
| Команды умножения  MUL | MUL источник |
| IMUL | IMUL источник |
| AAM | AAM |
| Команды деления  DIV | DIV источник |
| IDIV | IDIV источник |
| AAD | AAD |
| Команды расширения знака  CBW | CBW |
| CWD | CWD |

**Демонстрационные примеры**

Исследуйте примеры в TURBO Debugger. Для этого выполните ассемблирование и компоновку программы:

tasm /zi<имя>.asm

tlink /t<имя>.obj

td<имя>.exe

После загрузки программы в TURBO Debugger, выполните программу в пошаговом режиме.

Пример1. Демонстрация арифметических команд сложения и вычитания, производимых над целыми числами

CODE SEGMENT

ORG 100h

ASSUME cs:code, ds:code

Begin:

; -------------------------

; Сложение двух целых чисел

; -------------------------

mov al,0FFh ; (AL)=0FFh

mov bl,1 ; (BL)=1

add al,bl ; (AL)=(AL)+(BL) - превышает 8-разрядное число

; 00h=0FFh+1

; 1 --------> (CF)

;

adc ah,0 ; (AH)=(AH)+0+(CF) - корректировка суммы, ;полученной в регистре AL

; (AX)=(AH)|(AL)

; 1 00h

; ---------------------------

; Вычитание двух целых чисел

; -------------------------------------------

mov cx,0 ; Обнуление значения регистра CX

sub cl,bl ; (CL)=(CL)-1

; FFh=00h-1

; .

; 0 - заём из старшего разряда: 1 ---> (CF)

sbb ch,0 ; (CH)=(CH)-0-(CF) - корректировка разности,

; полученной в регистре CL

; (CX)=(CH)|(CL)

; FF FFh

mov ax,4c00h

int 21h

CODE ENDS

END Begin

 Пример2. Демонстрация арифметической команды умножения, производимой над целыми числами

CODE SEGMENT

ORG 100h

ASSUME cs:code, ds:code

Begin:

; ------------------------------------

; Умножение двух целых чисел без знака

; ------------------------------------

mov al,100 ; (AL)=100

mov bl,2 ; (BL)=2

mul bl ; (AX)=(AL)\*(BL)

; 200=100 \* 2

mov cx,ax

; ----------

mov ax,30000 ; (AX)=30000

mov bx,2 ; (BX)=2

mul bx ; (DX)(AX)=(AX)\*(BX)

; 60000=30000\*2

; ------------------------------------

; Умножение двух целых чисел со знаком

; ------------------------------------

mov ax,0 ; (AX)=0

mov bx,0 ; (BX)=0

mov al,-4 ; (AL)=-4

mov bl,4 ; (BL)=4

imul bl ; (AH)|(AL)=(AL)\*(BL)

; FF|F0h=FCh\*4

; (AH) - знаковое расширение регистра AL

mov cx,ax

; ----------

mov ax,1249h ; (AX)=4681

mov bx,-7 ; (BX)=-7

imul bx ; (DX)|(AX)=(AX)\*(BX)

; FFFF|8001h=1249h\*FFF9h

; (DX) - знаковое расширение регистра AX

mov ax,4c00h

int 21h

CODE ENDS

END Begin

 Пример3. Демонстрация арифметической команды деления, производимой над целыми числами.

CODE SEGMENT

ORG 100h

ASSUME cs:code, ds:code

Begin:

; ----------------------------------

; Деление двух целых чисел без знака

; ----------------------------------

mov ax,27 ; (AX)=27

mov bl,4 ; (BL)=4

div bl ; (AH)|(AL)=(AX)/(BL)

; 3 | 24 = 27 / 4

; (AH)=3 - остаток от деления

; (AL)=24 - частное от деления

mov cx,ax

; -------

mov dx,0 ; (DX)=0 - для сохранения остатка от деления

mov ax,40199 ; (AX)=40199

mov bx,200 ; (BX)=200

div bx ; (DX)|(AX)=(AX)/(BX)

; 199 | 200=40199/200

; (DX)=199 - остаток от деления

; (AX)=200 - частное от деления

; ------------------------------------------

; Деление двух целых чисел со знаком

; ----------------------------------

mov ax,0 ; (AX)=0

mov bx,0 ; (BX)=0

mov dx,0 ; (DX)=0 - для сохранения остатка от деления

mov ax,-48 ; (AL)=-48

mov bl,5 ; (BL)=5

idiv bl ; (AH)|(AL)=(AX)/(BL)

; -3 | -9 =(-48)/5

; (AH)=-3 - остаток от деления

; (AL)=-9 - частное от деления

mov cx,ax

; ---------

mov ax,-100 ; (AX)=-100

mov bl,-3 ; (BL)=-3

idiv bl ; (AH)|(AL)=(AX)/(BL)

; -1 | 33 =(-100)/(-3)

; (AH)=-1 - остаток от деления

; (AL)=33 - частное от деления

; ------

mov dx,0 ; (DX)=0 - для сохранения остатка от деления

mov ax,300 ; (AX)=300

mov bx,-200 ; (BX)=-200

idiv bx ; (DX)|(AX)=(AX)/(BX)

; 100 | -1 =300/(-200)

; (DX)=100 - остаток от деления

; (AX)=-1 - частное от деления

mov ax,4C00h

int 21h

CODE ENDS

END Begin

Пример4: Демонстрация нахождения суммы первых трёх членов арифметической прогрессии S(n)=(2\*a1+d\*(n-1))\*n/2 при заданных значениях a1=3 и d=5, n=3.

CODE SEGMENT

ORG 100h

ASSUME cs:code,ds:code

Begin:

; --------------------------------

; Ввод значений переменных a, d, n

; --------------------------------

mov ax,3 ; (AX)=(a)

mov bx,3 ; (BX)=(n)

mov cx,5 ; (CX)=(d)

; -----------------------------------------------------

; Вычисление значения выражения: S(n)=(2\*a+d\*(n-1))\*n/2

; -----------------------------------------------------

mov dl,2 ; (DL)=2

mul dl ; (AX)=(AX)\*(DL), (AX)=(a)\*2

mov dx,ax ; (DX)=(a)\*2

; -------

dec bx ; (BX)=(BX)-1, (BX)=(n)-1

mov ax,bx ; (AX)=(n)-1

; -------

mul cl ; (AX)=(AX)\*(CL), (AX)=((n)-1)\*(d)

; -------

add ax,dx ; (AX)=(AX)+(DX), (AX)=((n)-1)\*(d)+(a)\*2

; -------

inc bx ; (BX)=(BX)+1, (BX)=(n)-1+1

mul bl ;(AX)=(AX)\*(BL),(AX)=(((n)-1)\*(d)+(a)\*2)\*(n)

; ------

mov dl,2 ; (DL)=2

div dl ; (AH)|(AL)=(((n)-1)\*(d)+(a)\*2)\*(n)/2

; --------

mov ax,4C00h

int 21h

CODE ENDS

END Begin

**Задачи для самостоятельного решения**

1. Сложите два числа, находящихся соответственно в регистрах AX и BX. Результат поместите в регистр AX.
2. Сложите три числа. Результат поместите в регистр AX.
3. Произведите вычитание числа, находящегося в регистре AX, из числа, помещённого в регистр BX.
4. Перемножьте два числа. Результат оставьте в регистре AX.
5. Перемножьте три числа, находящихся соответственно в регистрах AX, BX, CX. Результат поместите в регистр AX.
6. По данным сторонам прямоугольника вычислите его периметр и площадь.
7. Найдите сумму остатков от деления целых чисел a, b и c (занимающих одно слово) на k и поместите её в регистр AX
8. Выполните следующие действия:

1. Х= А - 5 (В - 2С) + 2 2. Х= - 4А + (В + С) / 4 + 2

3. Х= 7А - 2В - 100 + С 4. Х= - А / 2 + 4 (В + 1) + 3С

5. Х= 5 (А - В) - 2С + 5 6. Х= (А/ 2 + В) / 4 + С – 1

7. Х= - (С + 2А + 4В + В) 8. Х= 6С + (В - С + 1) / 2

1. Оформите приведённую ниже последовательность команд в виде com-программы, зафиксировав в комментариях для каждой команды причину изменения того или иного флага и указав содержимое регистра, изменившего своё значение:

(1) mov ax,7FF0h (2) mov ax,0 (3) mov ax,0 (4) mov ax,0

add al,10h mov cx,0 sub al,1 mov bx,0

add ah,1 mov al,0F0h mov al,-128 mov al,22

add ax,2 mov cl,78h mov bl,26

add al,cl sub al,bl

adc ah,ch sbb ah,0

(5) mov ax,30000 (6) mov ax,0 (7) mov dx,0 (8) mov ax,23

mov bx,2 mov al,0 mov ax,2 mov bl,-5

imul bx mov bl,255 mov bx,3 idiv bl

mul bl div bx

# Лабораторная работа №3 . Команды для работы с битами.

Вопросы, необходимые рассмотреть при подготовке к лабораторной работе:

1. логические операции, таблицы истинности для логических операций;
2. логические команды;
3. команды обработки бит;
4. команды сдвига.

Команды этой группы сведены в таблицу.

|  |  |
| --- | --- |
| **Мнемокод** | **Формат** |
| Логические команды  AND | AND приемник, источник |
| OR | OR приемник, источник |
| ХOR | ХOR приемник, источник |
| NOT | NOT приемник |
| TEST | TEST приемник, источник |
| Команды сдвига  SAL, SHL | SAL, SHL приемник, источник |
| SAR | SAR приемник, источник |
| SHR | SHR приемник, источник |
| Команды циклического сдвига  ROL | ROL приемник, источник |
| ROR | ROR приемник, источник |
| RCL | RCL приемник, источник |
| RCR | RCR приемник, источник |

**Демонстрационные примеры**

Пример 1. Демонстрация синтаксиса и семантики логических команд and, or, xor, not, test

CODE SEGMENT

ORG 100h

ASSUME cs:code, ds:code

Begin:

mov ax,0100001100100001b ; (BX)=4321h

mov bx,ax

; --------------------------------------------

; Обнуление битов регистра BX c 4-го по 7-ой и с 12-го по 15-ый

; ----------------------

and bx,0000111100001111b ; (BX)=0301h

mov cx,bx ; (CX)=(BX)=0301h

; ----------------------------------------

; Установка 5-го и 14-го битов регистра CX

; ----------------------------------------

or cx,0100000000100000b ; (CX)=4321h

; ------------------------------------------

; Проверка того факта, совпадают ли значения

; регистров AX и CX. В случае положительного

; ответа значение регистра AX станет 0

; ------------------------------------

xor ax,cx ; (AX)=0

; ---------------------------------------

; Инвертирование 0-го, 5-го, 8-го, 9-го и

; 14-го битов регистра AX

; -----------------------------------

xor ax,0100001100100001b ; (AX)=4321h

; --------------------------------

; Инвертирование битов регистра DX

; -----------------------------------

not dx ; (DX)=FFFFh

; ---------------------------------

; Установка того факта, все ли биты

; регистра dx установлены в единицу

; ---------------------------------

test dx,0 ; (ZF)=1

; --------

Mov ax, 4C00h

int 21h

CODE ENDS ; Конец сегмента кода

END Begin

Пример 2. Демонстрация установления чётности данного целого числа

include mcr\_bibl.inc

CODE SEGMENT

ORG 100h

ASSUME cs:code, ds:code

Begin:

Print str1 ; Печать строки

Vvod\_AX ; Ввод целого числа в регистр AX

mov bx,1 ; (BX)=1

and ax,bx ; (AX)=0 или (AX)=1

; --------

Print str2 ; Печать строки

Vivod\_AX ; Печать целого числа из регистра AX

Print str3 ; Печать строки

; --------

mov ah,08h ; Функция задержки экрана

int 21h ; Вызов функции DOS 21h

mov ax,4C00h ; Функция стандартного выхода из программы

int 21h ; Вызов функции DOS 4C00h

; ----------------------------------------

str1 DB "Введите целое число: ",'$'

str2 DB "Результат: ",'$'

str3 DB 0Ah,0Dh,"'1' - знак того, что введённое число"

DB "чётное.",0Ah,0Dh,"'0'-знак того, что введённое"

DB "число нечётное.",'$'

; ------------------------------------

CODE ENDS ; Конец сегмента кода

END Begin

Пример 3. Демонстрация вычисления выражения 2 в степени n по заданному натуральному числу n

include mcr\_bibl.inc

CODE SEGMENT

ORG 100h

ASSUME cs:code, ds:code

Begin:

Print str1 ; Печать строки

Vvod\_AL\_pos ; Ввод целого числа n>=0 в регистр AL

mov cl,al ; (CL)=n

mov bx,1 ; (BX)=1

shl bx,cl ; Логический сдвиг (BX) влево на n битов

mov ax,bx ; (AX)=(BX)

; --------

Print str2 ; Печать строки

Vivod\_AX ; Печать целого числа из регистра AX

; --------

mov ah,08h ; Функция задержки экрана

int 21h ; Вызов функции DOS 08h

; --------

mov ax,4C00h

int 21h

; ----------------------------------------

str1 DB "Введите натуральное число n=",'$'

str2 DB " 2^n=",'$'

; ----------------------------------------

CODE ENDS ; Конец сегмента кода

END Begin

Пример 4. Демонстрация синтаксиса и семантики команд: shr (логический сдвиг вправо), sal (арифметический сдвиг вправо)

include mcr\_bibl.inc

CODE SEGMENT

ORG 100h

ASSUME cs:code, ds:code

Begin:

Print prigl

Vvod\_AX ; Ввод целого числа в регистр AX

mov bx,ax ; (AX)=(BX)

; -------

shr ax,1 ; Логический сдвиг (AX) вправо на 1 бит

sar bx,1 ; Арифметический сдвиг(BX)вправо на 1бит

Print str1 ; Печать строки

Vivod\_AX ; Печать целого числа из регистра AX

; -------

mov ax,bx

Print str2 ; Печать строки

Vivod\_AX ; Печать целого числа из регистра AX

; --------

mov ah,08h ; Функция задержки экрана

int 21h ; Вызов функции DOS 08h

; --------

mov ax,4C00h

int 21h

; ----------------------------------------

prigl DB "Введите целое число: ",'$'

str1 DB "Результат выполнения логической команды "

DB "'shr' : ",'$'

str2 DB 0Ah,0Dh,"Результат выполнения арифметической "

DB "команды 'sal': ",'$'

; -----------------------------------

CODE ENDS ; Конец сегмента кода

END Begin

Пример 5. Демонстрация умножения числа (размером в двойное слово), располагающегося в паре регистров DX и AX, на 4

CODE SEGMENT

ORG 100h

ASSUME cs:code

Begin:

; --------------------------

; Умножение числа 65537 на 4

; --------------------------

mov ax,65535

add ax,2

adc dx,dx

; (DX\_AX)=65537\*4

shl ax,1 ; Бит 15 регистра AX сдвигается во флаг переноса

rcl dx,1 ; Флаг переноса сдвигается в бит 0 регистра DX

shl ax,1 ; Бит 15 регистра AX сдвигается во флаг переноса

rcl dx,1 ; Флаг переноса сдвигается в бит 0 регистра DX

; -------------------------------------------------------

; Умножение числа -32769 на 4

; ---------------------------

mov ax,-32768

add ax,-1

sbb dx,dx

; (DX\_AX)=-32769\*4

shl ax,1 ; Бит 15 регистра AX сдвигается во флаг переноса

rcl dx,1 ; Флаг переноса сдвигается в бит 0 регистра DX

shl ax,1 ; Бит 15 регистра AX сдвигается во флаг переноса

rcl dx,1 ; Флаг переноса сдвигается в бит 0 регистра DX

mov ax,4C00h ; Функция стандартного выхода из программы

int 21h ; Вызов функции DOS 4C00h

CODE ENDS ; Конец сегмента кода

END Begin

Пример 6. Демонстрация установки флага ZF во флаг CF

CODE SEGMENT

ORG 100h

ASSUME cs:code

Begin:

xor ax,ax ; Установка флага ZF

lahf ; Содержимое регистра флагов - в регистре AH

; (AH)=(S)(Z)0(A)0(P)1(C)b

and ah,1000000b ; (AH)=01000000b

shl ah,2 ; Логический сдвиг (AH) влево на 2 бита

; ----------

mov ax,4C00h

int 21h

CODE ENDS

END Begin

Пример 7. Демонстрация установки флагов SF, ZF и CF по содержимому регистра AH

CODE SEGMENT

ORG 100h

ASSUME cs:code

Begin:

mov ah,11000011b ; (AH)=(S)(Z)0(A)0(P)1(C)b

sahf ; (SF)=1, (ZF)=1, (CF)=1

; ----------

mov ax,4C00h

int 21h

CODE ENDS

END Begin

**Задачи для самостоятельного решения**

1. Составьте программу, позволяющую записать в регистр AL пятый бит регистра BX.
2. Напишите программу умножения числа на любую степень двойки.
3. Напишите программу деления числа на любую степень двойки.
4. Составьте программу, позволяющую находить целую часть частного и остаток при делении заданного целого числа на заданную степень 2.
5. Составьте программу, позволяющую вычислять выражение 4 в степени n для данного натурального числа n.
6. Напишите программу, позволяющую делить числа (размером в двойное слово), располагающиеся в паре регистров DX и AX, на 8.
7. В задачах нижеследующего раздела в качестве исходного значения предполагается байт, условно названный здесь **DataByte,** который может быть предварительно записан (командой **MOV**) в один из регистров МП. Разработать программу, которая выделяет данные из указанных битовых полей (каждое такое поле рассматривается как целое без знака), производит над ними указанные действия и выводит результат в заданный регистр МП.
   1. В исходном байте DataByte биты 0…3 содержат двоичный код, который обозначим как А, биты 4…5 - код, который обозначим как В, биты 6…7 - код, который обозначим как С. Вычислить А+2\*В+2\*С. **Тест:** ввод - 51h, вывод - 5.
   2. В исходном байте DataByte биты 0…2 содержат двоичный код, который обозначим как А, биты 3…5 - код, который обозначим как В, биты 6…7 - код, который обозначим как С. Вычислить А-В+2\*С. **Тест:** ввод - 8Fh, вывод - 10 (0Ah).
   3. В исходном байте DataByte биты 0…1 содержат двоичный код, который обозначим как А, биты 2…5 - код, который обозначим как В, биты 6…7 - код, который обозначим как С. Вычислить В-A\*С. **Тест:** ввод - BEh, вывод - 11 (0Bh).
   4. В исходном байте DataByte биты 0…4 содержат двоичный код, который обозначим как А, биты 5…6 - код, который обозначим как В, бит 7 - код, который обозначим как С. Вычислить А+В+С. **Тест:** ввод - C7h, вывод - 10 (0Ah).
   5. В исходном байте DataByte биты 0…2 содержат двоичный код, который обозначим как А, биты 3…6 - код, который обозначим как В, бит 7 - код, который обозначим как С. Вычислить В-A+С. **Тест:** ввод - D6h, вывод - 5.

# Лабораторная работа №4 . Массивы

Вопросы, необходимые рассмотреть при подготовке к лабораторной работе:

1. Массив. Одномерный массив. Основные характеристики массива.
2. Описание массива.
3. Индексирование массивов.
4. Матрицы (двухмерные массивы). Описание матрицы. Адресация элементов двухмерных массивов.

**Демонстрационные примеры**

 Пример1. Найти сумму всех элементов массива. Результат поместить в регистр ax.

%TITLE "Сумма всех элементов массива"

IDEAL

MODEL small

STACK 256

;--------

DATASEG

exCode DB 0

x DW 1,2,3,4,5

;--------

UDATASEG

index DW ?

;--------

CODESEG

Start:

mov ax, @data

mov ds, ax

call AddArray

Exit:

mov ah,04Ch

mov al,[exCode]

int 21h

;---------------

PROC AddArray

mov si, [index] ; Получить значение индекса

mov ax, 0 ; Начальное значение суммы

mov cx, 5 ; Счетчик цикла

Repeat:

add ax,[x+si] ; ax:= ax+x[i]

add si, 2 ; Следующий индекс

loop Repeat ; Выполнение цикла 5 раз

ret

ENDP AddArray

END Start

Пример2. Использование цикла для инициализации значений области памяти, которую можно будет впоследствии трактовать как массив.

%TITLE "Инициализация массива в цикле."

MASM ; Режим

MODEL small ; Модель памяти

.STACK 256

.DATA ; Сегмент данных

mes DB 0ah,0dh,"Массив: ","$"

mas DB 10 DUP (?) ; Исходный массив

i DB 0

.CODE

Main: ; Точка входа в программу

mov ax,@data ; Загрузка сегментных регистров

mov ds,ax ; Настройка регистров ds и es

xor ax,ax ; Обнуление ax

mov cx,10 ; Значение счётчика цикла

mov si,0 ; Индекс начального элемента в cx

Go: ; Цикл инициализации

mov bh,i ; i в bh

mov mas[si],bh ; Запись в массив i

inc i ; Инкремент i

inc si ; Продвижение к следующему

; Элементу массива

loop Go ; Повторить цикл

;Вывод на экран получившегося массива

mov cx,10

mov si,0

mov ah,09h

lea dx,mes

int 21h

Show:

mov ah,02h ; Функция вывода значения из al

; на экран

mov dl,mas[si]

add dl,30h ; Преобразование числа в символ

int 21h

inc si

loop Show

Exit:

mov ax,4c00h ; Стандартный выход

int 21h

END Main

Пример3. Рассматривая элементы массива х как числа со знаком, записать в регистр al индекс (от 0 до n-1) максимального элемента массива (первого, если таких элементов несколько).

Указание. Так как у нас числа со знаком, то массив будет состоять из слов, индекс следующего элемента будет отличаться от предыдущего на 2. Чтобы получить действительный индекс максимального элемента, мы разделим его на 2.

%TITLE "Найти индекс максимального элемента."

IDEAL

MODEL small

STACK 256

;--------

DATASEG

exCode DB 0

n EQU 5

x DW -1,0,1,0,-1

;--------

UDATASEG

index DW ?

;--------

CODESEG

Start:

mov ax, @data

mov ds, ax

call MaxElement

Exit:

mov ah, 04Ch

mov al, [exCode]

int 21h

;---------------

PROC MaxElement

mov si,[index] ; Получить значение индекса

mov bx,x ; bx:= x[0] - начальный max

mov ax,0 ; В ax индекс максимума

add si,2 ; Перейти к следующему элементу

mov cx,n-1 ; Счетчик цикла

Repeat:

cmp [x+si],bx ; x[i]=bx?

jle Max1 ; Переход, при '=<'

mov bx,[x+si] ; Запомнить новый max

mov ax,si ; и его индекс

Max1:

add si,2 ; Следующий индекс

loop Repeat ; Цикл n-1 раз

mov cl,2 ; ax:=ax/2

div cl

ret

ENDP MaxElement

END Start

Пример4. Требуется циклически сдвинуть элементы массива Y на две позиции вперед, т.е. кортеж (Y[0],Y[1],Y[2],...,Y[n-1]) преобразовать в кортеж Y1=(Y[2],...,Y[n-1],Y[0],Y[1]).

Здесь в цикле нужно выполнять операцию Y[i]:=Y[i+2]. И надо помнить, что в памяти элементы массива располагаются последовательно.

%TITLE "Циклический сдвиг элементов массива вправо на 2"

IDEAL

MODEL small

STACK 256

;-------

DATASEG

exCode DB 0

n EQU 5

y DB 1,2,3,4,5

y1 DB 5 DUP(?)

index DW 0

;-------

CODESEG

Start:

mov ax, @data

mov ds,ax

call Rotate

Exit:

mov ah, 04Ch

mov al, [exCode]

int 21h

;---------------

PROC Rotate

mov si,[index]

mov di,[index]

mov cx,n-2

Repeat:

mov bh,[y+si+2]

mov [y1+di],bh

inc si

inc di

loop Repeat

mov si,[index]

mov ah,[y+si]

mov al,[y+si+1]

mov [y1+n-2],ah

mov [y1+n-1],al

ret

ENDP Rotate

END Start

Пример5 Поиск элемента в двухмерном массиве чисел. Элементы массива заданы статически.

%TITLE "Поиск элемента в двухмерном массиве."

MASM ; Режим

MODEL small ; Модель памяти

.STACK 256

.DATA ; Сегмент данных

;---------------------------------------------------

;Матрица размером 2x5 - если её не инициализировать,

;то для наглядности она может быть описана так:

;array dw 2 DUP (5 DUP (?))

;но мы её инициализируем.

;---------------------------------------------------

array dw 1,2,3,4,5,6,7,8,9,0

;---------------------------------------------------

;Логически это будет выглядеть так:

;array = {1 2}

; {3 4}

; {5 6}

; {7 8}

; {9 0}

;---------------------------------------------------

elem dw 3 ; Элемент для поиска

failed db 0ah,0dh,'Нет такого элемента в массиве!','$'

success db 0ah,0dh,'Такой элемент в массиве присутствует ","$"

foundtime db ? ; Количество найденных элементов

fnd db " раз(а)",0ah,0dh,'$'

.CODE

Main: ; Точка входа в программу

mov ax,@data ; Загрузка сегментных регистров

mov ds,ax ; Настройка регистров ds и es

xor ax,ax ; Обнуление ax

mov si,0 ; si=столбцы в матрице

mov bx,0 ; bx=строки в матрице

mov cx,5 ; Число для внешнего цикла (по строкам)

External: ; Внешний цикл по строкам

push cx ; Сохранение в стеке счётчика внешнего цикла

mov cx,2 ; Число для внутреннего цикла (по столбцам)

mov si,0

Iternal: ; Внутренний цикл по строкам

mov ax,array[bx][si] ; В регистр ax помещает первый элемент

; матрицы

ins si ; Передвижение на следующий элемент в строке

;---------------------------------------------------------------

;Сравниваем содержимое текущего элемента в ax с искомым элемен-

;том.

;---------------------------------------------------------------

cmp ax,elem

;---------------------------------------------------------------

;Если текущий элемент совпал с искомым, то переход на метку Here

;для обработки, иначе - цикл продолжения поиска.

;---------------------------------------------------------------

je Here

;---------------------------------

;Иначе - цикл по строке cx=2 раза.

;---------------------------------

loop Iternal

Here:

jcxz Move\_next ; Просмотрели строку?

inc foundtime ; Иначе - увеличиваем счётчик совпавших

Move\_next: ; Продвижение в матрице

pop cx ; Восстанавливаем cx из стека (5)

add bx,4 ; Передвигаемся на следующую строку

loop External ; Цикл (внешний)

cmp foundtime,0h ; Сравнение числа совпавших с 0

ja Eql ; Если больше 0, то переход

Not\_equal: ; Нет элементов, совпавших с искомым

mov ah,09h ; Вывод сообщения на экран

mov dx,offset failed

int 21h

jmp Exit ; На выход

Eql: ; Есть элементы, совпавшие с искомым

mov ah,09h ; Вывод сообщений на экран

mov dx,offset success

int 21h

mov ah,02h

mov dl,foundtime

add dl,30h

int 21h

mov ah,09h

mov dx,offset fnd

int 21h

Exit: ; Выход

mov ax,4c00h ; Стандартное завершение программы

int 21h

END Main

Пример6  Демонстрация сортировки массива по возрастанию

MASM

MODEL small

.STACK 256

.DATA

mes1 DB 0Ah,0Dh,'Исходный массив - $',0Ah,0Dh

mes2 DB 0Ah,0Dh,'Отсортированный массив - $',0Ah,0Dh

n EQU 9 ; Количество элементов в массиве, считая с 0

mas DW 2,7,4,0,1,9,3,6,5,8 ; Исходный массив

tmp DW 0 ; Переменные для работы с массивом

i DW 0

j DW 0

.CODE

Main:

mov ax,@data

mov ds,ax

xor ax,ax

; Вывод на экран исходного массива

mov ah,09h

lea dx,mes1

int 21h ; Вывод сообщения mes1

mov cx,10

mov si,0

Show\_primary: ; Вывод значения элементов исходного массива

mov dx,mas[si] ; на экран

add dl,30h

mov ah,02

int 21h

add si,2

loop Show\_primary

; ---------------------------------------

; Следующие строки программы эквивалентны

; следующему коду на языке C:

; for (i=0;i<9;i++)

; for (j=9;j>i;j-)

; if (mas[i]>mas[j])

; { tmp=mas[i];

; mas[i]=mas[j];

; mas[j]=tmp; }

; -----------------------------------

mov i,0 ; Инициализация i

; Внутренний цикл по j

Internal:

mov j,9 ; Инициализация j

jmp Cycl\_j ; Переход на тело цикла

Exchange:

mov bx,i ; В регистр BX - i

shl bx,1

mov ax,mas[bx] ; В регистр AX - mas[i]

mov bx,j ; В регистр BX - j

shl bx,1

cmp ax,mas[bx] ; mas[i]?mas[j] - сравнение элементов если

jle Lesser ; mas[i] меньше, то обмен не нужен и переход

; на продвижение далее по массиву, иначе

; tmp=mas[i], mas[i]=mas[j], mas[j]=tmp,

; tmp=mas[i]

mov bx,i ; В регистр BX - i

shl bx,1 ; Умножаем на 2, так как элементы - слова

mov tmp,ax ; tmp=mas[i]

; mas[i]=mas[j]

mov bx,j ; В регистр BX - j

shl bx,1 ; Умножаем на 2, так как элементы - слова

mov ax,mas[bx] ; В регистр AX - mas[j]

mov bx,i ; В регистр BX - i

shl bx,1 ; Умножаем на 2, так как элементы - слова

mov mas[bx],ax ; mas[i]=mas[j]

; mas[j]=tmp

mov bx,j ; В регистр BX - j

shl bx,1 ; Умножаем на 2, так как элементы - слова

mov ax,tmp ; В регистр AX - tmp

mov mas[bx],ax ; mas[j]=tmp

Lesser: ; Продвижение далее по массиву во внутреннем

; цикле

dec j

; Тело цикла по j

Cycl\_j:

mov ax,j ; В регистр AX - j

cmp ax,i ; Сравнить j?i

jg Exchange ; Если j>i, то переход на обмен, иначе

; на внешний цикл по i

inc i ; i++

cmp i,n ; Сравнить i?n - прошли до конца массива

jl Internal ; Если i<n - продолжение обработки

; Вывод отсортированного массива

mov ah,09h

lea dx,mes2

int 21h

Prepare:

mov cx,10

mov si,0

Show: ; Вывод значения элемента на экран

mov dx,mas[si]

add dl,30h

mov ah,02h

int 21h

add si,2

loop Show

Exit:

mov ah,08h ; Функция ожидания нажатия любой клавиши

int 21h ; Вызов DOS: код нажатой клавиши - в AL

mov ax,4C00h ; Стандартный выход

int 21h

END Main ; Конец программы

**Задачи для самостоятельного решения**

1. Для некоторого вектора произвольной длины найти максимум, минимум и значение, наиболее близкое к среднему.
2. Для некоторой двумерной матрицы A найдите определитель D(A)=a11a22-a21a12
3. В одномерном массиве переставьте первый и последний элементы местами.
4. Напишите программу, которая по заданному натуральному числу n находит наименьший элемент массива B[0,n-1].
5. Напишите программу, которая по заданному натуральному числу n находит индекс заданного элемента массива B[0,n-1].
6. Замените минимальный элемент массива А[1..5] его удвоенным индексом.
7. Осуществите циклическую перестановку элементов массива: первый элемент должен стать вторым, второй - третьим и т.д., последний - первым. Нового массива не заводить.
8. Напишите программу, увеличивающую на 2 все элементы массива Y[1..n], которые больше 10.
9. Переставьте элементы одномерного массива в обратном порядке.
10. Напишите программу, которая по заданному натуральному числу n вычисляет сумму элементов массива B[0,n-1].
11. Для целого массива A[1..n] найдите сумму всех: (а) положительных элементов; (б) отрицательных; (в) ненулевых элементов.
12. Вычислите произведение сумм положительных и отрицательных элементов массива A[1..5].
13. В данной последовательности целых чисел найдите количество различных чётных чисел.
14. Напишите программу замены отрицательных элементов массива Y[1..n] их квадратами.
15. Напишите программу замены положительных элементов массива Y[1..n] на 0.

# Лабораторная работа №5. Команды передачи управления

Вопросы, необходимые рассмотреть при подготовке к лабораторной работе:

Команды безусловной передачи управления:

* команда безусловного перехода;
* вызов процедуры и возврата из процедуры;
* вызов программных прерываний и возврат из программных прерываний.

2. Команды условной передачи управления:

* команды перехода по результату команды сравнения;
* команды перехода по состоянию определённого флага;
* команды перехода по содержимому регистра CX.

3. Команды управления циклом:

* команда организации цикла со счётчиком CX;
* команда организации цикла со счётчиком CX, с возможностью досрочного выхода из цикла по дополнительному условию.

Семантика команд условного и безусловного переходов содержится в следующей таблице:

|  |  |  |  |
| --- | --- | --- | --- |
| Команды условного перехода | | | |
| Мнемокод команды | | Семантика | Условие перехода |
| Команды, используемые после команды сравнения | | | |
| Типы операндов:  любые   числа | | | |
| JE | Перейти, если равно | | ZF=1 |
| JNE | Перейти, если не равно | | ZF=0 |
| Типы операндов:  числа без знака | | | |
| JB/JNAE | Перейти, если меньше | | CF=1 |
| JBE/JNA | Перейти, если меньше или равно | | CF=1 или ZF=1 |
| JA/JNBE | Перейти, если больше | | CF=0 и ZF=0 |
| JAE/JNB | Перейти, если больше или равно | | CF=0 |
| Типы операндов: числа со знаком | | | |
| JL/JNGE | Перейти, если меньше | | SF≠OF |
| JLE/JNG | Перейти, если меньше или равно | | ZF=1 или SF≠ OF |
| JG/JNLE | Перейти, если больше | | ZF=0 и SF=OF |
| JGE/JNL | Перейти, если больше или равно | | SF=OF |
| Команды условного перехода без использования команды сравнения | | | |
| JC | Перейти, если есть перенос | | CF=1 |
| JNC | Перейти, если нет переноса | | CF=0 |
| JZ | Перейти, если нуль | | ZF=1 |
| JNZ | Перейти, если не нуль | | ZF=0 |
| JS | Перейти, если знаковый бит равен1 | | SF=1 |
| JNS | Перейти, если знаковый ряд нулевой | | SF=0 |
| JO | Перейти, если переполнение | | OF=1 |
| JNO | Перейти, если нет переполнения | | OF=0 |
| JP | Перейти, если есть чётность | | PF=1 |
| JNP | Перейти, если нет чётности | | PF=0 |
| Команды, условием перехода которой является состояние регистра CX | | | |
| JCXZ | Перейти, если содержимое CX | | CX=0 |
| JECXZ | Перейти, если содержимое ECX | | ECX=0 |

|  |  |  |
| --- | --- | --- |
| Команда безусловного перехода | | |
| JMP | Безусловный переход: ***адрес\_перехода*.** |  |

**Демонстрационные примеры**

Пример1. Программа, определяющая большее из двух чисел. Результат помещается в регистр CX.

code segment

org 100h

assume cs:code

begin:

mov ax,-2

mov bx,-1

cmp ax,bx

jg Метка\_1

mov cx,bx

jmp Метка\_2

Метка\_1: mov cx,ax

Метка\_2:

mov ax,4ch

int 21h

code ends

end begin

   Замечание.

  Обратите внимание на то, что данные числа могут быть как со знаком, так и без него. Поэтому в программе использовалась команда условного перехода, операндами которой могут быть числа со знаком.

   Пример2. Программа, определяющая большее из двух чисел для беззнаковых данных с использованием арифметической команды.

code segment

org 100h

assume cs:code

begin:

mov ax,65535

mov bx,1

mov cx,0

sub ax,bx

ja Метка\_1

mov cx,bx

jmp Метка\_2

Метка\_1: mov cx,ax

add cx,bx

Метка\_2:

mov ax,4ch

int 21h

code ends

end begin 2

 Пример 3. Пусть A, B, C - беззнаковые байтовые переменные. Вычислить С=А2+В. Если ответ превосходит размер байта, тогда надо передать управление на метку ERROR.

code segment

org 100h

assume cs:code

begin:

mov al,98h

mov bl,30h

mul al

jc ERROR ;Если есть перенос (CF=1),то переход на метку ERROR

add al,bl

jc ERROR ; Если есть перенос (CF=1), то переход на метку ERROR

mov cl,al

ERROR:

mov ax,4ch

int 21h

code ends

end begin

Пример 4. Демонстрация вычисления целой части квадратного корня из натурального числа n.

 Алгоритм решения задачи состоит в подсчёте количества нечётных чисел, последовательно вычитаемых из числа n с получением неотрицательного результата.

Например: n=9

9-1=8 -\

8-3=5 3 вычитания

5-5=0 -/

code segment

org 100h

assume cs:code

begin: mov ax,25

mov dx,ax

mov cx,1

mov bx,0

cmp ax,0

jg Sqrt

jmp Fin

Sqrt: cmp ax,0

jz Fin

sub ax,cx

cmp ax,0

jl Fin

inc bx

inc cx

inc cx

jmp Sqrt

Konec:

mov ax,dx

ret

mov ax,4ch

int 21h

code ends

end begin

  Пример 5. Демонстрация вычисления 5 последовательных значений функции y=x+C, если начальное значение x равно 1, шаг равен 1, а C - постоянная.

code segment

org 100h

assume cs:code

begin: mov bx,01h

mov cx,04h

jmp Uslovie

Nachalo: mov ax,cx

add ax,bx

mov dx,ax

inc bx

Uslovie: mov ax,bx

cmp ax,06h

jb Nachalo

ret

mov ax,4ch

int 21h

code ends

end begin

 Пример 6. Демонстрация нахождения суммы первых 5 членов натурального ряда.

   IВариант

code segment

org 100h

assume cs:code

begin: mov ax,05h ax:=5;

jz Konec

mov cx, ax cx:=5;

mov ax,0 ax:=0;

Nachalo: add ax,cx

loop Nachalo

ret

Konec:    ax:=ax+i;

mov ax,4ch

int 21h

code ends

end begin

 II Вариант

code segment

org 100h

assume cs:code

begin: mov ax,05h

jz Fin ax:=5;

mov cx,ax cx:=5;

dec cx cx:=cx-1;

Repeat: add ax,cx

LOOP Repeat

Fin: ret ax:=ax+i;

mov ax,4ch

int 21h

code ends

end begin

   Пример 7. Демонстрация организации записи в регистр BL наименьшего числа из отрезка [2,k], на которое не делится число n (n и k - байтовые переменные, 0k<n), или 0 - если такого числа нет.

Для этого мы будем последовательно делить n на числа из отрезка (2, 3,...,k) и сравнивать остатки от деления с 0 до тех пор, пока не найдется ненулевой остаток либо не будут исчерпаны все числа отрезка.

code segment

org 100h

assume cs:code

begin:

mov dl,3 ;

mov dh,0 ; Загрузили в регистр DX число N=3

mov cl,6 ;

mov ch,0 ; Загрузили в регистр CX число K=6

dec cx ; Счетчик цикла (CX = K-1)

mov bl,1 ;

M1: inc bl ; Очередное число из отрезка

mov ax,dx ;

div bl ; AH = n mod (BL)

cmp ah,0 ; Остаток равен 0?

LOOPNE M1 ; Цикл СХ раз и пока остаток не равен 0

je M2 ; mod=0 ─> выход на метку М2

mov bl,0 ; Искомое число (mod<>0) отсутствует

M2: ret

mov ax,4ch

int 21h

code ends

end begin

   Пример 8. Программа вычисления n!, использующая обращение к подпрограмме, выполняющей умножение двух чисел посредством сложения чисел.

code segment

org 100h

assume cs:code

begin: mov al,число\_N

mov bl,al

dec bl

Proverka:

dec bl

jle Konec

call Procedure

jmp Proverka

Procedure:

mov dl,al

mov dh,ah

mov cl,bl

mov ch,bh

Summa:

add al,dl

adc ah,dh

dec cl

jnz Summa

ret

Konec:

ret

mov ax,4ch

int 21h

code ends

end begin

**Задачи для самостоятельного решения**

1. Поместите в регистр АХ максимум из двух чисел, лежащих в регистрах АХ и ВХ.
2. Поместите в регистр АХ минимум из трёх чисел, лежащих в регистрах АХ, ВХ и СХ.
3. Найдите наибольшее из трех чисел А, В и С.
4. Дано целое число. Если оно чётное, то записать в регистр DI значение FFFFh, если нечётное - FF00h.
5. Напишите программу, реализующую алгоритм нахождения модуля числа.
6. Даны два числа в регистрах АХ и ВХ. Поменяйте их местами, если они оба больше 10h, и уменьшите их на 10h в противном случа
7. По данным сторонам прямоугольника и квадрата вычислите их площади и периметры. Если площадь и периметр квадрата больше площади и периметра прямоугольника, то в регистр AL поместите 1, в противном случае поместите (-1).
8. Проверьте, делится ли без остатка целое число A на целое число B
9. Запишите в регистр BL наименьшее число из отрезка [2,k], на которое делится число n (k и n - байтовые переменные), или запишите 0, если такого числа нет.
10. Организуйте цикл для подсчёта суммы первых 20 натуральных чисел.
11. Организуйте цикл для подсчёта суммы первых 20 чётных натуральных чисел.
12. Организуйте цикл для подсчёта суммы первых 20 нечётных натуральных чисел.
13. Организуйте цикл для подсчёта суммы 20 первых натуральных чисел вида 2(3n-1)
14. Вычислите произведение m членов арифметической прогрессии, если известны значения первого члена а 1  и разность d.
15. Вычислите значение n! для n=7.
16. Найдите номер данного целого числа в заданном отрезке чисел.
17. Напишите алгоритм умножения двух чисел без знака без использования команд умножения.
18. Напишите алгоритм деления двух чисел без знака без использования команд деления.
19. Найдите k-ю частичную сумму ряда с членами вида n(n-1)
20. Дан четырёхзначный номер года. Определить номер столетия, например, при 1492 ответом должно быть "15", а при 1812 - "19". Учесть, что началом века считается первый, а не нулевой год: так, при вводе 2000 ответом будет "20".
21. Супермаркет доставляет продукты питания на дом. Эта услуга оплачивается следующим образом: если сумма покупки не превышает 600 рублей, то доставка обойдётся покупателю в 250 рублей; если сумма покупки лежит в пределах 600...990 рублей, то доставка стоит 150 рублей; если сумма покупки - более 990 рублей, то доставка вообще бесплатная. Написать программу, вводящую сумму покупки и вычисляющую:

а) стоимость доставки; б) стоимость покупки вместе с доставкой.

1. Дано текущее время (M S), где M - это минуты, а S - секунды. Написать программу, определяющую время спустя 1 секунду.
2. Дано текущее время (H M S), где H - это часы, M - это минуты, а S - секунды. Написать программу, определяющую время спустя 1 секунду.
3. Дано целое число в пределах 0..23, представляющее собой московское время с точностью до часа (без минут). Написать программу, вычисляющую иркутское время и выводящую его на экран. Разница во времени между Москвой и Иркутском u1089 составляет +5 часов. Решить эту задачу: (а) с применением ветвления; (б) без него, но с использованием команды *MOD*.
4. В задачах этого раздела под термином «массив» понимается последовательность данных, размещённая каким-либо способом в сегменте. Результаты подсчётов элементов следует получить в одном из регистров процессора. Второй массив (в задачах c,f,g) можно разместить как в сегменте DS, так и в сегменте ES.
   1. Дан массив байтов. Подсчитать количество элементов, равных трем.
   2. Дан массив байтов. Подсчитать количество элементов, кратных трем (команда DIV).
   3. Дано два массива байтов, один заполнен числами. Заполнить второй массив следующим образом: все элементы первого массива, стоящие на нечетных местах, возвести в квадрат, стоящие на четных местах – в куб.
   4. Дан массив: а) байтов, б) слов. Подсчитать количество четных элементов.
   5. Дан массив: а) байтов, б) слов. Подсчитать количество элементов, делящихся на 10 и не делящихся на 3.
   6. Дан массив байтов. Сформировать другой массив, в который нечетные элементы из первого массива переписать без изменений, а четные – увеличить на 1.
   7. Дан массив, заполненный двузначными числами. Сформировать другой массив, в котором каждый элемент представляет собой сумму цифр соответствующего числа первого массива.

# 

# Лабораторная работа №6. Прерывания.

Вопросы, необходимые рассмотреть при подготовке к лабораторной работе:

1. Прерывания. Функции DOS.
2. Ввод на уровне BIOS.Управление видео на уровне BIOS/

**Демонстрационные примеры**

Пример1. Программа вывода на экран строки.

; ---------------------------------

%TITLE "Вывод строки"

IDEAL

MODEL small

DATASEG

exCode DB 0

myString DB "Welcome to my program","$"

CODESEG

Start:

mov ax, @data

mov ds, ax

mov dx, OFFSET myString ; Строка размещена по адресу DS:DX

mov ah, 09 ; Функция, печатающая строку

int 21h ; Вызов функции

Exit:

mov ah,04Ch ; Функция DOS: выход из программы

mov al,[exCode] ; Возврат значения кода ошибки

int 21h ; Вызов DOS. Останов программы

END Start ; Конец программы / точка вход

 Пример 2. Демонстрация программы, осуществляющей вывод строки в определённом месте экрана

ASSUME cs:code, ds:data

STK SEGMENT STACK ; Открытие сегмента стека

db 256 dup (?) ; Выделение под стек 256 байтов

STK ENDS ; Закрытие сегмента стека

DATA SEGMENT PARA ; Сегмент данных начинается по адресу, кратному 16

mes DB 'Привет!$' ; Выводимое слово

DATA ENDS

CODE SEGMENT

Begin:

mov ax,data

mov ds,ax

; ----------

mov ax,0600h ; Запрос на очистку экрана

mov bh,07h ; Нормальный атрибут (чёрно-белый)

mov cx,0000h ; Верхняя левая позиция очищаемой области

mov dx,184fh ; Нижняя правая позиция очищаемой области

int 10h ; Передача управления в BIOS

; -------

mov ah,2h ; Функция установления положения курсора

mov bh,0 ; Указатель на начало видеостраницы

mov dh,12 ; Номер строки

mov dl,39 ; Номер столбца

int 10h ; Видеосервис (вызов функции)

; --------

mov ah,09h ; Вывод строки на экран

mov dx,offset mes

int 21h

; --------

mov ah,08h ; Функция ожидания нажатия любой клавиши

int 21h ; Вызов DOS: код нажатой клавиши - в AL

; ----------

mov ax,4C00h

int 21h

CODE ENDS

END Begin

Пример 3. Ввести строку с клавиатуры, посчитать, сколько и каких десятичных цифр имеется во введенной строке, посчитанные значения вывести на экран.

data segment

count db 10 dup (0) ; счетчик количества цифр

cifr db ‘0123456789abcdef’ ; таблица преобразования цифр

in\_str db 80, ?, 82 dup (?) ; буфер ввода

out\_str db 0dh, 0ah, ?,’-‘,?, ?,’$’ ; буфер вывода

data ends

code segment

assume cs:code, ds:data

START: mov ax, data

mov ds, ax ; Загрузить сегментный адрес данных

; Ввод строки

lea dx, in\_str

mov ah, 0ah

int 21h

; Обработка

xor ah, ah ; обнуление старшего байта AX

lea bx, in\_str+2 ; адрес начала введённой строки

xor cx, cx

mov cl, in\_str+1 ; количество введенных символов

ВВ: mov al, [bx] ; очередной символ строки

cmp al,’0’ ; код символа меньше чем код нуля?

jb NC ; да, не цифра

cmp al, ‘9’ ; код символа больше чем код девяти

ja NC ; да, не цифра

; символ - десятичная цифра

sub al, ‘0’ ; преобразуем ASCII код в число

mov si, ax ; индекс в массиве счетчиков COUNT

inc count[si] ; увеличиваем счетчик цифр

NC: inc bx ; получить очередной символ строки

loop BB

; Вывод результатов

mov cx, 10

lea bx, cifr ; адрес таблицы преобразования цифр в ASCII

xor si, si ; номер выводимой цифры

OUT: mov al, ‘0’

add ax, si ; ASCII код очередной цифры с номером в si

mov out\_str+2, al ; в буфер вывода

mov dl, count[si] ; читать количество цифр с номером в si

mov al, dl

push cx ;временное сохранение cx

mov cl, 4 ; сдвиг на четыре

shr al, cl ; выделить старшую цифру

xlat ; ASCII старшей цифры в al

pop cx ; восстановление cx

mov out\_str+4, al ; в буфер вывода

mov al, dl ; восстановить количество цифр в al

and al,00001111b ; выделить младшую цифру

xlat ; ASCII младшей цифры в al

mov out\_str+5, al ; в буфер вывода

lea dx, out\_str ; подготовка к выводу строки

mov ah, 09h ; номер функции

int 21h ; вывод строки

inc si ; Счетчик очередной цифры

loop OUT

QUIT: mov ax, 4C00h ; Код завершения 0

int 21h ; Выход в DOS

code ends

end START

Пример 4. Написать программу на ассемблере, выводящую в текущее положение курсора символ @. Следующий символ @ выводить в позицию выше, ниже, левее или правее текущего символа, в зависимости от нажатия клавиш “8”, “2”, “4”, “6” на цифровой клавиатуре. Вывод осуществлять непрерывно с некоторой задержкой. Нажатие клавиши “0” завершает выполнение программы.

• Примечание. В программе необходимо вести отсчёт времени для задержки вывода символа @. Для этого необходимо изменить подпрограмму обработки прерывания от таймера 08h. Так как эта подпрограмма выполняет важные операции по управлению компьютером, для полу-

чения временного интервала используется прерывание 1Ch. Это прерывание вызывается из подпрограммы обработки прерывания 08h и содержит только команду **iret**. Предназначено оно специально для пользовательских программ, которым необходимо следить за интервалами отсчёта таймера.

data segment

DIRECT db 1 ; направление перемещения

EXIT db 0 ; признак завершения программы (не 0)

SYM db "@" ; символ, выводимый на экран

ATRIBUT1 db 14 ; атрибут символа (жёлтый)

ATRIBUT2 db 10 ; атрибут символа (зелёный)

POS dw 3840 ; позиция начального вывода символа

OLD\_CS dw ? ; адрес сегмента старого вектора 1Сh

OLD\_IP dw ? ; адрес смещения старого вектора 1Сh

data ends

code segment

assume cs:code, ds:data

; Подпрограмма обработки прерывания 1Сh

NEW\_1C proc far

push ax ; сохранить все регистры

push bx

push cx

push dx

push ds

push es

mov ax, DATA ; установить ds на сегмент данных

mov ds, ax ; основной программы

mov ax, 40h ; установить es на

mov es, ax ; сегмент данных bios

mov ax, es:[1ch]

mov bx, es:[1ah]

cmp bx , ax

jne m5

jmp back

m5: mov al, es:[bx]

mov es:[1ch], bx

cmp al, 30h

jnz m1

mov EXIT, 1

jmp back

m1: cmp al, 35h

jne m6

mov dl, ATRIBUT1

mov dh, ATRIBUT2

mov ATRIBUT1, dh

mov ATRIBUT2, dl

jmp back

m6: cmp al, 38h ; стрелка вверх

jz m2

cmp al, 32h ; стрелка вниз

jz m3

cmp al, 34h ; стрелка влево

jz m4

cmp al, 36h ; стрелка вправо

jnz back ; неиспользуемая клавиша

mov DIRECT, 3

jmp back

m2: mov DIRECT, 1

jmp back

m3: mov DIRECT, 4

jmp back

m4: mov DIRECT, 2

back: pop es

pop ds

pop dx

pop cx

pop bx

pop ax

iret

NEW\_1C endp

; Подпрограмма очистки экрана

CLS proc near

push cx

push ax

push si

xor si, si

mov ah, 7

mov dl, ' '

mov cx, 2000

CL1: mov es:[si], ax

inc si

inc si

loop CL1

pop si

pop ax

pop cx

ret

CLS endp

; Подпрограмма задержки

DELAY proc near

push cx

mov cx, 100

d12: push cx

xor cx,cx

d11: nop

loop d11

pop cx

loop d12

pop cx

ret

DELAY endp

; Подпрограмма вывода символа с заданным атрибутом

OUT\_SYMBOL proc near

push ax

push bx

mov al, SYM

mov ah, ATRIBUT1

mov bx, POS

call DELAY

mov es:[bx], ax

pop bx

pop ax

ret

OUT\_SYMBOL endp

; Основная программа

START: mov ax, DATA

mov ds, ax

; чтение вектора прерывания

mov ah, 35h

mov al, 1Ch

int 21h

mov OLD\_IP, bx

mov OLD\_CS, es

; установка вектора прерывания

push ds

mov dx, offset NEW\_1C

mov ax, seg NEW\_1C

mov ds, ax

mov ah, 25h

mov al, 1Ch

int 21h

pop ds

mov ax, 0B800h

mov es, ax

call CLS

call DELAY

p1: cmp EXIT, 0

jne quit

cmp DIRECT, 1

jz p2

cmp DIRECT, 2

jz p3

cmp DIRECT, 3

jz p4

mov ax, POS

add ax,160

cmp ax, 3999

jg p1

mov POS, ax

call OUT\_SYMBOL

jmp p1

p2: mov ax, POS

sub ax, 160

jl p1

mov POS, ax

call OUT\_SYMBOL

jmp p1

p3: mov ax, POS

sub ax, 2

jl p1

mov POS, ax

call OUT\_SYMBOL

jmp p1

p4: mov ax, POS

add ax, 2

jg p1

mov POS, ax

call OUT\_SYMBOL

jmp p1

quit: call CLS

mov dx, OLD\_IP

mov ax, OLD\_CS

mov ds, ax

mov ah, 25h

mov al, 1Ch

int 21h

mov ax, 4c00h

int 21h

CODE ends

end START

**Задачи для самостоятельного решения**

1. Ввести с клавиатуры строку. Сжать строку, т.е. удалить пробелы и табуляции. Вывести результаты на экран.
2. Ввести с клавиатуры строку. Преобразовать все малые буквы в большие. Вывести результаты на экран.
3. Ввести с клавиатуры строку. Посчитать количество слов в строке. Определить, что является разделителем слов. Вывести результаты на экран.
4. Ввести с клавиатуры строку. Ввести с клавиатуры коротенькую строку - шаблон. Найти шаблон во введенной строке. Вывести на экран “ДА”, если
5. шаблон есть и “НЕТ”, если нет.
6. Ввести с клавиатуры две строки. Сравнить их. Вывести на экран “ДА”, если они равны и “НЕТ”, если нет.
7. Ввести с клавиатуры строку. Если она длиннее некоторой заданной величины, то обрезать, если короче растянуть, вставив нужное число пробелов между словами. Вывести результаты на экран.
8. Ввести с клавиатуры строку, состоящую из нескольких слов. Вывести каждое слово на экран в отдельной строке, т.е. выдать слова в столбик.
9. Ввести с клавиатуры строку. Переставить в ней символы, поменяв местами первый символ с последним, второй с предпоследним и т.д. Вывести результаты на экран.
10. Ввести с клавиатуры две строки. Сравнить их. Вывести на экран номер начала второй строки в первой.
11. Ввести с клавиатуры строку, содержащую несколько точек. Преобразовать строку, чтобы после каждой точки был пробел, и следующая буква после точки были заглавная. Вывести результаты на экран.
12. Ввести с клавиатуры строку, содержащую несколько слов, разделенных пробелом. Переставить в ней слова, поменяв местами первое слово с последним, второе с предпоследним и т.д. Вывести результаты на экран.
13. Ввести с клавиатуры строку, состоящую из нескольких слов. Вывести каждое слово на экран в отдельной строке лесенкой, т.е. выдать каждое слово в столбик, с фиксированным сдвигом относительно начала предыдущего.
14. Ввести с клавиатуры строку. Преобразовать все буквы в числа. Построить криптограмму (вместо букв вывести на экран соответствующие им числа).
15. Ввести с клавиатуры строку, состоящую из нескольких букв. Заменить каждую букву в строке на другую букву, следующую за данной буквой по алфавиту. Вывести результаты на экран.
16. Ввести с клавиатуры строку и строку из двух чисел. Первое число указывает начало подстроки для ввода на экран, второе количество символов из первой строки, которое необходимо вывести на экран. Ввести с клавиатуры две строки. Сравнить их. Вывести на экран номер начала первой строки во второй. Ввести с клавиатуры строку. Ввести с клавиатуры коротенькую строку - шаблон. Найти шаблон во введенной строке. Вывести на экран “ДА”, если шаблон есть и “НЕТ”, если нет.
17. Ввести с клавиатуры две строки. Сравнить их. Вывести на экран какая из строк больше и насколько.
18. Ввести с клавиатуры строку и некоторое число. Если строка длиннее заданного числа, то обрезать, если короче растянуть, вставив нужное число пробелов между словами. Вывести результаты на экран.
19. Ввести с клавиатуры строку, состоящую из нескольких слов. Вывести каждое слово на экран в отдельной строке, со смещением влево на одно знакоместо по отношению к предыдущей строке.
20. Ввести с клавиатуры строку. Посчитать в ней количество запятых. Вывести результаты на экран.
21. Ввести с клавиатуры две строки. Вывести на экран все символы, которые содержатся в обеих строках.
22. Ввести с клавиатуры строку, содержащую несколько точек. Преобразовать строку, чтобы после каждой точки был пробел, и следующая буква после точки были заглавная. Вывести результаты на экран.
23. Ввести с клавиатуры строку, содержащую несколько слов, разделенных пробелом. Переставить в ней слова, поменяв местами первое слово с последним, второе с предпоследним и т.д. Вывести результаты на экран.
24. Ввести с клавиатуры строку, состоящую из нескольких слов. Вывести каждое слово на экран в отдельной строке лесенкой, т.е. выдать каждое слово в столбик, с заданным сдвигом относительно начала предыдущего.
25. Ввести с клавиатуры строку, состоящую из нескольких букв. Заменить каждую букву в строке на другую букву, следующую за данной буквой через заданное число символов по алфавиту. Вывести результаты на экран.

# Лабораторная работа №7. Модульное программирование. Подпрограммы

Вопросы, необходимые рассмотреть при подготовке к лабораторной работе:

1. Концепция модульного программирования.
2. Понятие "процедура".Описание процедуры.
3. Команда вызова процедуры. Команда возврата управления.
4. Организация интерфейса с процедурой

- передача аргументов через регистры

- передача аргументов через стек.

- передача аргументов через общую область памяти

**Демонстрационные примеры**

Пример 1. Демонстрация программы, в которой реализован способ передачи аргументов в процедуру через регистры на примере вычисления значения выражения без использования команды умножения:

a+3\*(b+c), где a, b, c - целые однобайтовые числа со знаком

MASM

MODEL small

.STACK 256

; --------

.DATA

Prigl1 DB "Введите значение a: $"

Prigl2 DB "Введите значение b: $"

Prigl3 DB "Введите значение c: $"

Prigl4 DB "Значение выражения a+3\*(b+c): $"

; --------------------------------------------

.CODE

Start:

mov ax,@data

mov ds,ax

; --------------------------------

; Ввод значений переменных a,b и c

; ----------------------------------------------------------

lea dx,Prigl1 ; Загрузка в регистр DX адреса строки Prigl1

call Print\_S ; Печать строки prigl1

; ----------

mov ax,0

call Vvod\_AL ; Ввод целого числа в регистр AL

push ax ; Сохранение (AX)=(a) в стеке

; --------

lea dx,Prigl2 ; Загрузка в регистр DX адреса строки Prigl2

call Print\_S ; Печать строки prigl2

; ----------

mov ax,0

call Vvod\_AL ; Ввод целого числа в регистр AL

mov bx,ax ; (BX)=(b)

; --------

lea dx,Prigl3 ; Загрузка в регистр DX адреса строки Prigl3

call Print\_S ; Печать строки prigl3

; ----------

mov ax,0

call Vvod\_AL ; Ввод целого числа в регистр AL

mov cx,ax ; (CX)=(c)

; -----------

lea dx,Prigl4 ; Загрузка в регистр DX адреса строки Prigl4

call Print\_S ; Печать строки prigl4

; ----------

pop ax ; Извлечение (AX)=(a) из стека

; --------------------------------------------

; Вычисление значения выражения a+3\*(b+c)

; --------------------------------------------------------

call AddReg ; Вызов процедуры AddReg: (AX)=(a)+(b)+(c)

call AddReg ; Вызов процедуры AddReg: (AX)=(AX)+(b)+(c)

call AddReg ; Вызов процедуры AddReg: (AX)=(AX)+(b)+(c)

; -----------

call Vivod\_AX ; Вывод целого числа из регистра AX на экран

; -----------

mov ah,08h ; Функция "задержки" экрана

int 21h ; Вызов DOS-функции

Exit:

mov ax,4C00h

int 21h

; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Процедура Print\_S по выводу строки на экран дисплея

; Передаётся: (DX) - адрес строки в памяти

; Возвращается: строка на экране дисплея

; Модифицируемые регистры: AX

; ----------------------------------------------

Print\_S PROC ; Открытие процедуры Print\_S

mov ah,09h ; (AH) - номер функции печати строки

int 21h ; Вызов прерывания с номером 21h

ret ; Возврат в основную программу

Print\_S ENDP ; Завершение процедуры Print\_S

; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Процедура AddReg: суммирование значений регистров AX, BX и CX

; Передаётся: (AX), (BX), (CX)

; Возвращается: (AX)=(AX)+(BX)+(CX)

; Модифицируемые регистры: AX

; ---------------------------------------------

AddReg PROC ; Открытие процедуры AddReg

add ax,bx ; (AX)=(AX)+(BX)

add ax,cx ; (AX)=(AX)+(CX)

ret ; Возврат в основную программу

AddReg ENDP ; Завершение процедуры AddReg

; -----------------------------------------------

include bibl-prc.inc

END Start ; Конец программы

Пример. Демонстрация программы, позволяющей получить значение синуса угла от 0 до 360 градусов [Скэнлон,1989,с.166]

; --------------------

MASM

MODEL small

.STACK 256

.DATA

; -----------------------

; Таблица значений синуса

; ---------------------------------------

Sine DW 0,175,349,523,698,872 ; 0- 5

DW 1045,1219,1392,1564,1736 ; 6-10

DW 1908,2079,2250,2419,2588 ; 11-15

DW 2756,2924,3090,3256,3420 ; 16-20

DW 3584,3746,3907,4067,4226 ; 21-25

DW 4384,4540,4695,4848,5000 ; 26-30

DW 5150,5299,5446,5592,5736 ; 31-35

DW 5878,6018,6157,6293,6428 ; 36-40

DW 6561,6691,6820,6947,7071 ; 41-45

DW 7193,7313,7431,7547,7660 ; 46-50

DW 7771,7880,7986,8090,8191 ; 51-55

DW 8290,8387,8480,8572,8660 ; 56-60

DW 8746,8829,8910,8988,9063 ; 61-65

DW 9135,9205,9272,9336,9397 ; 66-70

DW 9455,9511,9563,9613,9659 ; 71-75

DW 9703,9744,9781,9816,9848 ; 76-80

DW 9877,9903,9926,9945,9962 ; 81-85

DW 9976,9986,9994,9998,10000 ; 86-90

; --------------------------------------------------

Str1 DB "Введите угол (от 0 до 360 градусов) x=",'$'

Str2 DB "SIN(",'$'

Str3 DB ")=",'$'

Str4 DB "\*10^(-4)",'$'

; -----------------------

.CODE

Start:

mov ax,@data

mov ds,ax

; --------------------------------------------

; Ввод значения угла для вычисления его синуса

; --------------------------------------------

Vvod\_a:

mov ah,09h ; Функция вывода строки на экран

lea dx,Str1 ; (DX)=<Адрес\_строки\_Str1>

int 21h ; Вызов функции

; ----------

call Vvod\_AX ; Ввод значения угла (в градусах)

cmp ax,0 ; Сравнение (AX) с 0

jb Vvod\_a ; Если (AX)<0, переход по метке

cmp ax,360 ; Сравнение (AX) с 360

ja Vvod\_a ; Если (AX)>360, переход по метке

; -----------------------------------------------

; Оформление предложения для написания ответа

; -------------------------------------------

push ax ; Сохранение (AX) в стеке

mov ah,09h ; Функция вывода строки на экран

lea dx,Str2 ; (DX)=<Адрес\_строки\_Str2>

int 21h ; Вызов функции

pop ax ; Возврат (AX) их стека

; -----------

call Vivod\_AX ; Печать (AX)

; -----------

push ax ; Сохранение (AX) в стеке

mov ah,09h ; Функция вывода строки на экран

lea dx,Str3 ; (DX)=<Адрес\_строки\_Str3>

int 21h ; Вызов функции

pop ax ; Возврат (AX) их стека

; -----------------------------------------------------

; Вычисление синуса заданного угла и вывод его на экран

; -----------------------------------------------------

call Find\_Sine ; Процедура по вычислению синуса угла

mov ax,bx

call Vivod\_AX ; Вывод целого числа из регистра AX на экран

; -----------

mov ah,09h ; Функция вывода строки на экран

lea dx,Str4 ; (DX)=<Адрес\_строки\_Str4>

int 21h ; Вызов функции

; --------

mov ah,08h ;

int 21h ; "Задержка" экрана

Exit:

mov ax,4C00h ;

int 21h ; Выход из программы

; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Процедура Find\_Sine: нахождение значения синуса угла

; Передаётся: (AX)=<Значение\_угла\_в\_градусах>

; Возвращается: (BX)=<Значение\_синуса\_заданного\_угла>

; Модифицируемые регистры: BX

; ----------------------------------------------------

; В зависимости от значения угла в процедуре осуществляются следующие ;преобразования:

;

; квадрант │диапазон угла │преобразования

;

; I 0<=x<= 90 sin(x) синусы углов I квадранта

; II 91<=x<=180 sin(180-x) "зеркальное отражение"

; синусов углов I квадранта

; III 181<=x<=270 -sin(x-180) "негативное обращение"

; синусов углов II квадранта

; IV 271<=x<=360 -sin(360-x) "негативное обращение и

; зеркальное отражение"

синусов углов I квадранта

; -----------------------------------------------------------

Find\_Sine PROC ; Открытие процедуры Find\_Sine

push ax ; Сохранение (AX) в стеке

push cx ; Сохранение (CX) в стеке

; -------

sub cx,cx ; (CX)=0 - флаг по определению знака числа

cmp ax,181 ; Сравнение значения угла x с числом 181

jl Sin\_Pos ; Переход по метке, если угол < 181

mov cx,1 ; (CX)=1

sub ax,180 ; (AX)=(x)-180, т.к. (x)>=181

Sin\_Pos:

cmp ax,91 ; Сравнение значения угла x с числом 91

jl Get\_Sin ; Переход по метке, если угол < 91

neg ax ; -\

add ax,180 ; -/ (AX)=180-(x), т.к. x>=91

; ----------------------------------------

; Вычисление синуса угла x

; ------------------------

Get\_Sin:

mov bx,ax ; (BX)=(AX)

shl bx,1 ; (BX)=2\*(BX)

mov bx,[sine+bx] ; (BX)=<Значение\_синуса\_из\_таблицы>

cmp cx,0 ; Сравнение (CX) с нулём

je Ext ; Переход, если (CX)=0

mov cx,0 ; -\

xchg bx,cx ; Изменение знака числа

sub bx,cx ; -/

Ext:

pop cx ; Возврат (CX) их стека

pop ax ; Возврат (AX) их стека

ret ; Возврат в программу

Find\_Sine ENDP ; Завершение процедуры Find\_Sine

; --------------------------------------------------

include bibl-prc.inc

END Start

Пример 3 ( файл \*-03.asm ).

; Демонстрация программы, в которой реализован способ передачи аргументов в процедуру через общую область памяти на примере вывода строки на экран дисплея, сформированной в модуле 09-031.asm

; --------------------------------------

; Этапы получения исполняемой программы:

; tasm.exe 09-03.asm => 09-03.obj

; tasm.exe 09-031.asm => 09-031.obj

; tlink.exe 09-03.obj 09-031.obj => 09-03.exe

; -------------------------------------------

MASM

MODEL small

.STACK 256

; ---------------------------------------------

SEGMENT Var PARA COMMON ; Общий сегмент Var

mas DB 6 DUP (" "),"$" ; Область памяти в 6 байтов

ENDS Var

; ----------------------------------------

.CODE ; Сегмент кода

EXTRN Vvod:proc, Print:proc ; Подключение процедур из модуля 9-031.obj

Start:

mov ax,@data ; (AX)=<Адрес\_сегмента\_данных>

mov ds,ax ; (DS)=(AX)

mov ax,Var ; (AX)=<Адрес\_сегмента\_Var>

mov es,ax ; (ES)=(AX)

; --------

call Print ; Вызов процедуры по печати строки

call Vvod ; Вызов процедуры по формированию строки

push es ; -\ Установка указателя DS на сегмент Var

pop ds ; -/ (DS)=(ES)

; --------

mov ah,09h ; Функция вывода строки на экран

lea dx,mas ; (DX)=<Адрес\_строки\_mas>

int 21h ; Вызов функции

; --------

mov ah,08h ; Функция ожидания нажатия любой клавиши

int 21h ; (AL)=<Код\_нажатой\_клавиши>

Exit:

mov ax,4C00h

int 21h

END Start

 ( файл \*-031.asm ).

; Демонстрация программы, в которой описана процедура по заполнению строки памяти и реализована организация передачи данной строки через общую область памяти в модуль 09-03.asm, откуда была вызвана эта процедура

; -------------------------------------------------

MASM

MODEL small

.STACK 256

; ----------------------------------------------------

SEGMENT All PARA PUBLIC "DATA" ; Сегмент данных

Str1 DB "Переданная строка: ",'$'

ENDS All

; ---------------------

SEGMENT Var PARA COMMON ; Общий сегмент Var

mas1 DB 6 DUP (" "),"$" ; Область памяти в 6 байтов

ENDS Var

; ----------------------------------------

.CODE ; Сегмент кода

PUBLIC Vvod, Print ; Объявление имён модулей, видимых в других модулях

ASSUME es:Var, ds:All ; (ES)=<Адрес\_сегмента\_Var>

; (DS)=<Адрес\_сегмента\_All>

; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Процедура печати строки из сегмента данных

; ----------------------------------------------------

Print PROC ; Открытие процедуры Print

mov ah,09h ; Функция вывода строки на экран

lea dx,Str1 ; (DX)=<Адрес\_строки\_Str1>

int 21h ; Вызов функции DOS с номером 9

ret ; Возврат в программу

Print ENDP ; Завершение процедуры Print

; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Процедура заполнение строки mas1 в сегменте Var символами

; ---------------------------------------------------------

Vvod PROC ; Открытие процедуры Vvod

mov si,0 ; (SI)=0 - смещение по строке mas1

mov [mas1+si],"о" ; mas1[0]:='о'

inc si ; (SI)=1 - смещение по строке mas1

mov [mas1+si],"б" ; mas1[1]:='б'

inc si ; (SI)=2 - смещение по строке mas1 mov [mas1+si],"щ" ; mas1[2]:='щ'

inc si ; (SI)=3 - смещение по строке mas1

mov [mas1+si],"а" ; mas1[3]:='а'

inc si ; (SI)=4 - смещение по строке mas1

mov [mas1+si],"я" ; mas1[4]:='я'

inc si ; (SI)=5 - смещение по строке mas1

mov [mas1+si],"!" ; mas1[5]:='!'

ret ; Возврат в программу

Vvod ENDP ; Завершение процедуры Vvod

END

 Замечание.

  Для выполнения программы (\*-03.asm) необходимо подключить модуль

\*-031.asm следующим образом:

tasm /zi \*-03.asm

tasm /zi \*-031.asm

tlink /v \*-03.obj \*-031.obj

\*-03.exe

Для выполнения программы в TDBG нужно набрать в командной строке

td \*-03.exe

 Пример 5. Демонстрация программы, в которой реализован способ передачи аргументов в процедуру через стек на примере вычисления суммы пяти целых чисел, введённых с клавиатуры

; ----------------------------------------------------

MASM

MODEL small

.STACK 256

; --------

.DATA

Prigl1 DB "Введите последовательность из пяти целых чисел: $"

Prigl2 DB "Сумма введённых чисел: $"

Error DB 0Ah,0Dh,"Переполнение: ",'$'

; --------------------------------------

.CODE

Start:

mov ax,@data

mov ds,ax

; ----------------------------------

; Ввод пяти целых чисел с клавиатуры

; ----------------------------------------------

mov ah,09h ; Функция вывода строки на экран

lea dx,Prigl1 ; Загрузка в регистр DX адреса строки Prigl1

int 21h

; ------

mov cx,5 ; (CX)=5 - счётчик цикла

mov bx,cx ; (BX)=(CX) для сохранения количества чисел

Vvod\_ch:

call Vvod\_AX ; Ввод числа в регистр AX

push ax ; Помещение (AX) в стек

loop Vvod\_ch ; (CX)=(CX)-1, переход по метке Vvod\_ch

; -----------------------------------------------------

; Вычисление суммы пяти целых чисел

; ----------------------------------------------

mov ah,09h ; Функция вывода строки на экран

lea dx,Prigl2 ; Загрузка в регистр DX адреса строки Prigl2

int 21h

; -----

push bx ; Помещение (BX)=5 - счётчик цикла в стек

call Sum ; Вызов процедуры Sum

; -----------

call Vivod\_AX ; Вывод целого числа из регистра AX на экран

; -----------

mov ah,08h

int 21h

Exit:

mov ax,4C00h

int 21h

; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Процедура Sum: суммирование пяти целых чисел,

; находящихся в стеке

; Передаётся: 5 чисел и их количество

; Возвращается: (AX)=<Сумма\_пяти\_целых\_чисел\_стека>

; Модифицируемые регистры: AX, BX, CX, DX, SI

; -------------------------------------------

Sum PROC ; Открытие процедуры Sum

pop si ; Сохранение адреса возврата из процедуры

pop cx ; (CX)=5 - счётчик цикла

pop ax ; (AX)=<Первое\_число>

dec cx ; (CX)=4 - счётчик цикла

Vivod\_ch:

pop bx ; (BX)=<Очередное\_число\_стека>

add ax,bx ; (AX)=(AX)+(BX)

jo Er ; Если переполнение AX, то переход по метке

loop Vivod\_ch ; (CX)=(CX)-1, переход по метке Vivod\_ch

jmp Home ; Переход по метке на выход из процедуры

Er:

mov ah,09h ; Функция вывода строки на экран

lea dx,Error ; Загрузка в регистр DX адреса строки Error

int 21h

Home:

push si ; Возвращение адреса возврата из процедуры

ret ; Возврат в программу

Sum ENDP ; Завершение процедуры Sum

; -------------------------------------------

include bibl-prc.inc

END Start

Пример 6. Демонстрация вывода различных заглавных букв в алфавитном порядке по заданной последовательности латинских букв

; -----------------------------------------------------

IDEAL

MODEL small

STACK 256

DATASEG

s DB 26 DUP(0)

prig1 DB "Введите строку, содержащую как строчные "

DB "так и прописные буквы",0Ah,0Dh

DB "латинского алфавита:",0Ah,0Dh,"$"

CODESEG

Start:

mov ax,@data

mov ds,ax

mov ah,09 ; Функция вывода строк

mov dx,OFFSET prig1 ; Инициализация смещения prig1

int 21h ; Вызов функции вывода строки

call WrLiters

Exit:

mov ah,08h ; Функция ожидания нажатия любой клавиши

int 21h ; Вызов DOS: код нажатой клавиши - в AL

mov ax,4C00h

int 21h

; \*\*\*\*\*\*\*\*\*\*\*

PROC WrLiters ; Ввод букв и заполнение массива s

mov ah,1 ; Функция DOS: ввод символа в al

xor bh,bh ; Обнуление старшего байта регистра BX

Vvod:

int 21h ; Вызов функции ввода символа

mov bl,al ; Копия кода введённого символа в BL

cmp bl,13 ; Проверка на конец ввода: {ENTER}

je Vivod ; Переход на вывод строки

mov [s+bx-'A'],1 ; Запомнить, что буква была

jmp Vvod ; Продолжить ввод

Vivod:

mov ah,2 ; Функция DOS: вывод символа

mov dl,10 ; ASCII-символ переход на новую строку

int 21h ; Вызов функции перевода курсора

mov cx,26 ; Длина массива

mov bx,'A' ; Помещение в BX кода символа 'A'

Print:

mov dl,bl ; В DL коды букв от 'A' до 'Z'

cmp [s+bx-'A'],1 ; Есть ли 1 в соответствующем элементе

jne Next ; Если нет, то переход к следующей букве

int 21h ; Если да, то печать этой буквы

Next:

inc bx ; Переход к следующей букве

loop Print ; Продолжить вывод

ret ; Возврат в основную программу

ENDP WrLiters

END Start

 Пример 7. Демонстрация программы, в которой реализован способ передачи аргументов в процедуру через стек на примере нахождения минимального элемента в последовательности целых чисел, введённых с клавиатуры

; -----------------------------

MASM

MODEL small

.STACK 256

; --------

.DATA

Prigl1 DB "Введите последовательность из пяти целых чисел:$"

Prigl2 DB "Минимальное число: $"

; ---------------------------------

.CODE

Start:

mov ax,@data

mov ds,ax

; ----------------------------------

; Ввод пяти целых чисел с клавиатуры

; ----------------------------------------------

mov ah,09h ; Функция вывода строки на экран

lea dx,Prigl1 ;Загрузка в регистр DX адреса строки Prigl1

int 21h

; ------

mov cx,5 ; (CX)=5 - счётчик цикла

mov bx,cx ; (BX)=(CX) для сохранения количества чисел

Vvod\_ch:

call Vvod\_AX ; Ввод числа в регистр AX

push ax ; Помещение (AX) в стек

loop Vvod\_ch ; (CX)=(CX)-1, переход по метке Vvod\_ch

; -----------------------------------------------------

; Нахождение минимального элемента в последовательности

; целых чисел

; ----------------------------------------------

mov ah,09h ; Функция вывода строки на экран

lea dx,Prigl2; Загрузка в регистр DX адреса строки Prigl2

int 21h

; -----

push bx ; Помещение (BX)=5 - счётчик цикла в стек

call Min ; Вызов процедуры Min

; -----------

call Vivod\_AX ;Вывод целого числа из регистра AX на экран

; -----------

mov ah,08h

int 21h

Exit:

mov ax,4C00h

int 21h

; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Процедура Min: нахождение минимального элемента в

; последовательности пяти целых чисел стека

; Передаётся: 5 чисел и их количество

; Возвращается: (AX)=<Минимальное\_число>

; Модифицируемые регистры: AX, CX, SI

; -------------------------------------------

Min PROC ; Открытие процедуры Min

push bp ; -\

mov bp,sp ; -/ Пролог процедуры

; -----------

mov cx,[bp+4] ; (CX)=<Количество\_чисел>

mov si,6 ; (SI)=6 - указатель на начало

; последовательности целых чисел

; ------------

mov ax,[bp+si] ; AX)=<Предполагаемое\_минимальное\_число>

Scan:

add si,2 ; (SI)=(SI)+2 - указатель на следующее

; число последовательности целых чисел

cmp ax,[bp+si] ; Сравнение (AX) со следующим числом

; последовательности целых чисел

jg Next\_ch ; Переход, если (AX) не меньшее число

; последовательности целых чисел

loop Scan ; (CX)=(CX)-1, переход по метке Scan

; -------

jmp Home ; (CX)=0, переход по метке Home

Next\_ch:

dec cx ; (CX)=(CX)-1

mov ax,[bp+si] ; AX)=<Предполагаемое\_минимальное\_число>

cmp cx,1 ; Сравнение (CX) с единицей

je Home ; Переход по метке, если сканирование чисел

; последовательности завершено

jmp Scan ; Переход по метке Scan

; ------

Home:

mov sp,bp ; -\

pop bp ; -/ Эпилог процедуры

ret 6 ; Возврат в программу с очисткой стека

Min ENDP ; Завершение процедуры Min

; ---------------------------------------------

include bibl-prc.inc

END Start

**Задачи для самостоятельного решения**

1. Найдите сумму и разность между максимумом и минимумом трёх чисел.
2. Найдите произведение и сумму последовательности 20 натураль­ных чисел.
3. Напишите программу, которая определяет, является ли число простым или нет.
4. Напишите программу, которая определяет, является ли число со­вершенным или нет.
5. Найдите на отрезке [2,K] сумму всех чисел, которые меньше N.
6. На отрезке [2,K] найдите разность между суммами элементов, стоящих на чётных и нечётных местах.
7. Для введённого текста вставьте пробелы после каждого символа.
8. Во введённом тексте подсчитайте количество вхождений заглавных букв.
9. Во введённом тексте подсчитайте, сколько раз встречается вто­рой элемент.
10. Используя программу ввода и вывода числа, напишите программу для ввода произвольной последовательности целых чисел и для её вывода.
11. Выведите в обратном порядке введённую последовательность це­лых чисел.
12. Среди последовательности целых чисел, введённых с клавиату­ры, найдите максимальное число, а также разность между ним и произ­вольным, наперёд заданным числом
13. В последовательности целых чисел подсчитайте количество чи­сел, которые меньше наперёд заданного числа.
14. Для последовательности целых чисел найдите разность между минимальным и максимальным числом.
15. Среди введённых чисел подсчитайте количество простых чисел.
16. Подсчитайте количество совершенных чисел в последовательнос­ти целых чисел.
17. Вставьте в "середину" введённой последовательности символ "\*".
18. Введите две последовательности целых чисел. Выведите третью, которая состоит из чётных элементов обоих последовательностей.
19. Найдите произведение максимумов из двух введённых последова­тельностей целых чисел.
20. Напишите программу, реализующую простейший калькулятор:

'+' - сложение чисел;

'-' - вычитание чисел;

'\*' - умножение чисел;

'/' - деление чисел.

1. Напишите программу, которая позволяет вычислить значение тригонометрической функции угла от 0 до 360 градусов, в зависимости от кода выбора пользователя:

'S' - вычисление значения синуса угла;

'C' - вычисление значения косинуса угла;

'T' - вычисление значения тангенса угла;

'c' - вычисление значения котангенса угла.

Вычисление значений тригонометрических функций оформите в виде отдельных процедур, используя процедуру по вычислению значения синуса угла (см. Пример в разделе "Демонстрационные примеры").

# Лабораторная работа №8. Стек.

Вопросы, необходимые рассмотреть при подготовке к лабораторной работе:

1. Особенности работы со стеком;
2. Алгоритм работы команд записи и чтения.

**Демонстрационные примеры**

 Пример1. Демонстрация использования операторов PUSH и POP в TURBO Debugger

IDEAL

MODEL small

STACK 256

DATASEG

exCode DB 0

CODESEG

Start:

mov ax, @data ; Установка в DS адреса

mov ds, ax ; сегмента данных

;-------------

push ax ; Сохранить ax и bx

push bx ; в стеке

;-------------

mov ax, -1 ; Установить

mov bx, -2 ; тестовые значения

mov cx, 0

mov dx, 0

;-------------

push ax ; Протолкнуть ax в стек

push bx ; Протолкнуть bx в стек

pop cx ; Вытолкнуть cx из стека

pop dx ; Вытолкнуть dx из стека

;-------------

pop bx ; Восстановить из стека сох-

pop ax ; ранённые значения ax, bx

Exit:

mov ah, 04Ch ; Функция DOS: выход

mov al, [exCode] ; Возврат кода ошибки

int 21h ; Останов программы

END Start ; Конец программы

Выполните ассемблирование и компоновку программы с такими опциями:

tasm /zi <имя>.asm

tlink /v<имя>.obj

td<имя>.exe

Осуществите следующие действия:

1. Загрузите программу в окно CPU.

2. Начните пошаговый запуск программы с помощью клавиши {F8}.

Наблюдайте за значением стека в правом нижнем углу - в окне с адресами, первый из которых регистр SS. Вершина стека, расположенная внизу окна, отмечается стрелкой.

3. Загрузив в регистры ax, bx, cx и dx тестовые значения, наблюдайте за содержанием стека и значением sp до и после проталкивания.

В конце программы значения регистров bx и ax, которые хранились там с начала выполнения программы, восстановятся.

Пример2**.** Напишем программу, которая выводит последовательность любых символов (кроме точки) с точкой в конце и проверяет, правильно ли этот текст сбалансирован по круглым и квадратным скобкам. В качестве ответа выдаём букву "Д", если текст сбалансирован, или "Н" - нет.

Для проверки соответствия открывающей и закрывающей скобок поступим так: при появлении закрывающей скобки, заменим её на соответствующую ей открывающую скобку и именно эту скобку будем сравнивать со скобкой, считанной из стека.

%TITLE "Проверка текста"

IDEAL

MODEL small

STACK 256

DATASEG

exCode DB 0

CODESEG

Start:

mov ax, @data ; Установка в DS адреса

mov ds, ax ; сегмента данных

Begin:

mov dl,'Д' ; Предварительный ответ

push dx ; Сохранить его в стеке

mov si,sp ; Запомнить исходное значение SP

vvod:

mov ah,1 ;

int 21h ; Ввод очередного символа

cmp al,13 ; Пока не нажата клавиша {ENTER}

je ENDTXT ;

cmp al,'(' ; Проверка на открывающую скобку

je OPEN

cmp al,'['

jne NOOPEN

OPEN:

push ax ; Запомнить открывающую скобку в стеке

jmp vvod ;

NOOPEN:

mov ah,'(' ; Проверка на закрывающую скобку

cmp al,')' ; (в AH парная ей открывающая скобка)

je CLOSE

mov ah,'['

cmp al,']'

jne vvod ; Не скобка, перейти к следующему символу

CLOSE:

cmp sp,si ; Проверка для закрывающей скобки:

je NO ; стек пуст -> NO

pop cx ; Считать из стека открывающую скобку

cmp cl,ah ;сравнить её с содержимым регистра AH

je vvod ; Есть парность, перейти к следующему

jmp NO ; символу, если нет - NO

ENDTXT:

cmp sp,si ; Проверка на "пусто" в конце текста

je otvet

NO:

mov sp,si ; Очистка стека

mov dl,'Н' ; Ответ "нет"

push dx ; Запомнить его в стеке

otvet:

mov ah,2 ; Функция DOS - печать символа

mov dl,13 ; переход

int 21h ; на

mov dl,10 ; новую строку

int 21h ; для печати ответа

pop dx

int 21h ; Печать ответа

Exit:

mov ah, 04Ch ; Функция DOS: выход

mov al, [exCode] ; Возврат кода ошибки

int 21h ; Останов программы

END Start ; Конец программы

Пример 3. Демонстрация организации изменения порядка следования символов в заданной строке на обратный, используя стек

; ------------------------------------------------------

include bibl-mcr.inc

MASM ; Режим

MODEL small ; Модель памяти

.STACK 256 ; Резервирование 256 байтов под сегмент стека

; ---------------------------------------------------------

.DATA

Str1 DB "А роза упала на лапу Азора",'$'

; ---------------------------------------------------------

Size\_str=($-Str1)-1 ; вычисления длины строки

; ---------------------------------------------------------

.CODE ; Сегмент кода

Start:

mov ax,@data ; (AX)=<Адрес\_сегмента\_данных>

mov ds,ax ; (DS)=(AX)

; --------

Print Str1 ; Печать строки Str1

call NewLine ; Перевод курсора на другую строку

; --------------------------------------------------

; Помещение строки Str1 в стек

; -----------------------------------------------------

mov cx, Size\_str; CX)=<Количество\_символов\_строки\_Str1>

mov si,0 ; SI - регистр для передвижения по строке

InStk:

xor ax,ax

mov al,[Str1+si]

push ax

inc si

loop InStk

; -------------------------------

; Извлечение строки Str1 из стека

; -----------------------------------------------------

mov cx,Size\_str; (CX)=<Количество\_символов\_строки\_Str1>

mov si,0 ; SI - регистр для передвижения по строке

OutStk:

pop ax

mov [Str1+si],al

inc si

loop OutStk

; ---------

Print Str1 ; Печать "перевёрнутой" строки Str1

call Stop ; "Задержка" экрана

mov ax,4C00h

int 21h

include bibl-prc.inc

END Start

Пример 4. Демонстрация программы, реализующей обмен значений крайних элементов стека

include bibl-mcr.inc

MASM ; Режим

MODEL small ; Модель памяти

.STACK 256 ; Резервирование 256 байтов под сегмент стека

; -------------------------------------------------------

.DATA

Stk1 DB "Дно стека: "

DB "------> Вершина стека",'$'

Prig1 DB 0Ah,0Dh,"Введите через пробел 5 чисел в стек: ",'$'

Prig2 DB 0Ah,0Dh,"Произведён обмен первого и последнего "

DB "элементов стека!",0Ah,0Dh,'$'

Prig3 DB "Содержимое стека : ",'$'

; -----------------------------------------------------

.CODE ; Сегмент кода

Start:

mov ax,@data ; (AX)=<Адрес\_сегмента\_данных>

mov ds,ax ; (DS)=(AX)

; -----------------------------------

; Организация ввода пяти чисел в стек

; --------------------------------------

mov si,sp ; (SI)=<Адрес\_дна\_стека>

sub si,2 ; (SI)=<Адрес\_первого\_элемента\_стека>

; --------

Print Stk1 ; Печать строки Stk1

Print Prig1 ; Печать строки Prig1

mov cx,5 ; Количество чисел для ввода в стек

InStk:

call Vvod\_AX ; Ввод целого числа в регистр AX

push ax ; Помещение числа из регистра AX в стек

loop InStk ; Переход на метку InStk, пока (CX)<>0

; ----------------------------------------------------

; Обмен значений крайних элементов стека

; -------------------------------------------------

mov bp,sp ; (BP) - указатель на вершину стека

mov ax,[bp] ; (AX)=<Последний\_элемент\_стека>

xchg [bp+8],ax ; Обмен крайних элементов стека

mov [bp],ax ; (AX)=<Первый\_элемент\_стека>

Print Prig2 ; Печать строки Prig2

; -----------------------------------------------------

; Организация вывода5 чисел из стека, начиная с его дна

; -----------------------------------------------------

Print Prig3 ; Печать строки Prig3

mov cx,5 ; Количество чисел для вывода из стека

mov bp,si ; (BP)=(SI) - адрес первого элемента стека

OutStk:

mov ax,[bp] ; (AX)=<Число\_из\_стека>

call Vivod\_AX ;Вывод целого числа из AX на экран

mov ah,02h ; Номер функции вывода символа на экран

mov dl,' ' ; (DL)=<Код\_выводимого\_символа>

int 21h ; Вызов функции DOS с номером 02h

sub bp,2 ; (BP)=(BP)-2

loop OutStk ; Переход на метку OutStk, пока (CX)<>0

; ---------

call Stop ; "Задержка" экрана

mov ax,4C00h

int 21h

include bibl-prc.inc

END Start

**Задачи для самостоятельного решения**

1. Поменять местами пятый и седьмой элементы стека.
2. Вывести на экран номер первого встретившегося элемента стека, значение которого меньше некоторого числа N.
3. Заполнить 20 элементов стека нулями.
4. Написать программу очистки стека.
5. В стеке, содержащем 20 элементов, найти наибольший.
6. В стеке, содержащем 30 элементов, найти минимальный.
7. Поменять значение регистров ax, bx, cx, dx, es с 1, 2, 3, 4 и 5 элементами стека.
8. Поменять значение регистров al, bl, cl, dl, el с 1, 2, 3, 4 и 5 элементами стека.
9. Напечатать первые 10 элементов стека.
10. В стеке из N элементов подсчитать количество 0.
11. Подсчитать сумму чётных элементов стека.
12. Найти произведение нечётных элементов стека.
13. Написать программу, которая проверяет для введённого текста: для каждой ли строчной буквы найдётся прописная. Причём заранее известно, что текст состоит из букв и строчная и прописная буквы могут быть разными.
14. Подсчитать для введённого текста количество нулей и единиц.
15. В стеке из 40 элементов найти произведение второго и максимального элементов.
16. В стеке из N элементов найти разность между максимальным и минимальным элементами.
17. В стеке из N элементов найти произведение минимального и максимального элементов.
18. В стеке из N элементов найти сумму всех элементов больших K.
19. В введённом тексте подсчитать количество вхождений заглавных букв и соответствующих им строчных.
20. Вычислить сумму элементов стека между минимальным и максимальным элементами стека.

# Лабораторная работа №9. Цепочки

Вопросы, необходимые рассмотреть при подготовке к лабораторной работе:

1. Понятие "цепочечная команда"
2. Синтаксис цепочечных команд
3. Пересылка цепочки
4. Сравнение цепочек
5. Сканирование цепочки
6. Загрузка элемента из цепочки
7. Сохранение элемента в цепочке
8. Префиксы повторения
9. Цепочечные индексные регистры
10. Направление обработки цепочки

**Демонстрационные примеры.**

 Пример1. Демонстрация перемещения цепочки-источника в цепочку-приёмник командой MOVS

MASM

MODEL small

.STACK 256

.DATA ; Сегмент данных

source DB "Тестируемая строка",'$' ; Строка-источник

dest DB 19 DUP (" ") ; Строка-приёмник

.CODE

ASSUME ds:@data, es:@data

Main: ; Точка входа в программу

mov ax,@data ; Загрузка сегментных регистров

mov ds,ax ; Настройка регистров ds и es

mov es,ax ; на адрес сегмента данных

cld ; Сброс флага DF - обработка

; строки от начала к концу

; строки-приёмника

lea si,source ; Загрузка в si смещения

; строки-источника

lea di,dest ; Загрузка в DS смещения

mov cx,20 ; Счётчик повторений (длина

; строки) для префикса rep

rep movs dest,source ; Пересылка строки

lea dx,dest

mov ah,09h ; Вывод на экран

int 21h ; строки-приёмника

mov ah,08h ; Функция ожидания нажатия любой клавиши

int 21h ; Вызов DOS: код нажатой клавиши - в AL

Exit:

mov ax,4C00h

int 21h

END Main

Пример 2. Демонстрация использования команды CMPS для сравнения двух строк, заданных в программе

IDEAL

MODEL small

STACK 256

DATASEG

match DB 'Строки совпадают','$'

failed DB 'Строки не совпадают, так как: ','$'

str1 DB "0123456789",0ah,0dh,'$' ; Исследуемые строки

str2 DB "0143406779",0ah,0dh,'$' ;

; --------------------------------------------------------

; После просмотра результата программы при исходных данных

; закомментируйте предыдущую строку и уберите знак

; комментария со следующей строки

; --------------------------------

; str2 DB "0123456789",0ah,0dh,'$'

CODESEG

ASSUME ds:@data, es:@data; Привязка DS и ES к сегменту данных

Main:

mov ax,@data ; Загрузка сегментных регистров

mov ds,ax

mov es,ax ; Настройка ES на DS

; --------

mov ah,09h ; Функция для вывода строки

lea dx,[str1] ; Загрузка адреса строки в регистр DX

int 21h ; Вызов функции для вывода строки

lea dx,[str2] ; Загрузка адреса строки в регистр DX

int 21h ; Вызов функции для вывода строки

cld ; Сброс флага DF для осуществления сравнения

; строк в направлении возрастания адресов

lea si,[str1] ; Загрузка в SI смещения строки str1

lea di,[str2] ; Загрузка в DI смещения строки str2

mov cx,10 ; Длина исследуемых строк

; -------------------------------------------------------

; Сравнение строк до тех пор, пока сравниваемые элементы

; строк равны. Выход при обнаружении несовпавшего элемента

; -------------------------------------------------------

mov ax,0 ; Для указания того факта, что строки равны

Cycl:

repe cmps [str1],[str2]

jcxz Equal ; Переход, если закончен просмотр строк

jne Not\_match ; Переход, если строки не совпадают

Equal:

cmp ax,1

je Exit ; Переход, если было несовпадение в строках

mov ah,09h

lea dx,[match]

int 21h ; Вывод сообщения о том, что строки равны

jmp Exit

Not\_match:

cmp ax,1 ; Если обнаружено больше одного несовпадения,

je See ; то не печатать повтор сообщения об этом

mov ah,09h

lea dx,[failed]

int 21h ; Вывод сообщения о том, что строки не равны

See:

dec si ; Возврат к несовпавшему элементу в строке

dec di ;

; --------

mov ah,02h ; Функция вывода символа на экран

mov dl,13 ; ASCII-символ возврата каретки

int 21h ; Вызов функции возврата каретки

mov dl,10 ; Код символа - переход на новую строку

int 21h ; Вызов функции

; ----------------------------------

; Вывод несовпадающих символов строк

; ----------------------------------

mov dl,[byte ptr si]

int 21h

mov dl,'<'

int 21h

mov dl,'>'

int 21h

mov dl,[byte ptr di]

int 21h

inc si ; Продолжение поиска в строке

inc di ; после обнаружения несовпадения

mov ax,1 ; Фиксирование факта о несовпадении символов

jmp Cycl

mov ah,08h ; Функция ожидания нажатия любой клавиши

int 21h ; Вызов DOS: код нажатой клавиши - в AL

Exit:

mov ax,4C00h

int 21h

END Main

Пример 3. Демонстрация установки пароля в программе для продолжения её работы при верном вводе пароля

STK SEGMENT STACK ; Откроем сегмент стека

DB 256 DUP (?) ; Отводим под стек 256 байтов

STK ENDS ; Закроем сегмент стека

DATA SEGMENT

password DB 'data' ; Ожидаемый пароль

leng=$-password

string DB 80 DUP (?) ; Поле для ввода пароля

prompt DB 'Введите пароль: $' ; Запрос

er DB 0Ah,0Dh,'Неверный пароль! $'

ok DB 0Ah,0Dh,'Доступ к программе открыт!',0Ah,0Dh,'$'

prog DB 'Секретный шифр: 1422-07-02 $'

DATA ENDS

CODE SEGMENT

ASSUME CS:code, DS:data

Begin:

mov ax,data

mov ds,ax

; --------

mov ah,09h

mov dx,OFFSET prompt

int 21h

; -----------

; Ввод пароля

; -----------

sub bx,bx ; (BX)=0

; (BX)=<количество\_введенных\_символов>

Pass:

mov ah,08h ; Функция ввода символа без отображения

int 21h ; на экране дисплея

cmp al,13 ; Проверка на конец ввода строки

je Compare ; Ввод строки закончен

mov string[bx],al ; Запись введённого символа в память

inc bx ; (BX)=(BX)+1

jmp Pass ; Продолжение ввода пароля

; ------------------------------------------------

; Сравнение строк для проверки правильности пароля

; ------------------------------------------------

Compare:

push ds ; Помещение в стек адреса DS

pop es ; Отображение данных в ES

lea si,string ; (SI)=<Смещение\_строки\_string>

lea di,password ; (DI)=<Смещение\_строки\_password>

cld ; Сброс флага направления

cmp bx,leng

jne Error

mov cx,bx ;(CX)=<Количество\_введённых\_символов>

repe cmpsb ; Посимвольное сравнение

jne Error ; Несовпадение строк

; -------------------------------------------------------

; Вывод сообщения ok, подтверждающего правильность пароля

; -------------------------------------------------------

mov ah,09h

mov dx,OFFSET ok

int 21h

; ------------------------------------

; Программа, к которой появился доступ

; ------------------------------------

mov ah,09h

mov dx,OFFSET prog

int 21h

Exit:

mov ah,08h ; Функция ожидания нажатия любой клавиши

int 21h ; Вызов DOS: код нажатой клавиши - в AL

mov ax,4C00h

int 21h

Error: ; Введён неправильный пароль

mov ah,09h

mov dx,OFFSET er

int 21h

jmp Exit

CODE ENDS

END Begin

Пример 4. Демонстрация использования команды LODS для определения факта, совпадают ли первый и предпоследний символы данной строки

MASM

MODEL small

.STACK 256

.DATA

s1 DB'Первыйипредпоследний символы строки совпадают!','$'

s2 DB 'Первый и предпоследний символы строки ','$'

s3 DB 'не совпадают!','$'

string DB "НЕ ДО ЛОГИКИ - ГОЛОДЕН!",0ah,0dh,'$'

; --------------------------------------------------------

; После просмотра результата программы при исходных данных

; закомментируйте предыдущую строку и уберите знак

; комментария со следующей строки

; ----------------------------------------------

; string DB "Не до логики - голоден!",0ah,0dh,'$'

N=$-string ; Длина строки string

.CODE

ASSUME ds:@data, es:@data;Привязка DS и ES к сегменту данных

Main:

mov ax,@data

mov ds,ax ; Настройка регистров ds и es

mov es,ax ; на адрес сегмента данных

; --------

mov ah,09h

lea dx,string

int 21h ; Вывод строки string

; ------------------

mov si,OFFSET string ; Присвоить si адрес цепочки

lods string ; Получить первый байт цепочки

dec si ; (SI)=<Адрес\_первого\_байта\_строки>

add si,N-5 ; (SI)=<Адрес\_предпоследнего\_байта\_строки>

cmp [si],al

je Yes

mov ah,09h

lea dx,s2

int 21h ; Вывод сообщения s2

lea dx,s3

int 21h ; Вывод сообщения s3

jmp Exit

Yes:

mov ah,09h

lea dx,s1

int 21h ; Вывод сообщения s1

Exit:

mov ah,08h ; Функция ожидания нажатия любой клавиши

int 21h ; Вызов DOS: код нажатой клавиши - в AL

mov ax,4C00h

int 21h

END Main

Пример 5. Программа осуществляет ввод строки с клавиатуры, размещение её в памяти по адресу ES:DI и вывод на экран

; Программа разбита на следующие процедуры:

; WrStr - печать Вашей строки

; WrStr1 - печать prigl1

; WrStr2 - печать prigl2

; RdStr - чтение Вашей строки с клавиатуры

; и запись ее по адресу ES:DI

; Space - переход курсора на начало новой строки

; ----------------------------------------------

IDEAL

MODEL small

STACK 256

DATASEG

exCode DB 0

prigl1 DB "Введите строку из 10 символов.","$"

prigl2 DB "Вот ваша строка: ","$"

UDATASEG

string DB 11 dup (?) ; Резервирование пространства под

; Вашу строку

CODESEG

Start:

mov ax,@data ; Инициализация сегментных регистров

mov ds,ax ; DS

mov es,ax ; ES

call WrStr1 ; Печать приглашения1

call Space

call RdStr ; Размещение вашей строки в памяти

call Space

call WrStr2 ; Печать приглашения2

call Space

call WrStr ; Печать Вашей строки

mov ah,8 ; Функция ожидания нажатия любой клавиши

int 21h ; Вызов DOS: код нажатой клавиши - в AL

Exit:

mov ah,04Ch ; Функция DOS: выход из программы

mov al,[exCode] ; Возврат значения кода ошибки

int 21h ; Вызов DOS. Останов программы

; \*\*\*\*\*\*\*\*\*

PROC WrStr2

mov dx,OFFSET prigl2 ; prigl2 по адресу DS:DX

mov ah,09 ; Функция DOS: печать ASCII-строки

int 21h

ret

ENDP WrStr2

; \*\*\*\*\*\*\*\*\*

PROC WrStr

mov dx,OFFSET string ; string по адресу DS:DX

mov ah,9 ; Функция DOS: печать ASCII-строки

int 21h

ret

ENDP WrStr

; \*\*\*\*\*\*\*\*\*

PROC WrStr1

mov dx,OFFSET prigl1 ; Инициализация смещения prigl1

mov ah,09 ; Функция вывода prigl1

int 21h

ret

ENDP WrStr1

; \*\*\*\*\*\*\*\*\*

PROC Space

mov ah,2 ; Функция DOS: печать символа из DL

mov dl,13 ; ASCII-символ возврата каретки

int 21h

mov dl,10 ; ASCII-символ перехода на новую строку

int 21h

ret ; Возврат в основную программу

ENDP Space

; \*\*\*\*\*\*\*\*

PROC RdStr

mov di,OFFSET string ; Адрес строки ES:DI

mov cx,10 ; Длина строки

mov ah,1 ; Функция DOS: чтение с клавиатуры символа

cmp cx,0

je Zero

cld ; Сброс флага направления

Repeat:

int 21h ; Запись введенного значения в регистр AL

stosb ; Запись символа из AL в ES:DI

loop Repeat ; DI<-DI+1

Zero:

mov al,24h ; Запись последнего символа "$"

stosb ; DI<-DI+1

ret

ENDP RdStr

END Start ; Конец программы / точка входа

**Задачи для самостоятельного решения**

 Указание. Во всех программах используйте процедуры.

1. Напишите программу, которая первый и последний символ меняла бы местами.
2. Реализуйте алгоритм перевода строчных букв в прописные, если заведомо известно, что строка состоит только из букв латинского алфавита.
3. Подсчитайте в строке количество пробелов.
4. Подсчитайте количество вхождений в строку ее первого символа.
5. Замените в строке все пробелы на "?".
6. С помощью оператора цикла выведите на экран цифры от 0 до 9.
7. Напишите программу, которая печатала бы Ваше имя, фамилию и знак зодиака, которые Вы введёте с клавиатуры.
8. Сравните три строки и выдайте на экран результат сравнения. Строки должны вводиться с клавиатуры.
9. Присоедините к одной строке другую, введённую с клавиатуры.
10. Напечатайте введённую строку в обратном порядке.
11. Переставьте в строке все заглавные буквы в начало строки, затем строчные, затем цифры, затем остальные символы.
12. Удалите в строке все пробелы.
13. В введённой строке вставьте пробелы после каждой цифры.
14. Для введённой строки выясните, является ли она предложением.
15. Подсчитайте, сколько в введённой строке слов.
16. Выясните, образует ли введённая строка палиндром.
17. Вставьте в "середину" строки символ "-".
18. Если две введённых строки не равны, то выведите тот символ, начиная с которого начинается различие.
19. Объедините три строки в том случае, если они не равны, если равны две из них, то объедините неравные строки. Если они все равны между собой, то выведите строку, состоящую из чётных элементов.
20. Выведите на экран 6 раз строчку 'И вдруг она поняла, отчетливо и горько, что 60 лет была счастлива с совершенно чужим человеком', каждый раз уменьшая количество лет на 10. (Указание: Заменяйте только старшую цифру числа.)
21. Напишите программу, которая выводит на экран сначала только второе слово из строки, состоящей более, чем из трёх слов, а потом и всю строку.

# Лабораторная работа №10. Структуры

Вопросы, необходимые рассмотреть при подготовке к лабораторной работе:

1. Структура. Шаблон структуры.
2. Описание шаблона структуры.
3. Определение данных типа «структура».
4. Основные положения при определении структур.
5. Ссылки на поля структур.

**Демонстрационные примеры.**

Пример 1**.** В переменной типа Date хранится дата марта месяца 2008 года. Демонстрация записи даты следующего дня года в эту переменную

IDEAL

MODEL small

STACK 256

; --------

STRUC Date

day DB ? ; Поле "День"

month DB 3 ; Поле "Месяц"

year DW 2008 ; Поле "Год"

ENDS Date

; -------

DATASEG

Dt1 Date <> ; Массив, содержащий одну структуру типа Date

prigl DB "Введите число месяца: ",'$'

er DB "Количество дней в марте 2008 года - 31.",'$'

CODESEG

EXTRN Vvod\_AL:proc, Vivod\_AX:proc, Kursor:proc, Stop:proc

Start:

mov ax,@data

mov ds,ax

; -------

mov ah,09

lea dx,[prigl]

int 21h

; ----------

call Vvod\_AL ; Вызов процедуры ввода числа месяца

cmp al,0

jle Exit

cmp al,31

jg Exit

mov [Dt1.day],al ; (Dt1.day)=(AL)

; --------------

call Vivod\_Data ; Вызов процедуры печати даты

; -------------

call Kursor ; Вызов процедуры перевода курсора

call MarthDate ; Вызов процедуры увеличивающую число

; -------------

call Vivod\_Data ; Вызов процедуры печати новой даты

jmp V

Exit:

mov ah,09

lea dx,[er]

int 21h ; Вывод сообщения об ошибке

V: call Stop ; Вызов процедуры задержки экрана

mov ax,4C00h

int 21h

; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Процедура, которая "переворачивает листок календаря

; на следующий день"

; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

PROC MarthDate

cmp [Dt1.day],31 ; Если 31 марта, то переход

je Apr1 ; на метку Apr1

inc [Dt1.day] ; Следующий день в марте

jmp Fin

Apr1:

mov [Dt1.month],4 ; Замена месяца март на апрель

mov [Dt1.day],1 ; Замена числа 31 на 1

Fin:

ret

ENDP MarthDate

; \*\*\*\*\*\*\*\*\*\*\*\*\*

PROC Vivod\_Data ; Процедура печати даты календаря

mov ah,0

mov al,[Dt1.day]

call Vivod\_AX ; Вывод на экран числа месяца

mov ah,2

mov dl,'.'

int 21h ; Печать знака ".", как разделителя чисел

mov ah,0

mov al,[Dt1.month]

call Vivod\_AX ; Вывод на экран порядкового номера месяца

mov ah,2

mov dl,'.'

int 21h ; Печать знака ".", как разделителя чисел

mov ax,[Dt1.year]

call Vivod\_AX ; Вывод на экран года

ret

ENDP Vivod\_Data

END Start

Пример 2. Демонстрация описания структуры Stud, содержащей фамилию, номер группы и отметки. Определение количества студентов- отличников из семи представленных кандидатур

MASM

MODEL small

.STACK 256

; --------

Stud STRUC

fam DB 20 DUP(?) ; Фамилия

gr DW ? ; Номер группы

otm DB 5 DUP(?) ; Отметки

fin DB 0Ah,0Dh

Stud ENDS

; -------

.DATA

prigl DB "Фамилия Группа Отметки",0Ah,0Dh,'$'

St1 Stud <"Фамилия\_1",'1','55555'>

St2 Stud <"Фамилия\_2",'1','55455'>

St3 Stud <"Фамилия\_3",'2','45534'>

St4 Stud <"Фамилия\_4",'1','55555'>

St5 Stud <"Фамилия\_5",'2','33434'>

St6 Stud <"Фамилия\_6",'1','55555'>

St7 Stud <"Фамилия\_7",'2','54545'>

otl DB "Количество студентов-отличников: ",'$'

.CODE

EXTRN Vivod\_AX:proc

Start:

mov ax,@data

mov ds,ax

; -------

mov ah,09 ; Функция вывода строки

lea dx,prigl

int 21h ; Вызов функции

; -----------

call OtlStuds ; Вызов процедуры

; -----------

push dx ; (DX)=<Число\_студентов\_отличников>

; -----------------------------------------------------

; Вывод на экран содержимого структуры Stud

; -------------------------------------------------

lea dx,St1 ; (DX)=<Адрес\_начала\_структуры>

mov ah,09 ; Функция вывода строк

int 21h ; Вызов функции вывода строки

; -----

pop ax ; (AX)=<Число\_студентов\_отличников>

call Vivod\_AX

Exit:

mov ah,08h

int 21h ; Задержка экрана

mov ax,4C00h

int 21h ; Выход из программы

; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Процедура для подсчёта студентов-отличников

; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

OtlStuds PROC

mov di,type Stud

lea bx,St1

mov dx,0 ; Счётчик студентов-отличников

; Внешний цикл (по студентам)

mov cx,7 ; Количество человек

Repeat1:

push cx

; Внутренний цикл (по оценкам)

mov cx,5 ; Количество отметок

Repeat2:

cmp [bx+si].otm,'5'

jne Next

inc si ; (SI)=<Число\_просмотренных\_отметок>

loop Repeat2

inc dx ; (DX)=<Количество\_студентов\_отличников>

; К следующему студенту

Next:

pop cx ; (CX)=<Количество\_человек>

xor si,si ; (SI)=0

add bx,di ; (DI)=<Адрес\_следующей\_структуры\_типа\_Stud>

loop Repeat1 ; Переход к следующей кандидатуре

ret

OtlStuds ENDP

END Start

Пример 3. Демонстрация ввода данных о двух студентах (Фамилия И.О., номер группы) в память компьютера и вывод информации о переводе второго по списку студента в группу первого студента при условии, что изначально студенты учились в разных группах

MASM

MODEL small

.STACK 256

; --------

Stud STRUC

fam DB 35 DUP(' '),'$' ; Фамилия И.О.

; ----------------------------------------

; Альтернатива записи предыдущей строки

; fam DB 32,34 DUP(' '),'$'

; ----------------------------------------

gr DW ?,'$' ; Номер группы

Stud ENDS

; -------

.DATA

prigl1 DB "Фамилия Номер группы",0Ah,0Dh,'$'

prigl2 DB "Второй студент переведён в группу первого ",'$'

prigl3 DB "студента:",0Ah,0Dh,'$'

Date Stud 2 DUP (<>)

.CODE

EXTRN Vvod\_AX:proc, Vivod\_AX:proc

Start:

mov ax,@data

mov ds,ax

; -------

mov ah,09

lea dx,prigl1 ; Информация о вводе

int 21h

; ------

mov cx,2 ; Количество человек, о которых собирается

xor bx,bx ; информация

Rpt:

push cx ; Сохранение содержимого регистра CX

xor cx,cx ; (CX)=0

; --------

mov ah,0Ah ; -\

mov dx,OFFSET Date.fam ; \

add dx,bx ; / Ввод фамилии

int 21h ; -/

; -------------------

mov cl,Date.fam[1]+bx ; (CL)=<Число\_введённых\_символов>

mov si,cx ; (SI)=(CX)

mov Date.fam[bx+si+2],' ' ; Последний символ введённой

mov dl,35 ; строки "пустой" либо "пробел"

call Long\_Probel

; --------------

call Vvod\_AX

mov Date.gr+bx,ax ; Ввод номера группы

; ---------------

pop cx ; (DI)=<Число\_байтов,\_отведённых\_

mov di,type Date ; \_под\_структуру>

add bx,di ; (BX)=<Адрес\_следующей\_структуры\_

loop Rpt ; \_из\_массива\_структур>

; -------

mov ah,09

lea dx,prigl2

int 21h

lea dx,prigl3

int 21h

; -------

mov ah,09 ; Функция вывода строк

lea dx,Date.fam+di+2 ; (DX)=<Адрес\_начала\_структуры>

int 21h ; Вызов функции вывода строки

mov dl,35 ; (DL)=<Номер\_столбца\_на\_экране>

call Long\_Probel ; Вызов процедуры по установке

; -------------- ; курсора на нужную позицию

mov ax,Date.gr ; (AX)=<Значение\_поля\_gr\_первого\_

call Vivod\_AX ; \_элемента\_массива\_структур>

; -----------

mov ah,08h

int 21h ; Задержка экрана

mov ax,4C00h

int 21h ; Выход из программы

; \*\*\*\*\*\*\*\*\*\*\*\*\*\*

Long\_Probel PROC ; Процедура установки курсора

push ax

push bx

push dx

mov ah,2 ; Запрос на установку курсора

mov bh,0 ; Номер экрана (или страницы) 0

mov dh,24 ; Строка (нижний левый угол)

int 10h ; Передача управления в BIOS

pop dx

pop bx

pop ax

ret

Long\_Probel ENDP

END Start

**Задачи для самостоятельного решения**

1. Опишите, используя структуру, телефонную книгу. Составьте программу, выдающую список абонентов, имеющих телефонный номер, начинающийся на 331.
2. Опишите, используя структуру, каталог книг в библиотеке. Составьте программу, выдающую список книг В.Пикуля, хранящихся в библиотеке.
3. Опишите, используя структуру, таблицу дат и событий русской истории. Составьте программу, выдающую список событий 19 века.
4. Опишите, используя структуру, таблицу дат и событий русской истории. Составьте программу, выдающую список событий 18 века.
5. Опишите, используя структуру, школьный класс (фамилия и инициалы, дата рождения, месяц рождения, год рождения). Составьте программу, выдающую список учеников, рождённых в мае месяце.
6. Опишите, используя структуру, записную книжку. Составьте программу, выдающую список друзей, кому в этом году исполняется 20 лет (фамилия и инициалы, год рождения, дата рождения, месяц рождения).
7. Опишите, используя структуру, школьный класс (фамилия и инициалы, дата рождения, месяц рождения, год рождения). Составьте программу, выдающую день рождения класса (среднее арифметическое дат и месяцев).
8. Опишите, используя структуру, выборы (фамилия кандидата и количество набранных голосов). Всего избирателей 2000. Составьте программу, определяющую кто из делегатов прошёл или необходимо проводить повторные выборы (должно быть набрано 1/3 голосов от общего количества).
9. Опишите, используя структуру, школьную нагрузку (фамилия преподавателя, класс, часы). Составьте программу, определяющую нагрузку каждого преподавателя. Определить, у какого преподавателя самая большая нагрузка и у кого самая низкая.
10. После поступления в ВУЗ о студентах собрана информация: фамилия, нуждается ли в общежитии, стаж, работал ли учителем, что окончил, какой язык изучал. Составьте программу, определяющую: (1) сколько человек нуждается в общежитии; (2) списки студентов, проработавших 2 и более лет учителем; (3) списки окончивших педучилище; 4) списки языковых групп.
11. Опишите, используя структуру, таблицу соревнований (название команды, количество набранных очков). Выберите команду, занявшую первое место. Упорядочите список команд в зависимости от занятого места.
12. При сдаче норм ГТО, были получены результаты забега на 100 метров и прыжков в длину. Задайте нормы ГТО по этим видам, определите списки учеников, не выполнивших нормативы, количество учеников сдавших нормативы, а также списки лучших.
13. Опишите, используя структуру, вступительные экзамены. Абитуриенты сдавали 3 экзамена, для поступления необходимо набрать 12 баллов. Определите списки абитуриентов: зачисленных в институт; количество несдавших экзамены; списки абитуриентов сдавших экзамены на "5".
14. Опишите, используя структуру, оценки за год. Посчитайте процент и качество успеваемости в классе за год, составьте списки неуспевающих и отличников.
15. Опишите, используя структуру записи, данные на учеников (фамилия, улица, дом, квартира). Составьте программу, определяющую количество учеников живущих на улице Свердлова, списки учеников, проживающих в доме номер 45.
16. В библиотеке для каждого заказывающего книгу читателя заполняется карточка: фамилия, дата заказа, дата выдачи книги. Определите: (1) самый маленький срок, за который нашли книгу; (2) сколько заказов было не удовлетворено; (3) кто чаще всего берёт книги; (4)­кому выдали книги 15.09.90; (5) сколько человек заказывали книги 25.04.90.
17. Опишите, используя структуру, почтовую сортировку (город, улица, дом, квартира, кому, ценность). Составьте программу, определяющую: (1) сколько посылок отправлено в г.Самару; (2) сколько и куда (список городов) отправлено посылок ценностью выше 10 рублей; (3) есть ли адреса, куда отправлено более одной посылки, если есть, то сколько и кому.
18. Во время сессии несколько студентов не сдали экзамен (фамилия, предмет, группа, дата). Определите список пересдавших, сколько не сдали, кто пересдал геометрию 18.01.90.
19. В читательском билете библиотеки есть данные о человеке (фамилия), записываются данные о книге (автор, название, дата, когда книгу брали читать). Определите, кто брал книгу И.Ефремова "Таис Афинская" 15.05.90, сколько читателей брали книги А.С.Пушкина, количество книг А.Дюма и какие есть в библиотеке, кто и какие книги брал 21.04.90.
20. Опишите, используя структуру, товар (наименование товара, старая цена, новая цена). Составьте программу, определяющую, на какие товары повысятся цены и на сколько процентов.

# Лабораторная работа №11. Файлы

Вопросы, необходимые рассмотреть при подготовке к лабораторной работе:

1. Файл. Понятие о файловой системе
2. Обработка файлов. Дескриптор (файловый номер).
3. Создание файла с усечением существующего до нулевой длины.
4. Запись в файл.
5. Открытие файла для чтения.
6. Чтение данных из файла. Файловый указатель.

**Демонстрационные примеры**

 Пример 1. Программа, демонстрирующая создание файла text.txt в каталоге LANGUAGE

ASSUME DS:data, CS:code

DATA SEGMENT

address DB 'C:\LANGUAGE\text.txt',0 ; Адрес файла

number DW ? ; Файловый номер

DATA ENDS

CODE SEGMENT

Begin:

mov ax,data ; Инициализация регистра DS значением

mov ds,ax ; из переменной data

mov ah,3Ch ; Создание файла text.txt на диске С:\

mov cx,0 ; без атрибутов

lea dx,address ; Помещение в DX адреса создаваемого файла

int 21h

; -----------

mov number,ax ; Сохранение номера файла

; -----------

mov ax,4C00h

int 21h

CODE ENDS

END Begin

 Пример 2. Программа, демонстрирующая создание файла text.txt в каталоге LANGUAGE с возникновением ошибки, сообщение о которой будет выведено на экран

ASSUME CS:code, DS:data

DATA SEGMENT

address DB 'C:\LEX&YACC\text.txt',0 ; Адрес файла

message DB 'Ошибка $' ; Сообщение об ошибке

number DW ? ; Файловый номер

DATA ENDS

CODE SEGMENT

Begin:

mov ax,data ; Инициализация регистра DS значением

mov ds,ax ; из переменной data

mov ah,33 ; Ошибка: функция неизвестна

mov cx,0 ; Атрибут создаваемого файла отсутствует

lea dx,address ; Помещение в DX адреса создаваемого файла

int 21h ; Вызов DOS

; ------

jc Error ; Переход на обработку ошибок, если (CF)=1

; -----------

mov number,ax ; Сохранение файлового номера

Error:

mov ah,09h ; Функция вывода сообщения на экран (09h)

mov dx,offset message

int 21h ; Вызов DOS

;-----------

mov ax,4C00h

int 21h

CODE ENDS

END Begin

 Пример 3. Демонстрация записи в несуществующий файл asm.txt каталога LANGUAGE строки, содержащей фамилию и имя автора языка программирования "ассемблер"

ASSUME CS:code, DS:data

DATA SEGMENT

address DB 'C:\LANGUAGE\asm.txt',0

surn DB 'Грейс М.Хоппер'

number DW ?

DATA ENDS

CODE SEGMENT

Begin:

mov ax,data ; Инициализация регистра DS значением

mov ds,ax ; из переменной data

mov ah,3Ch ; Запрос на создание файла asm.txt

mov cx,0 ; Атрибут файла отсутствует

lea dx,address ; Помещение в DX адреса создаваемого файла

int 21h

; -----------

mov number,ax ; Сохранение номера открытого файла

; -----------

mov ah,40h ; Запрос на запись данных в файл

mov bx,number ; Установка в BX номера открытого файла

mov cx,14 ; Установка в CX число записываемых байт

lea dx,surn ; Помещение в DX адреса области ОП

int 21h ; Вызов DOS

; --------

mov ah,3Eh ; Запрос на закрытие файла asm.txt

mov bx,number ; Установка в BX файлового номера

int 21h ; Вызов DOS

; ----------

mov ax,4C00h

int 21h

CODE ENDS

END Begin

 Пример 4. Демонстрация записи в существующий "ненулевой" файл test.txt, содержимое которого "......... - программа, переводящая текст программы, написанной в мнемонических машинных кодах, в числовые машинные коды.", вместо точек нужное слово

ASSUME CS:code, DS:data

DATA SEGMENT

address DB 'C:\LANGUAGE\test.txt',0 ; Адрес файла

number DW ? ; Дескриптор

wrdlen DW 10 ; Область памяти под слово

wrd DB 'Ассемблер' ; Вставляемое слово

DATA ENDS

CODE SEGMENT

Begin:

mov ax,data ; Инициализация регистра DS значением

mov ds,ax ; из переменной data

mov ah,3Dh ; Запрос на открытие файла test.txt

mov al,2 ; Режим доступа: чтение и запись

lea dx,address ; Помещение в DX адреса создаваемого файла

int 21h ; Вызов DOS

; -----------

mov number,ax ; Сохранение номера открытого файла

; -----------

mov ah,40h ; Запрос на запись данных в файл

mov bx,number ; Установка в BX файлового номера

mov cx,wrdlen ; Установка в CX число записываемых байт

lea dx,wrd ; Помещение в DX адреса области памяти wrd

int 21h ; Вызов DOS

; --------

mov ah,3Eh ; Запрос на закрытие файла test.txt

mov bx,number ; Установка в BX файлового номера

int 21h ; Вызов DOS

; ----------

mov ax,4C00h

int 21h

CODE ENDS

END Begin

 Пример 5. Демонстрация копирование информации, содержащейся в файле ex.txt, в файл copy\_ex.txt. Максимальная область памяти, отведённая для содержимого файла copy\_ex.txt 18 байтов

ASSUME CS:code, DS:data

DATA SEGMENT

addr DB 'ex.txt',0

number DW ?

addr1 DB 'copy\_ex.txt',0

number1 DW ?

memor DB 18 DUP(' ')

memorlen DW 18

DATA ENDS

CODE SEGMENT

Begin:

mov ax,data ; Инициализация регистра DS значением

mov ds,ax ; из переменной data

mov ah,3Ch ; Запрос на созданиe файла copy\_ex.txt

mov cx,0 ; Атрибут файла отсутствует

lea dx,addr1 ; Помещение в DX адреса создаваемого файла

int 21h ; Вызов DOS

; ------------

mov number1,ax ; Сохранение номера открытого файла

; ------------

mov ah,3Dh ; Запрос на открытие файла copy\_ex.txt

mov al,2 ; Режим доступа: чтение и запись

lea dx,addr1 ; Помещение в DX адреса создаваемого файла

int 21h ; Вызов DOS

; ------------

mov number1,ax ; Сохранение номера открытого файла

; ------------

mov ah,3Dh ; Запрос на открытие файла ex.txt

mov al,2 ; Режим доступа: чтение и запись

lea dx,addr ; Помещение в DX адреса создаваемого файла

int 21h ; Вызов DOS

; -----------

mov number,ax ; Сохранение номера открытого файла

; -----------

mov ah,3Fh ; Запрос на чтение данных из файла ex.txt

mov bx,number ; Установка в BX файлового номера

mov cx,memorlen ; Запись в cx число считываемых байт

lea dx,memor ; Помещение в DX адреса области памяти

int 21h

; --------

mov ah,40h ; Запрос на запись данных

; в файл copy\_ex.txt

mov bx,number1 ; Установка в BX номера открытого файла

mov cx,memorlen ; Запись в CX число записываемых байт

lea dx,memor ; Помещение в DX адреса области памяти

int 21h ; Вызов DOS

; --------

mov ah,3Eh ; Запрос на закрытие файла copy\_ex.txt

mov bx,number1 ; Установка в BX файлового номера

int 21h ; Вызов DOS

; --------

mov ah,3Eh ; Запрос на закрытие файла ex.txt

mov bx,number ; Установка в BX файлового номера

int 21h ; Вызов DOS

; ----------

mov ax,4C00h

int 21h

CODE ENDS

END Begin

 Найдите ошибки в следующей программе:

  ; Организуйте ввод с клавиатуры списка фамилий студентов своей группы вида:

; Фамилия Имя

; Фамилия Имя

; Данный список поместить в файл list.txt

; ---------------------------------------

ASSUME cs:code, ds:data

DATA SEGMENT

addr DB 'list.txt',0

f DB 0dh, 0ah,'$'

number DW ?

message DW 25 DUP(' ')

DATA ENDS

CODE SEGMENT

fin PROC ; Процедура вывода конца строки

mov ah,09h

mov dx,offset f

int 21h

ret

fin ENDP

main:

mov ax,data ; Инициализация регистра DS значением

mov ds,ax ; из переменной data

mov ah,3Ch ; Запрос на создание файла list.txt

mov cx,1 ; Атрибуты файла: только чтение

lea dx,addr ; Помещение в DX адреса создаваемого файла

int 21h ; Вызов DOS

; -----------

mov number,ax ; Сохранение номера открытого файла

; -----------

mov cx,2 ; Количество строк в списке

Cycl:

push cx ; Помещение данных с стек

mov ah,0Ah

mov dx,offset message

int 21h ; Вывод строки message на экран

; ------

call fin ; Вызов процедуры

; --------

mov ah,40h ; Запрос на запись данных в файл list.txt

mov bx,number ; Установка в BX номера открытого файла

mov cx,25 ; Запись в CX число записываемых байт

lea dx,message+2 ; Помещение в DX адреса области памяти

int 21h

; --------

mov ah,40h ; Запрос на запись данных в файл list.txt

mov bx,number ; Установка в BX номера открытого файла

mov cx,2 ; Запись в CX число записываемых байт

lea dx,f ; Помещение в DX адреса области памяти

int 21h

; -----

pop cx

loop Cycl

; ----------

mov ax,4C00h

int 21h

CODE ENDS

end main

**Задачи для самостоятельного решения**

1. Дан текстовый файл f. Получите все его строки, содержащие более 20 символов.
2. Дан текстовый файл. Перепишите в файл g все компоненты файла f с заменой в них символа "0" на символ "1" и наоборот.
3. Дан текстовый файл f. Получите самую длинную строку файла. Если в файле имеется несколько строк с наибольшей длиной, то получите одну их них.
4. Дан текстовый файл f. Запишите в "перевернутом" виде строки файла f в файл g. Порядок строк в файле g должен: (а) совпадать с порядком исходных строк в файле f; (б) быть обратным по отношению к порядку строк исходного файла.
5. Дан текстовый файл. Перепишите компоненты файла f в файл g, вставляя в начало каждой строки по одному пробелу. Порядок компонент должен быть сохранен.
6. Даны текстовый файл и строка s. Получите все строки файла f, содержащие в качестве фрагмента заданную строку s.
7. Дан текстовый файл f. Исключите пробелы, стоящие в концах его строк. Результат поместите в файл f1.
8. Даны два текстовых файла f и g. Определите, совпадают ли символы файла f с символами файла g. Если нет, то получите номер первой строки и позицию первого символа в этой строке, в которых файлы f и g отличаются между собой. В случае, когда один из файлов имеет n символов (n 7. 00) и повторяет начало другого (более длинного) файла, ответом должно быть число n+1.
9. Дан текстовый файл f, компоненты которого являются натуральными числами. Количество чисел в файле кратно 4. Первые два числа из каждых четырех задают координаты левого верхнего угла прямоугольника, следующие два числа ─ координаты его правого нижнего угла. Постройте прямоугольники, заданные в файле f.
10. Дан текстовый файл f, содержащий сведения о сотрудниках учреждения, записанные по следующему образцу:

Фамилия Имя Отчество

Фамилия Имя Отчество

1. Запишите эти сведения в текстовом файле g, используя образцы:

(а) Имя Отчество Фамилия (б) Фамилия И.О.

Имя Отчество Фамилия Фамилия И.О.

...

1. Дан текстовый файл f, содержащий список студентов университета (фамилия, имя). Создайте текстовый файл g следующего вида:

Предмет1 Предмет2 Предмет3

Фамилия1 Оценка Оценка Оценка

Фамилия2 Оценка Оценка Оценка

Фамилия3 Оценка Оценка Оценка

Оценки должны вводиться с клавиатуры через пробел.

1. Дан текстовый файл f. Найдите самое длинное слово среди слов, второй буквой которых является "e"; если слов с наибольшей длиной несколько, то найдите последнее. Если таких слов нет вообще, то сообщите об этом.
2. Напишите программу "склеивания" двух файлов.
3. Реализуйте последовательное добавление содержимого нескольких текстовых файлов к пустому файлу с заданным именем.

# Литература

1. Абель П.  Язык Ассемблера для IBM PC и программирование. ─ М.:

Высш.шк., 1992. ─ 444 с.

1. Вирт H.  Алгоритмы + структуры данных = программы. - М.: Мир,

1985. - 406 с.

1. Ирвин К.  Язык ассемблера для процессоров Intel. - М.: Издательский дом "Вильямс", 2005. - 912 с.
2. Использование  Turbo Assembler при разработке программ. - Киев:

Диалектика, 1994. - 288 с.

1. Кудрявцева И.А., Сазонова Н.В., Швецкий М.В.  Система лаборатор-

ных работ по обучению программированию на языке ассемблера для микропроцессоров Intel: Учебное пособие к курсу "Архитектура вычислительных систем". - СПб.: Изд-во "Интерлайн", 2004. - 336 с.

1. Орлов С.Б.  и др. Программа-справочник по системе программирования ТУРБО АССЕМБЛЕР 2.0. Руководство пользователя. - М.: 1990. -­256 с.
2. Пильщиков В.Н.  Программирование на языке ассемблера IBM PC.

-­М.: ДИАЛОГ-МИФИ, 1994. - 288 с.

1. Пирогов В.Ю. 0 Ассемблер и дизассемблирование. - СПб.: БХВ-Петербург, 2006. - 464 с.
2. Сван Т. 0 Освоение Turbo Assembler. - К.: Диалектика, 1996.
3. Таненбаум Э.  Архитектура компьютера. - СПб.: Питер, 2006.­ 699­с.
4. Юров В.И.  Assembler. - СПб: Питер, 2002. - 624 с.
5. Юров В.И.  Assembler: практикум. - СПб: Питер, 2001. - 400 с.