Содержание

Введение

1. Виртуальный лабораторный практикум в инженерном образовании

1.1 Особенности лабораторного практикума для естественнонаучных дисциплин

1.2 Роль технологии виртуальных приборов обучения в техническом вузе

2. Программный эмулятор (виртуальный ПК) Emu8086

2.1 Использование эмулятора Emu8086

2.2 Компиляция кода Ассемблера

2.3 Редактор исходного кода

2.4 Ассемблер

Лабораторная работа № 1

Лабораторная работа № 2

Лабораторная работа № 3

Лабораторная работа № 4

3. Вывод значения байта в десятеричной системе счисления

3.1 Методика выполнения

Лабораторная работа № 5, 6

Заключение

Список использованных источников

## Введение

Сегодня наше государство взяло курс на строительство постиндустриального общества - ступени экономического развития, следующей за периодом индустриализации и характеризующейся опережающим развитием и ростом доли сферы услуг в общей структуре экономики.

Одним из ярких примеров этому является Программа снижения информационного неравенства в Республике Казахстан на 2007-2009 годы разработана в целях реализации Указа Президента Республики Казахстан от 10 ноября 2004 года N 1471 "О Государственной программе формирования "электронного правительства" в Республике Казахстан на 2005-2007 годы".

Концепцией постиндустриального общества является информационное общество - новая историческая фаза развития цивилизации, в которой главными продуктами производства являются информация и знания. Отличительными чертами информационного общества являются:

увеличение роли информации и знаний в жизни общества;

возрастание доли информационных коммуникаций, продуктов и услуг в валовом внутреннем продукте;

создание глобального информационного пространства, обеспечивающего:

а) эффективное информационное взаимодействие людей;

б) их доступ к мировым информационным ресурсам;

в) удовлетворение их потребностей в информационных продуктах и услугах.

Неприятие цифровых технологий, слабое их развитие означает поражение во всемирной гонке за экономическим процветанием и политическим доминированием в будущем информационном обществе. Виртуальная реальность коренным образом меняет производство, образование и жизнь людей, создавая безграничное информационное пространство во всем мире.

Реализация задач "Программы снижения информационного неравенства в Республике Казахстан" обеспечивается выполнением следующих мероприятий… "Методическое обеспечение процесса обучения населения компьютерной грамотности по дистанционной и очной формам обучения".

Методическое обеспечение процесса обучения населения компьютерной грамотности по дистанционным и очным формам обучения - владение навыками пользования компьютером и другими информационными технологиями становятся неотъемлемой частью современной жизни.

В этой связи будет уделено внимание развитию методического обеспечения процесса обучения населения компьютерной грамотности, включающего разработку учебно-методического пособия, электронных учебников (для дистанционного и автономного обучения) на государственном и русском языках, а также тестов и программного обеспечения по контролю знаний.

Лабораторный практикум является обязательным компонентом обучения во всех инженерных курсах, принимаемых в обучении. Во время практикума студенты закрепляют теоретические знания практической работой с микропроцессором, учатся работать с контрольно-измерительной аппаратурой, приобретают исследовательские навыки. В связи с динамическим изменением элементной базы электроники, измерительной аппаратуры, электронный практикум должен своевременно обновляться и совершенствоваться. Дело это трудоемкое и достаточно дорогое, особенно в нынешних условиях.

При всех несомненных достоинствах существующего практикума имеется довольно много замечаний, которые в силу объективных и субъективных трудностей практической реализации не решены на сегодня:

1) Современная полупроводниковая и интегральная элементная база очень чувствительна к перегреву, перенапряжению, статическому электричеству, имеет миниатюрные размеры и поэтому требует сложной, дорогой технологической оснастки для реальной работы с современными электронными схемами. Использование вредных химических веществ при монтаже требует соответствующего оборудования помещения (тоже не дешевого).

2) Работа с современными быстродействующими компонентами требует постоянного обновления дорогой и сложной контрольно-измерительной аппаратуры. Современная аппаратура сложна, требует высокой квалификации исследователя и мало приспособлена для студенческого практикума.

3) Целый ряд исследований невозможно выполнить из-за уникальности необходимой аппаратуры (исследование фазовых характеристик, спектральных характеристик, нелинейных характеристик, исследование влияния температуры на работу электронного устройства и т.д.).

4) В существующем практикуме отсутствует возможность диагностики неисправности электронного устройства, обучения навыкам ремонта электронных схем, пуско-наладочных работ, то есть тех обязательных навыков, которыми обязан владеть электронщик при разработке и эксплуатации электронной аппаратуры.

5) В разработке современной электронной аппаратуры все шире используется вычислительная техника, системы автоматического проектирования, интеллектуальная диагностика работоспособности устройств. Это направление совершенно не представлено в существующем практикуме.

Перечисленные замечания конечно не полностью описывают проблему. Поэтому актуально стоит поиск альтернативных методических направлений обучения электронным дисциплинам.

Одно из таких направлений рассмотрено в данной работе - использование в лабораторном практикуме компьютерного моделирования на базе программного пакета Emu8086.

У этого пакета имеется целый ряд достоинств, привлекающих внимание:

1. Большая библиотека современных электронных компонент, дискретных, интегральных аналоговых, цифровых и смешанных аналогово-цифровых. Библиотека открытая, легко может пополняться новыми элементами, в том числе и отечественными.

2. Богатая библиотека электронных схем, позволяющая использовать готовые практические разработки и легко модернизировать под конкретную задачу. Библиотека открытая, позволяет пополнение как за счет новых разработок, так и за счет подключения библиотек более ранних версий.

Можно получить исчерпывающую информацию о процессах в данном узле.

3. Большие возможности документирования исследования, получение твердой копии как электрической схемы, параметров моделирования, информации с экрана измерительной аппаратуры, хорошо оформленных графических результатов исследования.

4. Не требует знаний по программированию. Требуется лишь знакомство со средой Windows. Интуитивный интерфейс позволяет быстро даже неподготовленному пользователю (буквально за полчаса) познакомится с основами и приступить непосредственно к электронным исследованиям.

5. Нельзя не упомянуть обширный, тщательно подготовленный Help, обеспечивающий как контекстную помощь по меню, компонентам, опциям моделирования, так и общие вопросы моделирования, возможные ошибки.

Достоинств в этом пакете больше, чем перечислено и о них еще будет говориться в процессе разработки лабораторного практикума. Однако то, что перечислено, позволило среди множества известных пакетов электронных выбрать именно Emu8086 как наиболее подходящий для использования в лабораторном практикуме.

До настоящего времени, студенты выполняли лабораторные работы на Лабораторной установке, которая состоит из учебного микропроцессорного комплекта /УМК/, выпускаемого рижским заводом "ВЭФ", набора модулей, подключаемых к ее системной шине и различных периферийных устройств УМК представляет собой учебную микро-ЭВМ, предназначенную для изучения программирования, проектирования и настройки микропроцессорных устройств и систем, выполненных на МП KP580.

Как уже понятно, рижский завод "ВЭФ", уже находиться в другом государстве, морально и физически устарел (уже не выпускается). Микропроцессор МП KP580В уже давно снят с производства, не выпускается, весь мир давно перешел на микропроцессоры фирмы Intel, AMD, Motorolla и т.д.

Последние пять лет - лабораторный практикум выполнялся на эмуляторе МП086, но он был выполнен под операционную систему ДОС, еще современные операционные системы поддерживают эмуляцию ДОС, но следующие ОС перестают поддерживать ДОС.

Поэтому возникла потребность выбора программного продукта, работающего под Windows.

К тому же, необходимой литературы не имеется, нашел ссылки, но они только в общем рассказывают про работу эмулятора Emu8086. Это своего рода - первая работа про данный продукт. Может применяться для проведения лабораторных работ по дисциплине "Основы микропроцессорной техники", "Организация ЭВМ" и т.д.

В своей дипломной работе провожу разработку моделирования лабораторных работ средствами проектирования Emu8086 (под Windows), тем более что программное обеспечение позволяют сделать это моделирование не менее наглядным чем работа на реальных стендах, и ориентированных на инженерные специальности высших учебных заведений.

## 1. Виртуальный лабораторный практикум в инженерном образовании

В учебных планах Казахской Академии транспорта и коммуникаций им. Тынышпаева (КазАТК) время, отводимое под лабораторный практикум, составляет 15-25% от общего объема аудиторных занятий. Примерно такой же по количеству часов (15-27%) лабораторный практикум предусмотрен и в учебных планах Алматинского энергетического. Данные цифры свидетельствуют, что в инженерном образовании лабораторный практикум является важным элементом, без которого немыслима подготовка полноценного специалиста.

Развитие информационных технологий привело к появлению понятия "виртуальный лабораторный практикум" (ВЛП), в основе которого лежит имитационное компьютерное моделирование. Основные способы использования ВЛП в учебном процессе:

в качестве компьютерного "тренажера" для подготовки к выполнению практикума в реальной лаборатории (при этом программы компьютерного и физического экспериментов, как правило, одинаковы);

как дополнение к реальному практикуму, предусматривающее такие компьютерные эксперименты, которые по различным причинам (техническим, финансовым, организационным и т.п.) не могут быть реализованы на физическом оборудовании.

Использование ВЛП в качестве компьютерного "тренажера" позволяет обучающемуся лучше подготовиться к проведению физического эксперимента, глубже уяснить исследуемые эффекты, приобрести навыки работы с измерительными приборами (в случае, если виртуальный практикум включает компьютерные модели измерительных приборов, близкие по своим свойствам к свойствам реальных приборов). Обычно такой подход можно рекомендовать для студентов заочно-дистанционной формы обучения, поскольку он не только способствует лучшему усвоению изучаемого материала, но и позволяет сократить продолжительность выполнения практикума в реальной лаборатории в период пребывания в стенах учебного заведения.

Если ВЛП используется как дополнение к реальному практикуму, то он должен быть ориентирован на проведение исследований повышенного уровня сложности или исследований, требующих дорогостоящего оборудования, которым не располагает университет.

По технологиям создания ВЛП можно выделить следующие основные варианты.

1. ВЛП на основе универсальных пакетов программ, обеспечивающих возможность применения в широком спектре предметных областей. Примером может служить система LabVIEW фирмы National Instruments. Универсальные пакеты содержат обширные библиотеки элементов, предназначенных для разработки виртуальных интерфейсов физических приборов и лабораторных установок.

2. ВЛП на основе специализированных предметно-ориентированных пакетов программ, предназначенных для сравнительно ограниченного набора предметных областей. В качестве примера отметим систему Multisim фирмы Electronics Workbench, созданную для моделирования электронных схем, систему ChemOffice фирмы CambridgeSoft, предназначенную для моделирования и анализа химических процессов и т.п. Так же как и в предыдущем случае, программное обеспечение данного класса представляет собой универсальную среду, предназначенную для решения прикладных задач пользователя.

3. ВЛП на основе Java-апплетов. В отличие от предыдущих случаев, где пользователь (преподаватель) обычно работает в режиме графического программирования, процесс создания Java-апплетов является гораздо более трудоемким и требует программирования в кодах. Тем не менее, данная технология имеет и определенные достоинства, особенно когда речь заходит о ВЛП, предназначенном для сетевого применения. Так, например, приложения, создаваемые в системе LabVIEW, занимают примерно 2,5-3 Мбайт памяти, а типичный размер виртуальной лабораторной работы на основе Java-апплета - десятки-сотни килобайт.

Развитие сетевых компьютерных технологий привело к появлению лабораторного практикума, реализуемого в режиме удаленного доступа к реальному оборудованию. Учитывая, что реализация удаленного доступа к реальному оборудованию связана с решением ряда проблем (необходимостью сопряжения лабораторного макета с ПК, обеспечением надежной защиты оборудования от возникновения аварийных режимов, низкой эффективностью использования оборудования из-за невозможности в ряде случаев реализовать коллективный доступ и т.д.), данная технология имеет достаточно много оппонентов. Тем не менее, она тоже имеет свое право на существование, а в ряде случаев имеет очевидные преимущества перед ВЛП.

Важным является вопрос, не является ли ВЛП альтернативой реальному лабораторному практикуму. С одной стороны, современные компьютерные технологии имитационного моделирования позволяют создавать виртуальные интерфейсы реального лабораторного оборудования, воспроизводящие и внешний вид, и его параметры с очень высокой точностью. С другой стороны, поддержание в рабочем состоянии и своевременное обновление лабораторного оборудования, включая и измерительные приборы, требует немалых финансовых средств.

Тем не менее, любой, даже сколь угодно высококлассный ВЛП, в большинстве случаев не заменит по своему обучающему воздействию, оказываемому на студента, работу с реальным оборудованием.

Это осознают и сами студенты, что подтверждается результатами анкетирования, проводившегося на кафедре АТС КазАТК в рамках педагогического эксперимента по апробации ВЛП по дисциплине "Схемотехника аналоговых электронных устройств". В целом, дав положительную оценку новому ВЛП (84% опрошенных), 54% студентов, участвовавших в опросе, ответили, что считают работу с реальным лабораторным оборудованием более полезной по сравнению с виртуальными лабораторными работами.

## 1.1 Особенности лабораторного практикума для естественнонаучных дисциплин

На мой взгляд, основная проблема заключается в различии способов подачи информации, преобладающих в системе естественнонаучного и гуманитарного образования.

В соответствии с одной из существующих концепций, информация воспринимается человеком, проходя следующие этапы: сенсорно-моторный, символьный, логический и лингвистический.

На первом этапе происходит чувственное восприятие информации, на втором осуществляется ее преобразование в образы, на третьем - ее осмысление, на четвертом - информация фиксируется в сознании через "слово-образ".

В преподавании же естественнонаучных дисциплин всегда присутствовал сенсорно-моторный этап. В этом большую роль играют лабораторные практикумы и учебные эксперименты.

Учебный эксперимент является одним из важнейших методов обучения, источником знаний и средством наглядности одновременно. Он может использоваться в качестве введения к той или иной теме курса (мотивация), как иллюстрация к объяснению нового материала (восприятие и осмысление), как повторение или обобщение пройденного (интериоризация) или как контроль приобретенных знаний, умений, навыков, т.е. на всех этапах процесса обучения.

Различают следующие виды учебного эксперимента:

Демонстрационный эксперимент. (Проводится в ходе объяснения для иллюстрации законов и явлений и их применения, демонстрации принципов действия технических установок, а также для знакомства учащихся с фундаментальными опытами. Готовится и выполняется только преподавателем).

Фронтальные лабораторные работы, опыты и наблюдения. (Выполняются на уроке всеми учащимися одновременно на однотипном оборудовании, под непосредственным руководством преподавателя).

Практикум. (Является формой самостоятельной работы учащихся, готовится учащимися заранее и выполняется по письменной инструкции).

Внеурочные опыты и наблюдения. (Разновидность домашнего задания. Используется для актуализации теоретических знаний или как мотивация к изучению нового материала).

Методически грамотно организованный эксперимент способствует как формированию практических умений, так и активизации теоретических знаний, полученных ранее. В процесс обучения вовлекаются различные каналы восприятия (слух, зрение, осязание, обоняние и т.д.). Это позволяет организовать полученную информацию, как систему ярких образов, и заложить ее в долговременную память.

С другой стороны, подготовка и проведение лабораторных работ являются довольно непростым делом и требуют от учителя знания некоторых методических особенностей, в значительной степени зависит от наличия тех или иных приборов и инструментов.

Частично эти и другие проблемы можно решить, используя при обучении компьютерный эксперимент. Под данным термином мы понимаем лабораторную работу, проводимую полностью на компьютере, без использования других технических средств обучения.

Некоторые эксперименты, проводимые в области компьютеризации учебного процесса, показывают, что применение компьютерного эксперимента позволяет существенно сократить время, которое тратится на рутинную работу (варьирование параметров эксперимента путем изменения схемы установки, расчета результатов измерения и т.д.), тем самым, высвобождая время для более серьезного уяснения целей и задач проводимого эксперимента.

Кроме того, появляется возможность продемонстрировать опыты, которые невозможно провести в условиях учебного кабинета.

Отметим характерные особенности компьютерного эксперимента.

Формой работы является диалог учащегося с компьютером. При этом в функции компьютера входит:

Реализация программными средствами модели изучаемого объекта, установки, процесса или ситуации

Имитация средств измерения и выполнение рутинной части обработки измерений.

Оценка действий учащегося.

Функции учащегося (несколько отличаются от его функций в традиционном эксперименте):

Анализ информации, которую программа выдает на экран дисплея.

Выбор условий эксперимента.

Проведение серий экспериментов для достижения цели, сформулированной в начале работы.

Корректировка последующих шагов с целью получения более высокой оценки и решения задачи более рациональным способом.

Разумеется, что возможна реализация эксперимента только в его компьютерном варианте. Однако при всей привлекательности и несомненной дидактической выгоде компьютерного эксперимента в обучении остаются нерешенными некоторые проблемы.

Во-первых, восприятие информации учащимся уже существенным образом отличается от того, как это происходит при выполнении традиционной лабораторной работы. В частности, сенсорно-моторный этап практически отсутствует. Без данного этапа восприятие не может быть полноценным. Следовательно, неполноценным может оказаться и преподавание соответствующей дисциплины.

Во-вторых, возникает проблема получения политехнических навыков работы с реальными приборами и установками.

Очень важна и все еще мало исследована проблема формирования адекватного представления о мире при работе с нереальными объектами. Возможно, эту проблему можно решить, используя в процессе обучения программные продукты, максимально точно отображающие процессы и явления, происходящие в реальном мире. В этом смысле особое внимание стоит обратить на средства виртуальной реальности.

Таким образом, одной из основных проблем преподавания предметов естественнонаучного цикла является отсутствие возможности реальной постановки учебного, и лабораторного эксперимента.

Хотя в настоящее время имеются разработки виртуальных лабораторных практикумов, однако окончательно решение проблемы требует пристального внимания специалистов различных профилей, в том числе и психолого-педагогического.

## 1.2 Роль технологии виртуальных приборов обучения в техническом вузе

Лабораторная н экспериментальная база вуза достаточно консервативна в силу финансовых и материальных ограничений. Поэтому она практически не успевает отслеживать бурное развитие техники н неизбежно морально устаревает. В современных быстро изменяющихся условиях, в которых сетка специальностей и специализаций должна быстро и непрерывно адаптироваться к запросам промышленности, реальная лабораторная и экспериментальная база вуза не *в* состоянии поддерживать учебный процесс на должном уровне.

Следует также отмстить, что отставание материальной базы от требований жизни является не только вузовской проблемой, но и обшей проблемой в промышленности и для различных организаций и фирм, занимающихся разработками ноной техники.

Современная *технология виртуальных приборов* позволяет существенно сократить этот разрыв и сэкономить значительные финансовые ресурсы, не снижая качества обучения. Мировая вузовская практика подтверждает устойчивую и усиливающуюся тенденцию продвижения виртуальных технологий в учебном процессе.

*Программная среда Emu8086,* поддерживающая технологию виртуальных приборов, и соответствующее аппаратное обеспечение позволяют модернизировать учебные лаборатории гибким, программно перестраиваемым измерительным оборудованием или модернизировать имеющиеся средства измерений практически любой сложности, а также внедрять автоматизированные измерительные системы и станции для учебного процесса и комплексных исследований н вузовской науке.

Для практической реализации технологии *виртуальных приборов и систем измерения* в учебном процессе достаточно приобрести недорогую стандартную плату аналогового ввода-вывода, основными составляющими которой являются многоканальный коммутатор и аналого-цифровой преобразователь.

*Технико-экономические* оценки показывают, что рабочее место, включающее минимальный типовой набор традиционных измерительных приборов (осциллограф, импульсный и низкочастотный генераторы, частотомер, цифровой тестер), по текущим каталогам цен обходится в три раза дороже по сравнению с затратами на виртуальный эквивалент, включая приобретение современной персональной ЭВМ типа Pentium IV.

Поскольку компьютеры так или иначе приобретаются, а также с учетом того, что па кафедрах уже имеется парк компьютеров, модернизация оборудования лабораторий на базе виртуальных технологий будет стоить в 5-10 раз дешевле.

Так, оценочные расчеты, проведенные для общетехнической кафедры, показывают, что для полной модернизации приборного оборудования пяти учебных лабораторий, в каждой из которых организовано по 5 рабочих мест, необходимо затратить около 8,0 млн. тг. (цены на 2006 г), если модернизацию осуществлять традиционное покупкой необходимых "реальных" приборов.

При использовании технологии виртуальных приборов, с учетом уже имеющихся на кафедре компьютеров, модернизация обойдется примерно в 8 раз дешевле - в 1 000 млн. тг. Последние цифры уже приемлемы и подъемны, особенно при поэтапном подходе к модернизации.

Например, при ежегодном вложении по 400 тыс. тг, полное переоснащение лабораторий можно завершить в течение 2.5-3 лет.

## 2. Программный эмулятор (виртуальный ПК) Emu8086

Все для изучения ассемблера в одном пакете! Emu8086 сочетает в себе мощный редактор исходного кода, ассемблер, дизассемблер, программный эмулятор (виртуальный ПК) с отладчиком и поэтапное обучение.

Эта программа чрезвычайно полезна для тех, кто только начинает изучать ассемблер. Она компилирует исходный код и выполняет его с помощью эмулятора шаг за шагом.

Визуальный интерфейс очень прост в работе. Вы можете наблюдать регистры, флаги и память во время выполнения вашей программы.

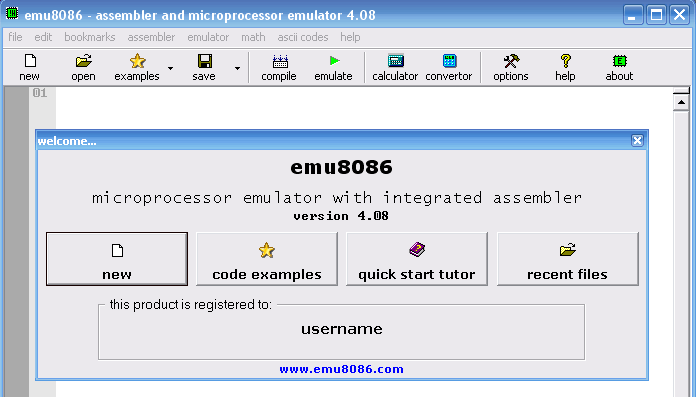
Арифметико-логическое устройство (АЛУ) показывает внутреннюю работу центрального процессора (CPU).

Эмулятор выполняет программы на виртуальном ПК, который полностью исключает возможность доступа из вашей программы к реальным аппаратным средствам, таким как жесткие диски и память. Так как ваш код ассемблера выполняется на виртуальной машине, то отладка становится более легкой.

Машинный код 8086 полностью совместим со всеми последовавшими за ним поколениями микропроцессоров Intel, включая Pentium II и Pentium 4, и я уверен, что Pentium 5 будет также поддерживать команды 8086. Это делает код 8086 очень привлекательным, так как он выполняется как на старых, так и на современных компьютерных системах. Другим преимуществом является то, что набор команд 8086 сравнительно невелик, и поэтому изучить его будет легче.

*Emu8086* имеет более легкий синтаксис, чем любые другие ассемблеры, но будет генерировать программу, которая сможет быть выполнена на любом компьютере, поддерживающем машинный код 8086. Это большой плюс для начинающих! Примечание: Если вы не используете *Emu8086* для компиляции кода, вы не сможете выполнить ваш исходный код в пошаговом режиме.

*Примечание:* Если при попытке запустить вашу программу, вам выдаются сообщения о том, что те или иные команды не поддерживаются, то вы не сможете выполнить эту программу в отладочном режиме. Однако ее можно просто откомпилировать и запустить обычным образом (не под управлением Emu8086).



Как начать?

Запустите *Emu8086*, выбрав ее значок в меню "Пуск", или непосредственно запустите приложение Emu8086. exe.

Выберите "Samples (примеры)" из меню "File".

Щелкните кнопку [Compile and Emulate] (или нажмите клавишу F5).

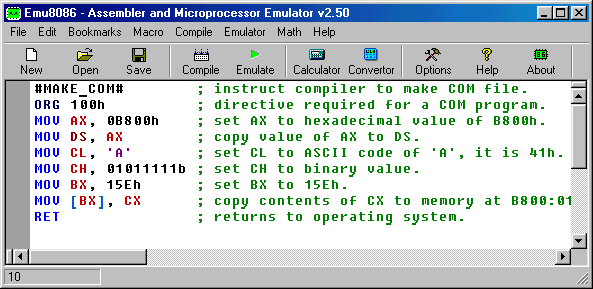
Щелкните кнопку [Single Step] (пошаговый режим) (или нажмите клавишу F8), и наблюдайте за выполнением кода.

Попытайтесь открыть другие примеры. Все примеры имеют подробные комментарии, так что для обучения они весьма полезны.

## 2.1 Использование эмулятора Emu8086

Напечатайте ваш код внутри текстовой области, и после щелчка кнопки [Compile]. Вас спросят о месте, где сохранить компилируемый файл.

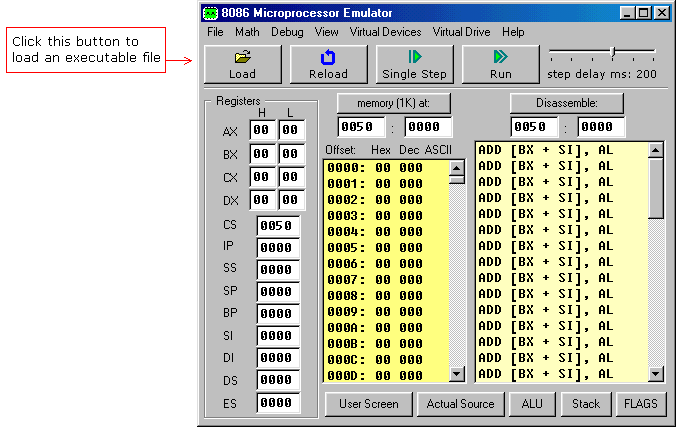
После успешной компиляции Вы можете щелкать кнопку [Emulate], чтобы загрузить компилируемый файл в эмуляторе.



Если Вы хотите загрузить ваш код в эмулятор, только щелкните кнопку "Emulate".

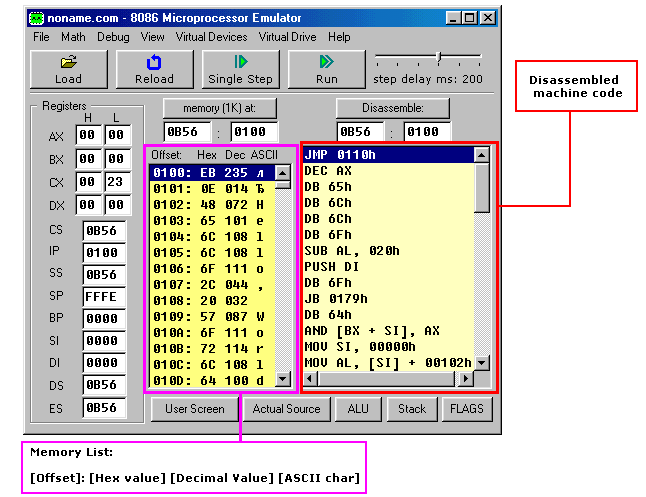


Но Вы также можете использовать эмулятор, чтобы загрузить executables, даже если Вы не имеете первоначального исходного текста. Выберите " Show Emulator", меню "Emulator".



Попытка, загружающая файлы от папки "MyBuild". Если не имеется никаких файлов в "MyBuild" папке, возвращают исходному редактору, выбирают *Samples* от меню *File*, загружают любой образец, выбирают и затем загружаются в эмулятор.

2) произвести исследование области памяти согласно варианта задания. В результате исследования необходимо заполнить табл.1.1;



В памяти перечисляют первую строку - смещение, вторая строка - значение hexadecimal, третья строка - десятичное значение, и последняя строка - значение символа ASCII.

Кнопка [Single Step] выполняет команды, один за другим останавливающие после каждой команды.

[Run] кнопка выполняет команды один за другим с задержкой, установленной задержкой шага между командами.

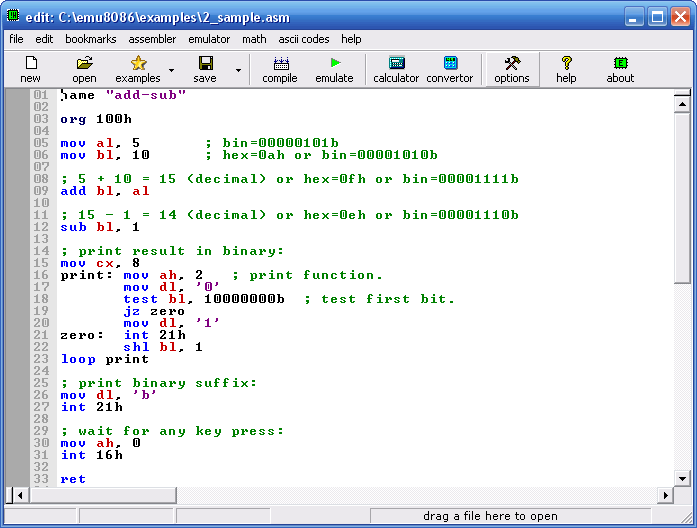
Дважды щелкните на текстовых полях регистра, открывается окно "Extended Viewer " со значением того регистра, преобразованного ко всем возможным формам. Вы можете изменять значение регистра непосредственно в этом окне.

Дважды щелкните на элементе списка памяти, открывается " Extended Viewer" со значением WORD, загруженным со списка памяти в выбранном местоположении. Менее существенный байт - в младшем адресе: LOW BYTE загружен от выбранной позиции и HIGH BYTE от следующего адреса памяти. Вы можете изменять значение слова памяти непосредственно в окне " Extended Viewer",

Вы можете изменять значения регистров во времени выполнения, печатая по существующим значениям.

Кнопка [Flags] позволяет Вам рассматривать и изменять флажки на времени выполнения.

## 2.2 Компиляция кода Ассемблера



Напечатайте ваш код внутри текстовой области и щелкните кнопку [Compile]. Вас спросят, где сохранить откомпилированный файл. После завершения компиляции вы можете щелкнуть кнопку [Emulate] для загрузки откомпилированного файла в эмулятор.

Директивы, определяющие тип исполнимого файла:

#MAKE\_COM#

#MAKE\_BIN#

#MAKE\_BOOT#

#MAKE\_EXE#

Вы можете вставить эти директивы в исходный код для определения нужного вам типа исполнимого файла. В том случае, если компилятор не найдет ни одной из этих директив, он спросит у вас *тип файла* перед его созданием.

Описание типов исполнимых файлов:

#MAKE\_COM# - самый старый и самый простой формат исполнимого файла. Такие файлы загружаются с префиксом 100h (256 байтов). Выберите СОМ Template из меню New, если вы планируете компилировать COM-файл. Директива компилятора ORG 100h должна быть добавлена перед кодом. Выполнение всегда начинается с первого байта файла. Поддерживается командной строкой DOS и Windows.

#MAKE\_EXE# - более "продвинутый" формат исполнимого файла. Не ограничены размер и количество сегментов. Сегмент стека должен быть определен в программе. Вы можете выбрать EXE Template из меню New для создания простой ЕХЕ-программы с определенными сегментмами Данных, Стека и Кода. Точка входа (где начинается выполнение) определяется программистом. Поддерживается командной строкой DOS и Windows.

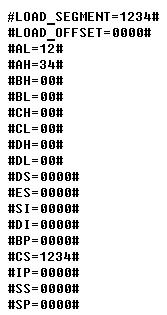
#MAKE\_BIN# - простой исполнимый файл. Вы можете определить значения всех регистров, сегмент и смещение для области памяти, куда этот файл будет загружен. Если загрузить файл "MY. BIN" в эмулятор, он будет виден для файла - "MY. BINF" и загрузится файл "MY. BIN" в местоположение, определенное в файле "MY. BINF". Регистры также установятся с учетом информации из этого файла (откройте этот файл в редакторе для изменения или изучения). В том случае, если эмулятор не найдет файл "MY. BINF", будет использоваться текущие значения регистров и файл "MY. BIN" загрузится в текущий CS: IP.

Выполнение начинается со значения в CS: IP.

Этот тип файла уникален для *Emu8086*.

Файл ". BINF создается автоматически компилятором, если он находит директиву #MAKE\_BIN#.

ПРЕДУПРЕЖДЕНИЕ! если файл ". binf" существует, то он будет перезаписан!



Значения должны быть шестнадцатиричными! Если эти значения не определены, то они устанавливаются по умолчанию:

LOAD\_SEGMENT = 0100 LOAD\_OFFSET = 0000 CS = ES = SS = DS = 0100 IP = 0000 Если LOAD\_SEGMENT и LOAD\_OFFSET не определены, то используются значения CS и IP, и наоборот.

Если значение Load to offset не равно нулю (0000), то ORG???? h должна быть добавлена в файл. BIN, где???? h - это *смещение для загрузки*. Это должно быть сделано для того, чтобы компилятор мог вычислить правильные адреса.

#MAKE\_BOOT# - эта директива копирует первую дорожку дискеты (загрузочный сектор).

Вы можете записать загрузочный сектор виртуального дисковода (FLOPPY\_0) через меню эмулятора:

[Virtual Drive] - > [Write 512 bytes at 7C00 to Boot Sector] Сначала вы должны откомпилировать ". boot"-файл, а затем загрузить его в эмулятор (см. "micro-os\_loader. asm" и "micro-os\_kernel. asm" в разделе "Samples").

Затем выберите в меню [Virtual Drive] - > [Boot from Floppy], чтобы загрузить эмулятор с виртуального дисковода.

Затем, если вам любопытно, вы можете записать виртуальную дискету на реальную дискету и загрузить с нее ваш компьютер. Я рекомендую использовать "RawWrite for Windows" с: http://uranus. it. swin.edu. au/~jn/linux/rawwrite. htm (учтите, что "micro-os\_loader. asm" не использует MS-DOS-совместимый загрузочный сектор, так что лучше использовать чистую дискету, хотя она должна быть отформатирована IBM (MS-DOS)).

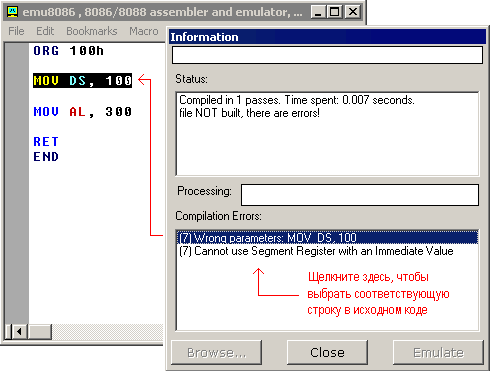
Директива компилятора ORG 7C00h должна быть добавлена перед кодом, если компьютер начинает загружаться с первой дорожки дискеты в адрес 0000: 7C00.

Размер. BOOT-файла должен быть менее 512 байтов (ограничен размером сектора дискеты).

Выполнение всегда начинается с первого байта файла. Этот тип файла уникален для эмулятора *Emu8086*.

Обработка ошибок

Компилятор выводит отчет об ошибках в отдельном окне:



MOV DS, 100 - это недопустимая команда, потому что в сегментный регистр нельзя устанавливать непосредственное значение - должны использоваться регистры общего назначения:

MOV AX, 100 MOV DS, AX MOV AL, 300 - это недопустимая команда, т.к регистр AL имеет только 8 битов, и его максимальное значение 255 (или 11111111b), а минимальное - 128.

Компилятор делает несколько проходов перед генерацией правильного машинного кода. Если он находит ошибку и не выполняет требуемое количество проходов, он может выдать неправильное сообщение об ошибке. Например:

#make\_COM#

ORG 100h

MOV AX, 0

MOV CX, 5

m1: INC AX

LOOP m1; это не настоящая ошибка!

MOV AL, 0FFFFh; ошибка здесь.

RET

Список генерируемых ошибок:

(7) Condition Jump out of range (Условие перехода за пределами диапазона) !: LOOP m1 (9) Wrong parameters (Неправильнве параметры): MOV AL, 0FFFFh (9) Operands do not match (Операнды не соответствуют): Second operand is over 8 bits (Второй операнд более 8 битов) ! Первое сообщение (7) - неправильное. Компилятор не закончил вычисление смещений для меток, поэтому он думает, что смещение метки m1 - это 0000. Этот адрес лежит за пределами диапазона, т.к мы начинаем со смещения 100h.

Внесите исправления в эту строку: MOV AL, 0FFFFh (AL не может содержать значение 0FFFFh). Это устранит обе ошибки! Например:

#make\_COM#

ORG 100h

MOV AX, 0

MOV CX, 5

m1: INC AX

LOOP m1; тот же самый код без ошибки!

MOV AL, 0FFh; все!

RET

При сохранении компилируемого файла, компилятор также сохраняет 2 других файла, которые используются эмулятором для отображения фактического исходного кода при его исполнении и выборе соответствующей строки.

asm - этот файл содержит оригинальный исходный код, который был использован для создания исполнимого файла.

debug - этот файл содержит информацию, которая позволяет эмулятору выбирать строки оригинального исходного кода во время выполнения машинного кода.

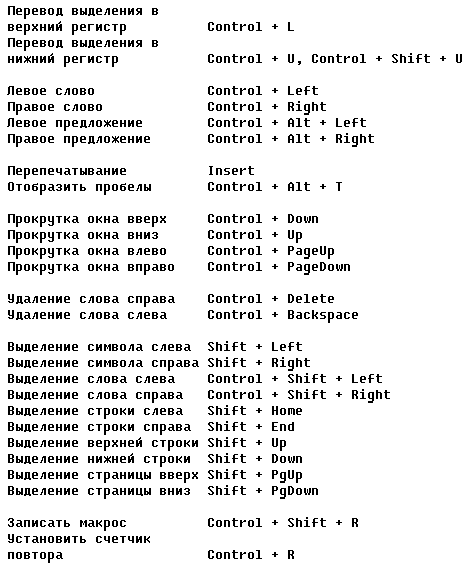
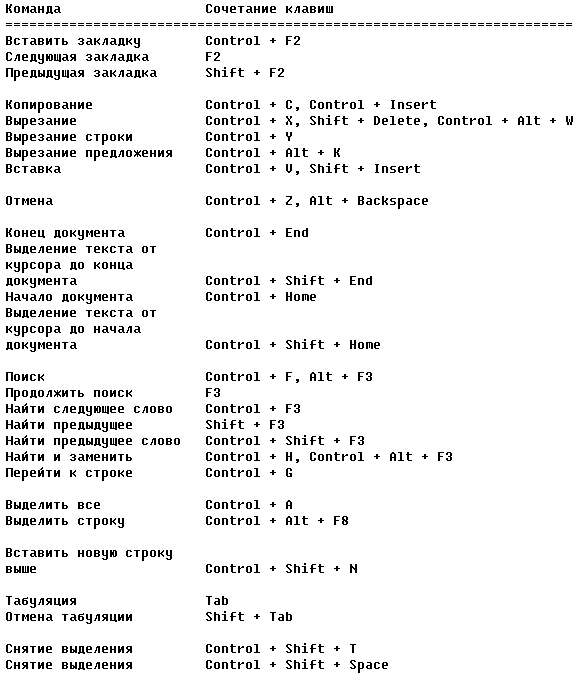
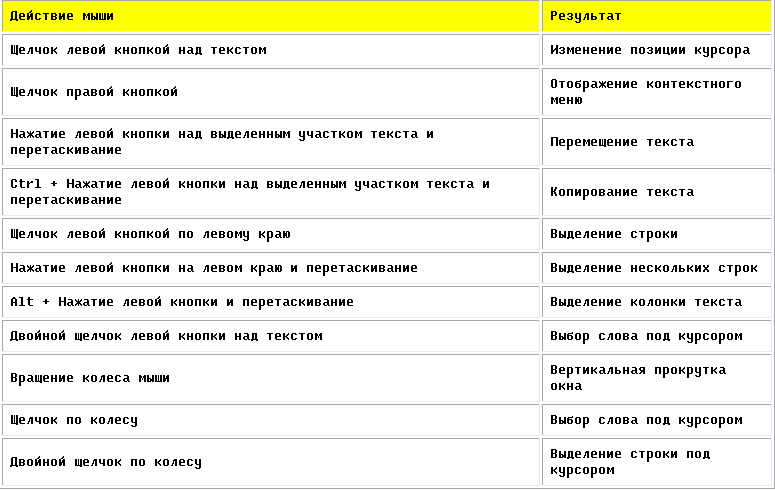
symbol - Таблица символов. Она содержит информацию, которая позволяет отображать окно "Variables" (Переменные). Это текстовый файл, так что вы можете посмотреть его из текстового редактора.

binf - этот файл содержит информацию, которая используется эмулятором для загрузки BIN-файла в указанное местоположение, и установки значений регистров предшествующих выполнению (создается только в том случае, если исполнимый файл - это BIN-файл).

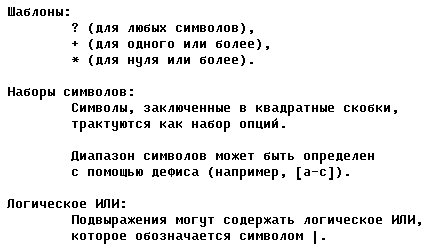
## 2.3 Редактор исходного кода

Использование мыши Редактор поддерживает следующие действия мыши:

Горячие клавиши редактора:



Допустимые правила синтаксиса выражений для поиска и замены



Если появятся проблемы с редактором исходного кода, вы можете вручную скопировать файл "cmax20. ocx" из каталога, где установлена программа, в папку Windows\System или Windows\System32, заменив существующую версию этого файла (после этого может потребоваться перезагрузка компьютера

## 2.4 Ассемблер

*Что такое ассемблер? Ассемблер -* это программа, преобразовывающая исходный текст программы, написанной на языке ассемблера, в машинный код. Дополнительно ассемблер может создавать листинг программы с номерами строк, адресами переменных, операторами исходного языка и таблицей перекрестных ссылок символов и переменных, используемых в программе. Совместно с ассемблером используется программа, называемая *компоновщиком (linker)* или *редактором связей (linkage editor).* Она объединяет отдельные файлы, созданные ассемблером, в единую исполняемую программу. В блок базовых программ входит также *отладчик (debugger)* позволяющий программисту пошагово выполнять программу, проверять и изменять содержимое памяти.

*Какие типы программ мы будем создавать?* Мы будем писать два основных типа программ, которые перечислены ниже.

*16-разрядные программы для реального режима адресации.* Эти программы предназначены для выполнения в системе MS DOS либо в среде эмулятора DOS под Linux. Большинство примеров из этой книги можно адаптировать для выполнения в реальном режиме адресации. О программировании для реального режима адресации речь пойдет в многочисленных примечаниях книги. Кроме того, две главы полностью посвящены выводу текстовой и графической информации на экран монитора в режиме MS DOS.

*32-разрядные программы для защищенного режима.* Эти программы предназначены для запуска в окне текстового терминала (консоли) вереде операционной системы Microsoft Windows. С их помощью вы сможете отобразить на экране монитора как текстовые, так и графические данные.

*Как язык ассемблера связан с машинным кодом?* Во-первых, *машинный код -* это набор чисел, которые интерпретируются центральным процессором компьютера и определяют выполняемые им действия. Например, все процессоры Intel семейства IA-32 имеют совместимый между собой машинный код. Машинный код состоит исключительно из двоичных чисел. Во-вторых, *язык ассемблера* состоит из набора операторов, понятных человеку. Каждый оператор начинается с короткого мнемонического обозначения выполняемых процессором действий, например ADD (сложить), MOV (переслать), SUB (вычесть) или CALL (вызвать). Язык ассемблера *однозначно* связан с машинным кодом. Это значит, что *каждый* оператор языка ассемблера соответствует *одной* команде машинного кода.

*Какое отношение имеет язык ассемблера к языкам высокого уровня, таким как* C++ *или* Java? Языки высокого уровня, такие как C++ или Java, не имеют однозначного соответствия с языком ассемблера и, следовательно, с машинным кодом. Например, один оператор языка C++ транслируется в несколько операторов языка ассемблера или несколько машинных команд. Давайте посмотрим, как происходит процесс трансляции оператора языка C++ в машинный код. Поскольку анализировать двоичный машинный код очень трудно, вместо него мы рассмотрим эквивалентные операторы языка ассемблера. В приведенном ниже операторе языка C++ выполняются две арифметические операции и полученный результат присваивается переменной. Предположим, что существуют целочисленные переменные X и Y:

X = (У + 4) 3;

В результате трансляции получится приведенный ниже набор ассемблерных команд. Обратите внимание, что одному оператору языка высокого уровня соответствует несколько команд языка ассемблера, так как последний однозначно связан с машинным кодом:

mov еах, У; Загрузить значение переменной Y в регистр ЕАХ

add eax,4; Прибавить число 4 к регистру ЕАХ

mov ebx,3; Загрузить число 3 в регистр ЕВХ

imul ebx; Умножить содержимое регистра ЕАХ на содержимое ЕВХ

mov Х, еах; Переслать содержимое регистра ЕАХ в переменную X

С точки зрения программиста регистры - это обычные переменные, которым присвоены стандартные имена, находящиеся внутри центрального процессора. Обычно регистры используются в качестве одного из операндов при выполнении команд процессором.

С помощью этого примера мы вовсе не хотели показать, что язык C++ "лучше" или что он мощнее языка ассемблера. Нашей целью было продемонстрировать, как один оператор языка высокого уровня порождает несколько команд языка ассемблера. Как вы уже знаете, язык ассемблера однозначно связан с машинным кодом. Последний состоит из набора чисел, с помощью которых закодированы выполняемые процессором действия.

*Являются ли программы на языке ассемблера переносимыми?* Важным отличием языка ассемблера от языков высокого уровня является то, что написанные на нем программы не являются переносимыми. Говорят, что язык программирования является *переносимым {portable),* если написанные на нем программы можно скомпилировать и запустить на разных компьютерных платформах. Например, программы, написанные на языке C++, Moiyr быть скомпилированы и запушены практически на любом компьютере и п любой операционной системе при условии, что в них не используются вызовы библиотечных функций, характерные для конкретной операционной системы. Основным отличием языка Java является то, что написанные на нем программы *после* компиляции могут выполняться в *любой* компьютерной системе, для которой *существует* реализация виртуальной машины Java.

Учитывая изложенные выше моменты, язык ассемблера не может быть переносимым по определению, поскольку он тесно связан с архитектурой процессоров определенного семейства. Таким образом, на сегодняшний день существует несколько совершенно разных языков ассемблера. Каждый из них привязан либо к конкретному семейству процессоров, либо к конкретной архитектуре компьютера. Среди них можно выделить семейство процессоров Motorola 68x00, Intel IA-32, SUN Sparc, VAX и IBM-370. Команды в языке ассемблера соответствуют командам конкретного процессора. Например, язык ассемблера, рассмотренный в этой книге, предназначен для программирования только процессоров Intel, принадлежащих семейству 1А-32.

*Зачем изучать язык ассемблера?* А если взять хорошую книгу, в которой описана архитектура конкретного компьютера и структура его процессора? Неужели она не заменит это описание программирования на языке ассемблера?

Я уверен, что вы связаны с написанием программ для *встраиваемых компьютерных систем.* Подобные программы обычно пишут на языках С, Java или ассемблере, после чего полученный машинный код записывают в запоминающее устройство (постоянное или перепрограммируемое) микроконтроллера. Затем сам микроконтроллер устанавливают в управляемое им устройство. В качестве примера встраиваемых устройств можно привести системы питания и зажигания автомобилей, системы управления кондиционерами, охранные системы, системы управления полетами, электронные записные книжки, модемы, принтеры и другие "умные" устройства, содержащие встроенный микропроцессор.

В большинстве специализированных игровых приставок к программам предъявляются довольно жесткие требования к объему используемой в них памяти и к быстродействию самих программ. Все это требует высокой степени оптимизации этих программ как по скорости, так и по используемой ими памяти. Поэтому программисты, занимающиеся написанием кода для таких приставок, должны учитывать особенности их аппаратного обеспечения. В подобных ситуациях они часто в качестве средства разработки выбирают язык ассемблера, поскольку он позволяет им получить полный контроль над процессом создания машинного кода.

Для прикладного программиста язык ассемблера поможет преодолеть ограничения, накладываемые используемым ими языком высокого уровня в плане выполнения определенных типов операций. Например, в языке Microsoft Visual Basic обработка строковых данных выполняется крайне неэффективно. Поэтому для выполнения операций со строками, такими как шифрование данных и обработка битовых строк, программисты обычно используют подпрограммы, написанные на языке C++ или ассемблере и размешенные в DLL (*Dynamic Link Libraries,* или *динамически загружаемые библиотеки).*

Если вы связаны с разработкой специализированного оборудования, то наверняка вам придется написать драйвер устройства, управляющий работой того оборудования, которое выпускает ваша фирма. *Драйверы устройств (device drivers) -* это низкоуровневые системные программы, напрямую взаимодействующие с обслуживаемыми ими устройствами. В задачу драйвера входит преобразование обобщенных запросов, посылаемых операционной системой на конкретное устройство, в последовательность низкоуровневых команд, характерных для данного конкретного устройства. Например, производители принтеров комплектуют каждое выпускаемое ими устройство отдельными драйверами для каждой из поддерживаемых операционных систем, таких как Microsoft Windows, Mac OS, Linux и др.

*Существуют ли какие-либо правила в языке ассемблера?* Да, конечно, в языке ассемблера приняты несколько правил, обусловленные внутренней физической структурой самого процессора и его системой команд. Например, два операнда, используемые в одной команде, должны иметь одинаковый размер. Тем не менее, в языке ассемблера гораздо меньше правил, чем, например, вС++.

В программах на языке ассемблера можно легко обойти любые ограничения, принятые в языках высокого уровня. Например, в языке C++ не разрешается присваивать значение указателя одного типа указателю другого типа. Как правило, в этом ограничении нет ничего плохого, поскольку оно позволяет избежать логических ошибок в программах.

Опытный программист может найти способ, как преодолеть это ограничение, однако полученный в результате код будет слишком сложным. В отличие от C++, язык ассемблера не накладывает никаких ограничений относительно указателей. Здесь операции присваивания значений указателям целиком и полностью определяются программистом.

Естественно, что цена такой свободы чрезвычайно высока: программист тратит очень много времени на отладку ассемблерных программ на уровне машинного кода.

## Лабораторная работа № 1

Ознакомление с работой эмулятора Emu8086

Цель работы: ознакомление со структурой учебной микроЭВМ (эмулятора Emu8086), органами управления и режимами ее работы.

*1. Краткие теоретические сведения*

*1.1 Структура ассемблерной программы*

Каждый язык программирования имеет свои особенности. Язык ассемблера - не исключение. Традиционно первая программа выводит приветственное сообщение на экран ‘Hello World’.

В отличие от многих современных языков программирования в ассемблерной программе каждая команда располагается на ОТДЕЛЬНОЙ СТРОКЕ. Нельзя разместить несколько команд на одной строке. Не принято, также, разбивать одну команду на несколько строк.

Язык ассемблера является РЕГИСТРОНЕЧУВСТВИТЕЛЬНЫМ.Т. е. в большинстве случаев нет разницы между большими и малыми буквами. Команда может быть ДИРЕКТИВОЙ - указанием транслятору. Они выполняются в процессе превращения программы в машинный код. Многие директивы начинаются с точки. Для удобства чтения программы они обычно пишутся БОЛЬШИМИ БУКВАМИ. Кроме директив еще бывают ИНСТРУКЦИИ - команды процессору. Именно они и будут составлять машинный код программы.

Нужно отметить, что понятие "машинного кода" очень условно. Часто оно обозначает просто содержимое выполняемого файла, хранящего кроме собственно машинных команд еще и данные. В нашем случае это будет текст выводимого сообщения "Hello".

*1.2 Особенности создания ассемблерной программы в среде DOS средствами TASM и MASM*

Язык ассемблера является самым низкоуровневым языком программирования.Т. е. он ближе любых других приближен к архитектуре ЭВМ и ее аппаратным возможностям, позволяя получить к ним полный доступ. В отличие от языков высокого уровня (ЯВУ) ассемблерная программа содержит только тот код, который ВВЕЛ ПРОГРАММИСТ. Никаких дополнительных "обвязок". Вся ответственность за "логичность" кода ПОЛНОСТЬЮ лежит на узких плечах ПРОГРАММИСТА.

Простой пример. Обычно подпрограммы заканчиваются командой возврата. Если ее не задать явно, транслятор все равно добавит ее в конец подпрограммы. Ассемблерная подпрограмма без команды возврата НЕ ВЕРНЕТСЯ в точку вызова, а будет выполнять код, следующий за подпрограммой, как-будто он является ее продолжением. Еще пример. Можно попробовать "выполнить" данные вместо кода. Часто это лишено смысла. Но если программист это сделает, транслятор промолчит. Язык ассемблера позволяет делать все! Тут нет НИКАКИХ ограничений. Но с другой стороны это часто является источником ошибок.

Эти особенности приводят к тому, что ассемблерные программы часто "подвешивают" компьютер, особенно у начинающих программистов.

Выделим три разновидности "зависания" по способу борьбы с ним.

Простое - для выхода из него достаточно нажать Ctrl+Break или Ctrl+C (сначала нажимается клавиша Ctrl и, НЕ ОТПУСКАЯ ее, нажимается вторая клавиша - C или Break; отпускаются в обратном порядке). Программа при этом аварийно завершается выходом в DOS.

Мягкое - машина не реагирует на Ctrl+Break, но клавиатура "дышит".Т. е. при нажатии на клавиши, типа NumLock, моргают соответствующие светодиоды. В этом случае машину нужно будет перегрузить, нажав Ctrl+Alt+Del. В среде Windows нужно просто "убить" сеанс, закрыв окно.

Жесткое - машина никак не реагирует на клавиатуру и не воспринимает комбинацию Ctrl+Alt+Del. В этом случае поможет аппаратный сброс при помощи кнопки "Reset", расположенной на передней панели системного блока. Не нужно ВЫКЛЮЧАТЬ и включать ЭВМ. Вы как будущие разработчики аппаратуры должны знать, что она выходит из строя в основном при включении и выключении.

*1.3 Процесс обработки программы на языке ассемблера*

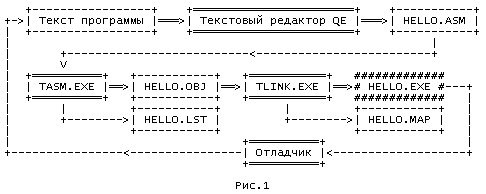
Из-за своей специфики, а также по традиции, для программирования на языке ассемблера нет никаких сред-оболочек типа Turbo C, Turbo Pascal и т.д. Тут приходится пользоваться "утилитами командных строк", как 30 лет назад. Весь процесс технического создания ассемблерной программы можно разбить на 4 шага (исключены этапы создания алгоритма, выбора структур данных и т.д.).

Набор программы в текстовом редакторе и сохранение ее в отдельном файле. Каждый файл имеет имя и тип, называемый иногда расширением. Тип в основном используется для определения назначения файла. Например, программа на C имеет тип C, на Pascal - PAS, на языке ассемблера - ASM.

Обработка текста программы транслятором. На этом этапе текст превращается в машинный код, называемый объектным. Кроме того, есть возможность получить листинг программы, содержащий кроме текста программы различную дополнительную информацию и таблицы, созданные транслятором. Тип объектного файла - OBJ, файла листинга - LST. Этот этап называется ТРАНСЛЯЦИЕЙ.

Обработка полученного объектного кода компоновщиком. Тут программа "привязывается" к конкретным условиям выполнения на ЭВМ. Полученный машинный код называется выполняемым. Кроме того, обычно получается карта загрузки программы в ОЗУ. Выполняемый файл имеет тип EXE, карта загрузки - MAP. Этот этап называется КОМПОНОВКОЙ или ЛИНКОВКОЙ.

Запуск программы. Если программа работает не совсем корректно, перед этим может присутствовать этап ОТЛАДКИ программы при помощи специальной программы - отладчика. При нахождении ошибки приходится проводить коррекцию программы, возвращаясь к шагу 1. Таким образом, процесс создания ассемблерной программы можно изобразить в виде следующей схемы. Конечной целью, напомним, является работоспособный выполняемый файл HELLO. EXE.



*1.4 Особенности создания ассемблерной программы в среде эмулятора EMU8086*

Этот программный продукт содержит все необходимое для создания программы на языке Assembler.

Пакет Emu8086 сочетает в себе продвинутый текстовый редактор, assembler, disassembler, эмулятор программного обеспечения (Виртуальную машину) с пошаговым отладчиком, примеры.

В процессе выполнения программы мы можем наблюдать программные регистры, флаги и память, АЛУ показывает работу центрального процессора.

Встроенная виртуальная машина полностью блокирует обращение программы к реальным аппаратным средствам ЭВМ, накопителям памяти, это делает процесс отладки намного более легкой

*1.5 Правила оформления ассемблерных программ*

При наборе программ на языке ассемблера придерживайтесь следующих правил:

директивы набирайте большими буквами, инструкции - малыми;

пишите текст широко - не скупердяйничайте;

не выходите за край экрана, т.е. не делайте текст шире 80 знаков - его не удобно будет редактировать и печатать;

для отступов пользуйтесь табуляцией (клавиша TAB);

блоки комментариев задавайте с одинаковым отступом. Оптимальной считается такая строка:

<TAB><TAB>mov<TAB>ax,<пробел>bx< (1-3) TAB>; <пробел>текст комментария

Количество табуляций перед комментарием определяется длиной аргументов команды и может быть от 1 до 3. По мере знакомства с синтаксисом языка будут приводиться дополнительные правила.

*2. Задание для выполнения*

2.1 Запустить эмулятор EMU8086.

2.2 Пользуясь правилами оформления ассемблерных программ, исправьте слова "Please Register." на любые понравившиеся (Не забудьте заключить их в апострофы).

2.3 Запустите приложение, нажав кнопку ‘Emulate’ или клавишу F5.

2.4 Запустите полученный код на выполнение, используя кнопку “RUN” или нажмите функциональную клавишу F9.

2.5 Откомпилируйте программу. Вернитесь в главное окно формы, предварительно закрыв все открытые окна, далее нажмите кнопку “Compile”.

2.6 Полученный com-файл запустите во встроенной командной строке WINDOWS 98 на выполнение или запустите сеанс dos в total commander’e.

2.7 Поэкспериментируйте с другими примерами которые открываются при нажатие клавиши “Samples” в главном окне эмулятора.

2.8 Ознакомитесь со встроенной в эмулятор EMU8086 справкой. В ней содержится вся необходимая информация для работы с программой, азы написания программ на языке assembler и др.

*3. Контрольные вопросы*

3.1 Каковы основные отличия ассемблерных программ от ЯВУ?

3.2 Какова структура ассемблерной программы?

3.3 В чем отличие инструкции от директивы?

3.4 Каковы правила оформления программ на языке ассемблера?

3.5 Каковы этапы получения выполняемого файла?

3.6. Для чего нужен этап отладки программы?

3.7. Опишите основные моменты создания исполняемого файла и эмуляции работы программы?

3.8. Каковы шаги технического создания ассемблерной программы в программах TASM и MASM?

3.9 Основные возможности эмулятора EMU8086?

3.10 Методы борьбы с зависанием в DOS’e?

## Лабораторная работа № 2

РАЗРАБОТКА ПЕРВОЙ ПРОГРАММЫ НА ЯЗЫКЕ АССЕМБЛЕРА

Цель работы: Знакомство со структурой ассемблерной программы, создание первой программы на языке ассемблера.

*Структура ассемблерной программы*

Чтобы программа выполнилась любой ОС, она должна быть скомпилирована в исполнимый файл. Основные два формата исполнимых файлов в DOS - СОМ и ЕХЕ.

Файлы типа СОМ содержат только скомпилированный код без какой-либо дополнительной информации о программе Весь код, данные и стек такой полагаются в одном сегменте и не могут превышать 64 Кб.

|  |  |
| --- | --- |
|  | . model tiny |
|  | . code |
|  | org 100h |
|  | begin: |
|  | mov ah, 9 |
|  | mov dx,offset message |
|  | int 21h |
|  | ret |
|  | message db "Привет", 0dh, 0ah, '$' |
|  | end begin |

Рассмотрим исходный текст программы, чтобы понять, как она работает.

Первая строка определяет модель памяти TINY, в которой сегменты кода, данных и стека объединены. Эта модель предназначена для создания файлов типа СОМ.

В DOS для формирования адреса используется сегмент и смещение. Для формирования адреса строки "ПРИВЕТ" используется пара регистров DS (сегмент) и DX (смещение). При загрузке \*.com-программы в память, все сегментные регистры принимают значение равное тому сегменту, в который загрузилась наша программа (в т. ч. и DS). Поэтому нет необходимости загружать в DS сегмент строки (он уже загружен).

Директива. CODE начинает сегмент кода, который в нашем случае также должен содержать и данные.

ORG 100h устанавливает значение программного счетчика (IP) в 100h, потому что при загрузке СОМ-файла в память DOS занимает первые 256 байт (100h) блоком данных PSP и располагает код программы только после этого блока. Все программы, которые компилируются в файлы типа СОМ, должны начинаться с этой директивы.

Метка BEGIN: располагается перед первой командой в программе и будет использоваться в директиве END (Begin - англ. начало; end - конец), чтобы указать, с какой команды начинается программа.

Вообще вместо слова BEGIN можно было бы использовать что-нибудь другое. Например, START:. В таком случае, нам пришлось бы и завершать программу END START.

Строки (5) - (7) выводят на экран сообщение “ПРИВЕТ”.

Рассмотрим вкратце о регистрах процессора.

Регистр процессора - это специально отведенная память для хранения какого-нибудь числа.

Например:

Если мы хотим сложить два числа, то в математике запишем так:

A=5

B=8

C=A+B.

A, B и C - это своего рода регистры (если говорить о компьютере), в которых могут хранится некоторые данные. А=5 можно прочитать как: Присваиваем А число 5.

Для присвоения регистру какого-нибудь значения, в Ассемблере существует оператор mov (от англ. move - загрузить). Команда MOV АН,9 помещает число 9 в регистр АН - номер функции DOS "вывод строки".

Команда MOV DX, OFFSET MESSAGE помещает в регистр DX смещение метки MESSAGE относительно начала сегмента данных, который в нашем случае совпадает с сегментом кода.

OFFSET (по-английски - это смещение). Когда, при ассемблировании, Ассемблер дойдет до этой строки, он заменит OFFSET MESSAGE на АДРЕС (смещение) этой строки в памяти. Если мы запишем OFFSET MESSAGE (хотя, правильнее будет MOV DX, WORD OFFSET MESSAGE), то в DX загрузится не адрес (смещение), а первые два символа нашей строки (в данном случае "Пр"). Так как DX - шестнадцатиразрядный регистр, в него можно загрузить только два байта (один символ всегда один байт).

Команда INT 21H вызывает системную функцию DOS (int от англ. interrupt - прерывание). Можно заменить строку INT 21H на INT 33, программа будет работать корректно. Однако в Ассемблере принято указывать номер прерывания в шестнадцатеричной системе.

Прерывание MS-DOS - это своего рода подпрограмма (часть MS-DOS), которая находится постоянно в памяти и может вызываться в любое время из любой программы.

Эта команда - основное средство взаимодействия программ с операционной системой. В примере вызывается функция DOS номер 9 - вывести строку на экран. Эта функция выводит строку от начала, адрес которого задается в регистрах DS: DX, до первого встречного символа $. При запуске СОМ-файла регистр DS автоматически загружается сегментным адресом программы, а регистр DX был подготовлен предыдущей командой.

Рассмотрим вышесказанное на примере (мелким шрифтом выделим примечания):

Программа сложения двух чисел

Начало программы

A=5 в переменную A заносим значение 5

B=8 в переменную B значение 8

Вызов подпрограммы Сложение

теперь С равно 13

A=10 тоже самое, только другие числа

B=25

Вызов подпрограммы Сложение

теперь С равно 35

Конец Программы

…

Подпрограмма Сложение

C=A+B

Возврат из подпрограммы возвращаемся в то место, откуда вызывали

Конец подпрограммы

В данном примере мы дважды вызвали подпрограмму Сложение, которая сложила два числа, переданные ей в переменных A и B. Результат помещается в переменную С. Когда вызывается подпрограмма, компьютер запоминает с какого места она была вызвана, а затем, когда закончила работу подпрограмма, компьютер возвращается в то место, откуда она вызывалась. Т.о. можно вызывать подпрограммы неопределенное количество раз с любого места.

Команда RET пользуется обычно для возвращения из процедуры. DOS вызывается COM-программы так, что команда RET корректно завершает программу.

DOS при вызове СОМ-файла помещает в стек сегментный адрес программы и ноль, так что RET передает управление на нулевой адрес текущего сегмента, то есть на первый байт PSP. Там находится код команды INT 20H, которая и используется для возвращения управления в DOS. Можно сразу заканчивать программу командой INT 20h, хотя это длиннее на 1 байт.

Следующая строка примера определяет строку данных, содержащую текст “ПРИВЕТ" управляющий символ ASCII возврат каретки с кодом ODh, управляющий символ ASCII перевод строки с кодом 0Ah и символ $ завершающий строку (если мы его уберем, то 21h прерывание продолжит вывод до тех пор, пока не встретится где-нибудь в памяти символ $, на экране мы увидим “мусор). Первое слово (message - сообщение) - название сообщения. Оно может быть любым (например, mess или string и пр).

Управляющие символы (ODh и 0Ah) переводят курсор на первую позицию следующей строки.

Директива END завершает программу, одновременно указывая, с какой метки должно начинаться ее выполнение.

В качестве дополнительного примера создадим еще одну строку, которую назовем message1. Затем, начиная со строки (9) вставим следующие команды и скомпилируем программу заново.

|  |  |
| --- | --- |
| 9 | mov dx,offset message1 |
| 10 | int 21h |
| 11 | int 20h |
| 12 | message db "Привет", 0dh, 0ah, '$' |
| 13 | message1 db "Группа", 0dh, 0ah, '$' |
| 14 | end begin |

*2. Задание для выполнения*

2.1 Запустить эмулятор EMU8086.

2.2 Пользуясь правилами оформления ассемблерных программ, наберите код, приведенный в примере 1, запустите код на выполнение.

2.3 Откомпилируйте пример №2;

2.4 Вернитесь в главное окно формы, предварительно закрыв все открытые окна, далее нажмите кнопку “Compile".

2.5 Полученный com-файл запустите в сеансе dos.

2.6 Создайте на языка Pascal программу выводящую на экран слово “Привет" и сравните размеры получаемых файлов (Pascal и Assembler).

*3. Контрольные вопросы*

3.1 Характеристика структуры файла типа \*.com?

3.2 Какова структура ассемблерной программы?

3.3 С какой целью в код программы на ассемблере для DOS вводится строка ORG 100h?

3.4 Назначение команды MOV?

3.5 Прерывания 21h и 20h. Назначение?

3.6 Сущность и целесообразность использования команды RET вместо прерывания 20h?

3.6. Символ ‘$’ методика применения?

3.7. Связка “BEGIN: - END BEGIN". Правила применения?

ЛАБОРАТОРНАЯ РАБОТА №3

СТРУКТУРА ИСПОЛНИМЫХ ФАЙЛОВ ТИПА \*. EXE.

ПРОСТЫЕ АРИФМЕТИЧЕСКИЕ ДЕЙСТВИЯ НА ЯЗЫКЕ АССЕМБЛЕРА

Цель работы: *Изучение принципов составления простейших\*. exe программ. Изучение приемов работы с простейшими операторами арифметических действий.*

Краткие теоретические сведения (программа типа \*. ЕХЕ).

Файлы типа ЕХЕ содержат заголовок, в котором описывается размер файла, требуемый объем памяти, список команд в программе, использующих абсолютные адреса, которые зависят от расположения программы в памяти, и т.д. ЕХЕ-файл может иметь любой размер. Формат ЕХЕ также используется для исполнимых файлов в различных версиях DOS-расширителей и Windows, но со значительными изменениями.

Операционная система DOS не использует расширения для определения типа файла. Первые два байта заголовка ЕХЕ-файла - символы "MZ" или "ZM", и если файл начинается с этих символов и длиннее некоторого порогового значения, разного для разных версий DOS, он загружается как ЕХЕ, если нет - как СОМ.

ЕХЕ-программы немного сложнее в исполнении, но для них отсутствует ограничение размера в 64 килобайта, так что все достаточно большие программы используют именно этот формат. Конечно, ассемблер позволяет уместить и в 64 килобайтах весьма сложные и большие алгоритмы, а все данные хранить в отдельных файлах, но ограничение размера все равно очень серьезно, и даже чисто ассемблерные программы могут с ним сталкиваться.

*Простой пример* *ЕХЕ-файла*:

|  |  |
| --- | --- |
| . model small | ; сегмент стека размером в 256 байт |
| . stack 100h | ; сегмент стека размером в 256 байт |
| . code | ; сегмент кода, который содержит и данные. |
| Begin: | ; метка начала кода программы |
| mov ax,@data; | ; сегментный адрес строки message помещается в DS |
| mov ds,ax |
| mov dx,offset string | помещает в регистр DX смещение метки String относительно начала сегмента данных |
| mov ah,9 | ; помещаем номер функции DOS "вывод строки (9)" в регистр АН. |
| int 21h | ; функция DOS "вывод строки" |
| mov ax,4C00h | ; завершение программы типа - exe |
| int 21h | ; функция DOS "завершить программу" |
| . data | ; начало сегмента данных |
| string db "Privet", 0Dh,0Ah,'$' | ; cтрока с содержащая выводимые данные. |
| end begin | ; метка окончания кода программы |

В примере определяются три сегмента - сегмент стека директивой. STACK размером в 256 байт, сегмент кода, начинающийся с директивы. CODE, и сегмент данных, начинающийся с. DATA. При запуске ЕХE-программы регистр DS уже не содержит адреса сегмента со строкой string (он указывает на сегмент, содержащий блок данных PSP), а для вызова используемой функции DOS этот регистр должен иметь сегментный адрес строки. Команда MOV AX,@DATA загружает в АХ сегментный адрес группы сегментов данных @DATA, a MOV DS,AX копирует его в DS. Программы типа ЕХЕ должны завершаться системным вызовом DOS 4Ch: в регистр АН помещается значение 4Ch, в регистр AL помещается код возврата (в данном примере код возврата 0 и регистры АН и AL загружаются одной командой MOV AX,4C00h), после чего вызывается прерывание 21h.

*Простые арифметические операторы.*

Арифметические команды любого микропроцессора привлекают к себе наибольшее внимание. Каждый заинтересован в выполнении арифметических вычислений, и именно эти команды проделывают такую работу. Хотя их немного, они выполняют большинство преобразований данных, а микропроцессоре. В реальных же условиях арифметические команды занимают лишь малую часть всех исполняемых команд.

*2.1 Сложение.*

Команда ADD (Addition - сложение (гл. to add - сложить)) осуществляет сложение первого и второго операндов. Исходное значение первого операнда (приемника) теряется, замещаясь результатом сложения. Второй операнд не изменяется. В качестве первого операнда команды ADD можно указывать регистр (кроме сегментного) или ячейку памяти, в качестве второго - регистр (кроме сегментного), ячейку памяти или непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами и представлять числа со знаком или без знака. Команду ADD можно использовать для сложения как обычных целых чисел, так и двоично-десятичных (с использованием регистра АХ для хранения результата). Команда воздействует на флаги OF, SF, ZF, AF, PF и CF.

|  |  |  |
| --- | --- | --- |
| Команда | Назначение | Процессор |
| ADD приемник, источник | Сложение | 8086 |

*Примеры:*

mov al,10 - --> загружаем в регистр AL число 10

add al,15 - --> al = 25; al - приемник, 15 - источник

mov ax,25000 - --> загружаем в регистр AX число 25000

add ax,10000 - --> ax = 35000; ax - приемник, 10000 - источник

mov cx, 200 - --> загружаем в регистр CX число 200

mov bx,760 - --> а в регистр BX - 760

add cx,bx - --> cx = 960, bx = 760 (bx не меняется); cx - приемник, bx - источник

*2.2 Вычитание.*

Команда SUB (Subtraction - вычитание) вычитает второй операнд (источник) из первого (приемника) и помещает результат на место первого операнда. Исходное значение первого операнда (уменьшаемое) теряется. Таким образом, если команду вычитания записать в общем виде

SUB операнд1, операнд2

то ее действие можно условно изобразить следующим образом:

операнд1 - операнд2 - > операнд1

В качестве первого операнда можно указывать регистр (кроме сегментного) или ячейку памяти, в качестве второго - регистр (кроме сегментного), ячейку памяти или непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами и представлять числа со знаком или без знака. Команда воздействует на флаги OF, SF, ZF, AF, PF и CF.

|  |  |  |
| --- | --- | --- |
| Команда | Назначение | Процессор |
| SUB приемник, источник | Вычитание | 8086 |

*Примеры:*

mov al,10

sub al,7 - --> al = 3; al - приемник, 7 - источник

mov ax,25000

sub ax,10000 - --> ax = 15000; ax - приемник, 10000 - источник

mov cx,100

mov bx,15

sub cx,bx - --> cx = 85, bx = 15 (bx не меняется); cx - приемник, bx - источник

*2.3 Инкремент (увеличение на 1).*

Команда INC (Increment - инкремент) прибавляет 1 к операнду, в качестве которого можно указывать регистр (кроме сегментного) или ячейку памяти размером как в байт, так и в слово.

Не допускается использовать в качестве операнда непосредственное значение. Операнд интерпретируется как число без знака. Команда воздействует на флаги OF, SF, ZF, AF и PF. Команда не воздействует на флаг CF; если требуется воздействие на этот флаг, необходимо использовать команду Add Op,l.

Команда INC (Increment - инкремент) увеличивает на единицу регистр или значение операнда в памяти.

Она эквивалентна команде ADD источник, 1 только выполняется гораздо быстрее.

|  |  |  |
| --- | --- | --- |
| Команда | Назначение | Процессор |
| INC приемник | Увеличение на единицу | 8086 |

Примеры:

mov al,15

inc al - --> теперь AL = 16 (эквивалентна add al,1)

mov dh,39h

inc dh - --> DH = 3Ah (эквивалентна add dh,1)

mov cl,4Fh

inc cl - --> CL = 50h (эквивалентна add cl,1)

*2.4 Декремент (уменьшение на 1).*

Команда DEC (Decrement - декремент) вычитает 1 из операнда, в качестве которого можно указывать регистр (кроме сегментного) или ячейку памяти размером как в байт, так и в слово. Не допускается использовать в качестве операнда непосредственное значение. Операнд интерпретируется как число без знака. Команда воздействует на флаги OF, SF, ZF, AF и PF.

Она эквивалентна команде SUB источник, 1 только выполняется гораздо быстрее.

|  |  |  |
| --- | --- | --- |
| Команда | Назначение | Процессор |
| INC приемник | Увеличение на единицу | 8086 |

Примеры:

mov al,15

dec al - --> теперь AL = 14 (эквивалентна sub al,1)

mov dh,39h

dec dh - --> DH = 38h (эквивалентна sub dh,1)

mov cl,4Fh

dec cl - --> CL = 4Dh (эквивалентна sub cl,1)

3. Задание для выполнения.

3.1 Запустить эмулятор EMU8086.

3.2 Получите задание у преподавателя (один из пяти вариантов табл. №1) и, пользуясь правилами оформления ассемблерных программ, напишите программы расчета значения A (два-три варианта).

2.3 Программу ассемблируйте в файл типа \*. exe;

*3. Контрольные вопросы*

3.1 Структура файлов типа \*. exe?

3.2 Структурные отличия файлов \*. exe от \*.com в операционной среде DOS?

3.3 Команда add основное назначение?

3.4 Команда sub основное назначение?

3.5 Команда inc основное назначение?

3.6. Команда dec основное назначение?

Табл. №1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| №  вар. | Расчетная формула | B | C | D |
|  | A=B+C-D | 1 | 35 | 23 |
|  | A=B+C+D | 65 | 1 | 1 |
|  | A=C-D+B | 1 | 33 | 1 |
|  | A=D+A-B | 18 | 1 | 88 |
|  | A= B-C+D | 45 | 10 | 1 |

По согласованию с преподавателем можно изменить как расчетную формулу, так и значения коэффициентов (B, C, D).

## Лабораторная работа № 3

СПОСОБЫ АДРЕСАЦИИ НА ЯЗЫКЕ АССЕМБЛЕРА

Цель работы: *Изучить основные способы адресации*.

Способы адресации.

Способом, или режимом адресации называют процедуру нахождения операнда для выполняемой команды. Если команда использует два операнда, то для каждого из них должен быть задан способ адресации, причем режимы адресации первого и второго операнда могут, как совпадать, так и различаться. Операнды команды могут находиться в разных местах: непосредственно в составе кода команды, в каком-либо регистре, в ячейке памяти; в последнем случае существует несколько возможностей указания его адреса. Строго говоря, способы адресации являются элементом архитектуры процессора, отражая заложенные в нем возможности поиска операндов. С другой стороны, различные способы адресации определенным образом обозначаются в языке ассемблера и являются разделом языка.

В программах, написанных на языке ассемблера термин "операнд" применим к обозначению тех физических объектов, с которыми имеет дело процессор при выполнении машинной команды и, говоря об операндах команд языка, понимают в действительности операнды машинных команд. По отношению к командам ассемблера используется термин "параметры".

В архитектуре современных 32-разрядных процессоров Intel предусмотрены довольно изощренные способы адресации; в МП 86 способов адресации меньше. Мы в настоящей лабораторной работе ознакомимся с режимами адресации, используемые в МП 86.

Различают следующие режимы адресации:

регистровый;

непосредственный;

прямой;

регистровый косвенный (базовый или индексный);

регистровый косвенный со смещением (базовый или индексный);

базово-индексный;

базовый индексный со смещением.

1.1 Регистровый режим

Значение операнда-источника предварительно запоминается в одном из встроенных регистров микропроцессора.

Сам регистр становится эффективным адресом. Операнд (байт или слово) находится в регистре. Этот способ применим ко всем программно-адресуемым регистрам процессора:

|  |  |
| --- | --- |
| inc CX | ; Увеличение на 1 содержимого CX |
| push DS | ; Сегментный адрес сохраняется в стеке |
| xchg BX,BP | ; Регистры BX и BP обмениваются содержимым |
| mov ES,AX | ; Содержимое AX пересылается в ES |

1.2 Непосредственный режим

Непосредственная адресация. Операнд (байт или слово) указывается в команде и после трансляции поступает в код команды; он может иметь любой смысл (число, адрес, код ASCII), а также быть представлен в виде символического обозначения.

|  |  |
| --- | --- |
| mov АН, 40h | ; Число 40h загружается в АН |
| mov AL,'\*' | ; Код ASCII символа "\*' загружается в AL |
| int 21h | ; Команда прерывания с аргументом 21h |
| limit equ 528 | ; Число 528 получает обозначение limit |
| mov CX,limit | ; Число, обозначенное limit, загружается в СХ |

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

Важным применением непосредственной адресации является пересылка относительных адресов (смещений), для этого используется описатель offset (смещение):

*; Сегмент данных*

string db “Privet"; Строка символов

*; Сегмент команд*

mov DX,offset string; Адрес строки засылается в DX

1.3 Прямой режим.

Адресуется память; адрес ячейки памяти (слова или байта) указывается в команде (обычно в символической форме) и поступает в код команды:

*; 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 menu 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. Адрес любой ячейки состоит из двух компонентов: сегментного адреса и смещения. Обозначения meml и mem2 в предыдущем примере, 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?

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 - 361i, для CS - 2Eh. Если префикс отсутствует, сегментный адрес берется из регистра DS (хотя для него тоже предусмотрен свой префикс).

В приведенном примере, по умолчанию, все данные адресуются через сегментный регистр DS, так что вместо inc meml можно было написать inc DS: mem. В случае замены сегментного регистра его обязательно нужно указывать явно:

inc ES: mem1

inc CS: mem2

Обращение к ячейке памяти по известному абсолютному адресу осуществляется следующим образом:

|  |  |
| --- | --- |
| mov AL,DS: [17h] | Загрузка в AL содержимого ячейки со смещением 17h в сегменте, определяемом содержимым DS |

1.4 Регистровый косвенный (базовый и индексный).

Адресуется память (байт или слово). Относительный адрес ячейки памяти находится в регистре, обозначение которого заключается в прямые скобки. 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.

Если косвенная адресация осуществляется через один из регистров ВХ, SI или DI, то подразумевается сегмент, адресуемый через DS, поэтому при адресации через этот регистр обозначение DS: можно опустить:

mov es: [bx],'1' ――――→ mov [bx],'1'

Кстати, этот фрагмент немного эффективнее предыдущего в смысле расходования памяти. Из-за отсутствия в коде последней команды префикса замены сегмента он занимает на 1 байт меньше места.

Регистры ВХ, SI и DI в данном применении совершенно равнозначны, и с одинаковым успехом можно воспользоваться любым из них:

Не так обстоит дело с регистром ВР. Этот регистр специально предназначен для работы со стеком, и при адресации через этот регистр в режимах косвенной адресации подразумевается сегмент стека; другими словами, в качестве сегментного регистра по умолчанию используется регистр SS.

Обычно косвенная адресация к стеку используется в тех случаях, когда необходимо обратиться к данным, содержащимся в стеке, без изъятия их оттуда (например, если к эти данные приходится считывать неоднократно).

Обозначение этого способа адресации:

|  |  |
| --- | --- |
| [BX] | (подразумевается DS: [BX]) |
| [BP] | (подразумевается SS: [BP]) |
| [SI] | (подразумевается DS: [SI]) |
| [DI] | (подразумевается DS: [DI]) |

Использование базовой адресации, на первый взгляд, снижает эффективность программы, так как требует дополнительной операции - загрузки в базовый регистр требуемого адреса. Однако команда с базовой адресацией занимает меньше места в памяти (так как в нее не входит адрес ячейки) и выполняется быстрее команды с прямой адресацией (из-за того, что команда короче, процессору требуется меньше времени на ее считывание из памяти). Поэтому базовая адресация эффективна в тех случаях, когда по заданному адресу приходится обращаться многократно, особенно, в цикле. Выигрыш оказывается тем больше, чем большее число, раз происходит обращение по указанному адресу. С другой стороны, возможности этого режима адресации невелики, и на практике чаще используют более сложные способы.

Примеры:

|  |  |
| --- | --- |
| mov SI, offset string | ; В SI загружается относительный адрес ячейки string |
| mov AX, [SI] | ; Содержимое ячейки string загружается в AX |
| inc [SI] | ; Увеличиваться содержимое ячейки string |
| mov BX, [SI] | ; Новое содержимое ячейки string загружается в BX |
| mov DI, SI | ; Относительный адрес ячейки string копируется в DI |

1.5 Регистровый косвенный режим со смещением (базовый и индексный).

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.

смещение = {SP, BP, DI, SI, BX} + смещение из команды

Иногда можно встретиться с альтернативными обозначениями того же способа адресации, которые допускает ассемблер. Вместо, например, 4 [ВХ] можно с таким же успехом написать [ВХ+4], 4+ [ВХ] или [ВХ] +4. Такая неоднозначность языка ничего, кроме путаницы, не приносит, однако ее надо иметь в виду, так как с этими обозначениями можно столкнуться, например, рассматривая текст деассемблированной программы.

Рассмотрим теперь пример использования базовой адресации со смещением при обращении к стеку:

смещение = {SP, BP, DI, SI, BX} + смещение из команды

Здесь квадратные скобки [] - это тоже оператор. Он вычисляет адрес как сумму того, что находится внутри скобок с тем, что находится снаружи.

array db 0, 10, 20, 30, 40, 50, 60*;* Пусть в сегменте данных определен массив:

*Последовательность команд*:

mov BX,5

mov AL,array [5] ; загрузит в AL элемент массива с индексом 5, то есть 50.

*Тот же результат будет получен и в таких последовательностях команд:*

mov BX,offset array

mov AL,5 [BX]

*или*

mov AL, [BX] +5

mov AL, [BX+5]

1.6 Базово-индексный режим

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

|  |  |
| --- | --- |
| *смещение* [BX] [SI] | (подразумевается DS: *смещение* [BX] [SI]) |
| *смещение* [BX] [DI] | (подразумевается DS: *смещение* [BX] [DI]) |
| *смещение* [BP] [SI] | (подразумевается SS: *смещение* [BP] [SI]) |
| *смещение* [BP] [DI] | (подразумевается SS: *смещение* [BP] [DI]) |

Во всех этих случаях можно также писать:

|  |
| --- |
| *смещение* [BX+SI] |
| [*смещение* +BX+SI] |
| [BX+SI] +*смещение* |

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.

1.7 Базово-индексная адресация со смещением.

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

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.

Приведем формальный пример рассматриваемого режима адресации.

Пусть в сегменте данных определен массив из 24 байт

syms db 'ЙЦУКЕНГШЩЗХЪ'

db 'йцукенгшщзхъ'

*Последовательность команд*

mov BX,12mov SI,6

mov DL,syms [BX] [SI] *;* загрузит в регистр DL элемент с индексом 6 из второго ряда, то есть код ASCII буквы г

*Тот же результат будет получен и в таком варианте:*

mov BX,offset syms

mov SI,6

mov DL,12 [BX] [SI]

2. Порядок выполнения работы:

1. С помощью редактора эмулятора EMU 8086 напишите программу, исходный текст которой приводится в листинге №1:

2. Создайте исполняемый файл типа MZ.

3. Изучите структуру программы, также изучите структуру сегмента данных программы: найдите в нем все переменные, определенные в тексте программы.

4. Переделайте программу с использованием упрощенных директив сегментации так, чтобы получить исполняемый файл типа.com и сравните размеры программ.

5. Выполните первые 5 шагов программы, анализируя и записывая состояние регистров на каждом шаге.

6. Занесите в СХ 00FFh. Определите по способу адресации ячейку памяти в сегменте, где произойдут изменения, записать ее адрес.

7. Выполните дальнейшие шаги программы, анализируя возможные способы адресации.

8. Подготовьте отчет, который должен содержать тексты программ, адреса сегментных регистров и записи адресов ячеек памяти против соответствующих команд, а также запись содержимого этих ячеек.

9. В отчете должны содержаться ответы на следующие вопросы.

3. Контрольные вопросы

Как переслать содержимое X в Y?

Чем отличаются команды

MOV [si], cx

*и*

MOV si, cx?

К какому способу адресации относится команда MOV dx, offset message?

Какие сегменты используются при следующих вариантах адресации: [BX] [SI], [BX] [DI], [BP] [SI], [BP] [DI] ?

5. Что произойдет при выполнении инструкции

MOV AL, DS: 17h?

Чем эта команда отличается от следующей:

MOV AL, DS: [17h] ?

6. Пусть в сегменте данных определен массив

Array db 0,15,22,31,44,45,62,67,76,99

Что окажется в регистре AL после выполнения команд:

MOV BX, 5

MOV AL, array [BX] ?

7. Какой это способ адресации (пример вопроса 6)?

8. Укажите, какие инструкции в программе (листинг №1), созданной в данной лабораторной работе, относятся к инструкциям:

с непосредственным;

косвенным режимом адресации?

9. Укажите способ записи обращения напрямую к ячейке памяти по известному абсолютному адресу?

10. Префиксы, Виды префиксов. Префиксы замены сегмента?

11. Перечислите регистры косвенной и базовой адресации. Опишите отличия?

12. Сущность эффективности базовой адресации в сравнении с прямой?

Листинг №1.

TITLE MOVE2 MOVE2 SEGMENT 'CODE' ASSUME CS: MOVE2, DS: DATA MYPROC PROC OUTPROC:

MOV AX,DATA MOV DS,AX MOV AH,BH MOV AH,X MOV CH,3 MOV AX,3 MOV AX,Y MOV [SI],CX MOV [BP],CX MOV [SI],258 MOV [BP+516],1027 MOV BYTE PTR X,255 MOV BYTE PTR [DI+515],4 MOV WORD PTR [DI+515],4 MOV [DI+BP+515],258 MOV AX, [SI+BX+258] MOV AH,4CH INT 21H MYPROC ENDP MOVE2 ENDS DATA SEGMENT X DB 1 Y DW 2 DATA ENDS END MYPROC

ЛАБОРАТОРНАЯ РАБОТА № 5,6

ВЫВОД НА ЭКРАН В ТЕКСТОВОМ РЕЖИМЕ

Цель работы: *Ознакомится с основными средствами вывода текстовых данных на экран посредством средств операционной системы DOS, средствами BIOS и средствами непосредственного (прямого) отображением в видеобуфер.*

Средства DOS.

1.1 Функция DOS 02h.

Функция DOS 02h - Записать символ в STDOUT с проверкой на Ctrl-Break

|  |  |
| --- | --- |
| Ввод: | АН = 02h  DL = ASCII-код символа |
| Вывод: | Никакого, согласно документации, но на самом деле: AL = код последнего записанного символа (равен DL, кроме случая, когда DL = 09h (табуляция), тогда в AL возвращается 20h). |

Эта функция при выводе на экран обрабатывает некоторые управляющие символы - вывод символа BEL (07h) приводит к звуковому сигналу, символ BS (08h) приводит к движению курсора влево на одну позицию, символ НТ (09h) заменяется на несколько пробелов, символ LF (0Ah) опускает курсор на одну позицию вниз, и CR (0Dh) приводит к переходу на начало текущей строки.

Если в ходе работы этой функции была нажата комбинация клавиш Ctrl-Break, вызывается прерывание 23h, которое по умолчанию осуществляет выход из программы.

*Простой пример работы функции DOS 02h.*

Пример № 1.1

|  |  |
| --- | --- |
| . model tiny | ; модель памяти в которой сегменты кода, данных и стека объединены. |
| . code | ; сегмент кода, который содержит данные. |
| org 100h | ; начало СОМ-файла |
| begin: | ; метка начала кода программы |
| mov dl,< ASCII-код символа > | ; заносим в регистр dl - любой ASCII-код символа |
| mov ah,2 | ; номер функции DOS "вывод символа" |
| int 21h | ; вызов DOS |
| ret | ; функция DOS "завершить программу" |
| end begin | ; метка окончания кода программы |

Эта программа, выводит на экран любой ASCII-символ, в установленную позицию курсора.

Все функции DOS вывода на экран используют устройство STDOUT, стандартный вывод. Это позволяет перенаправлять вывод программы в файл или на стандартный ввод другой программы. Например, если откомпилировать приведен пример (создать файл cod.com) и написать в командной строке

cod.com > cod. out

то на экран ничего выдано не будет, а в текущем каталоге появится файл cod. out, содержащий ASCII-код символа.

Функция DOS 06h.

Функция DOS 06h - Записать символ в STDOUT без проверки на Ctrl-Break

|  |  |
| --- | --- |
| Ввод: | АН = 06h  DL = ASCII-код символа (кроме FFh) |
| Вывод: | Никакого, согласно документации, но на самом деле: AL = код записанного символа (копия DL) |

Эта функция не обрабатывает управляющие символы (CR, LF, HT и BS выполняют свои функции при выводе на экран, но сохраняются при перенаправлении вывода в файл) и не проверяет нажатие Ctrl-Break.

Заменим в примере № 1.1 MOV АН,2 на MOV АН,6 и перекомпилируем этот пример. Работу откомпилированного примера смотрим в операционной системе MS-DOS.

Функция DOS 09h

Функция DOS 09h - Записать строку в STDOUT с проверкой на Ctrl-Break

|  |  |
| --- | --- |
| Ввод: | АН = 09h DS: DX = адрес строки, заканчивающейся символом $ (24h) |
| Вывод: | Никакого, согласно документации, но на самом деле: AL = 24h (код последнего символа) |

Действие этой функции полностью аналогично действию функции 02h, но выводится не один символ, а целая строка (смотри лабораторную работу №2).

1.4 Функция DOS 40h

Функция DOS 40h - Записать в файл или устройство

|  |  |
| --- | --- |
| Ввод: | АН = 40h ВХ = 1 для STDOUT или 2 для STDERR DS: DX = адрес начала строки СХ = длина строки |
| Вывод: | CF = 0, АХ = число записанных байт |

Эта функция предназначена для записи в файл, но, если в регистр ВХ поместить число 1, функция 40h будет выводить данные на STDOUT, а если ВХ = 2 - на устройство STDERR. STDERR всегда выводит данные на экран и не перенаправляется в файлы. На этой функции основаны используемые в С функции стандартного вывода - фактически функция С fputs () просто вызывает это прерывание, помещая свой первый аргумент в ВХ, адрес строки (второй аргумент) - в DS: DX и длину - в СХ.

*Простой пример работы функции DOS 40h.*

Пример № 1.2

|  |  |
| --- | --- |
| . model tiny | ; модель памяти в которой сегменты кода, данных и стека объединены. |
| . code | ; сегмент кода, который содержит данные. |
| org 100h | ; начало СОМ-файла |
| begin: | ; метка начала кода программы |
| mov ah,40h | ; номер функции DOS |
| mov bx,2 | ; указываем устройство STDERR |
| mov dx,offset message | ; DS: DX - адрес строки |
| mov cx,25 | ; CX - длина строки |
| int 21h | ; вызов DOS |
| ret | ; функция DOS "завершить программу" |
| message db "This function can print $" | ; cтрока с содержащая выводимые данные. |
| end begin | ; метка окончания кода программы |

Если скомпилировать этот пример и запустить ее командой

dosout.com > dosout. out

то сообщение появится на экране, а файл dosout2. out окажется пустым.

Прерывание INT 29H

INT 29h: Быстрый вывод символа на экран

|  |  |
| --- | --- |
| Ввод: | AL = ASCII-код символа |

*Простой пример работы прерывания INT 29h.*

Пример № 1.3

|  |  |
| --- | --- |
| . model tiny | ; модель памяти в которой сегменты кода, данных и стека объединены. |
| . code | ; сегмент кода, который содержит данные. |
| org 100h | ; начало СОМ-файла |
| begin: | ; метка начала кода программы |
| mov ax, < ASCII-код символа > | ; заносим в регистр ax - любой ASCII-код символа |
| int 29h | ; вызов прерывания DOS - вызов символа; |
| ret | ; функция DOS "завершить программу" |
| end begin | ; метка окончания кода программы |

В большинстве случаев INT 29h просто немедленно вызывает функцию BIOS "вывод символа на экран в режиме телетайпа", так что никаких преимуществ, кроме экономии байт при написании как можно более коротких программ, она не имеет.

2. Средства BIOS

Функции DOS вывода на экран позволяют перенаправить вывод в файл, но не позволяют вывести текст в любую позицию экрана и не позволяют изменить цвет текста. DOS предполагает, что для более тонкой работы с экраном программы должны использоваться видеофункции BIOS. BIOS - обеспечивает доступ к некоторым устройствам, в частности к видеоадаптеру. Все функции видеосервиса BIOS вызываются через прерывание 10h.

*2.1 Выбор видеорежима*

BIOS предоставляет возможность переключения экрана в различные текстовые и графические режимы. Режимы отличаются друг от друга разрешением (для графических) и количеством строк и столбцов (для текстовых), а также количеством возможных цветов.

*2.1.1 Стандартные видеорежимы*

INT 10h, АН = 00 - Установить видеорежим

|  |  |
| --- | --- |
| Ввод: | AL = номер режима в младших 7 битах |
| Вывод: | Обычно никакого, но некоторые BIOS (Phoenix и AMI) помещают в AL 30Н для текстовых режимов и 20h для графических |

*Пример работы.*

Пример № 2.1

|  |  |
| --- | --- |
| . model tiny | ; модель памяти в которой сегменты кода, данных и стека объединены. |
| . code | ; сегмент кода, который содержит данные. |
| org 100h | ; начало СОМ-файла |
| begin: | ; метка начала кода программы |
| mov ah,00 | ; устанавливаем видеорежим |
| mov al,5 | ; устанавливаем номер режима |
| int 10h | ; вызов прерывания DOS - вызов видеосервиса; |
| ret | ; функция DOS "завершить программу" |
| end begin | ; метка окончания кода программы |

Вызов этой функции приводит к тому, что экран переводится в выбранный режим. Если старший бит AL не установлен в 1, экран очищается. Номера текстовых режимов - 0, 1, 2, 3 и 7.0 и 1 - 16-цветные режимы 40x25 (с 25 строками по 40 символов в строке), 2 и 3 - 16-цветные режимы 80x25, 7 - монохромный режим 80x25. Существует еще много текстовых режимов с более высоким разрешением (80x43, 80x60, 132x50 и т.д.), но их номера для вызова через эту функцию различны для разных видеоадаптеров (например, режим 61h - 132x50 для Cirrus 5320 и 132x29 для Genoa 6400). Однако, если видеоадаптер поддерживает стандарт VESA BIOS Extention, в режимы с высоким разрешением можно переключаться, используя функцию 4Fh.

*2.1.2 SuperVGA-видеорежим*

INT 10h, АН = 4Fh, AL = 02 - Установить SuperVGA-видеорежим

|  |  |
| --- | --- |
| Ввод: | ВХ = номер режима в младших 13 битах |
| Вывод: | AL = 4Fh, если эта функция поддерживается АН = 0, если переключение произошло успешно АН = 1, если произошла ошибка |

Если бит 15 регистра ВХ установлен в 1, видеопамять не очищается. Текстовые режимы, которые можно вызвать с использованием этой функции: 80x60 (режим 108h), 132x25 (109h), 132x43 (10Ah), 132x50 (10Bh), 132x60 (10Ch).

Видеорежим, используемый в DOS по умолчанию, - текстовый режим 3.

2.2 Управление положением курсора

*2.2.1 Устанавливаем положение курсора*

INT 10h, АН = 02 - Установить положение курсора

|  |  |
| --- | --- |
| Ввод: | АН = 02 ВН = номер страницы DH = строка DL = столбец |

*Пример работы.*

Пример № 2.2.1

|  |  |
| --- | --- |
| . model tiny | ; модель памяти, в которой сегменты кода, данных и стека объединены. |
| . code | ; сегмент кода, который содержит данные. |
| org 100h | ; начало СОМ-файла |
| begin: | ; метка начала кода программы |
| mov ah,02 | ; устанавливаем положение курсора |
| mov bh,0 | ; устанавливаем номер страницы |
| mov dh,12 | ; строка 12 |
| mov dl,29 | ; столбец 29 |
| int 10h | ; прерывания DOS - установить положение курсора в точку 12,29 |
| mov ax, < ASCII-код символа > | ; заносим в регистр ax - любой ASCII-код символа |
| int 29h | ; вызов прерывания DOS - вызов символа; |
| ret | ; функция DOS "завершить программу" |
| end begin | ; метка окончания кода программы |

С помощью этой функции можно установить курсор в любую позицию экрана, и дальнейший вывод текста будет происходить из этой позиции. Отсчет номера строки и столбца ведется от верхнего левого угла экрана (символ в левой верхней позиции имеет координаты 0,0). Номера страниц 0 - 3 (для режимов 2 и 3) и 0 - 7 (для режимов 1 и 2) соответствуют области памяти, содержимое которой в данный момент отображается на экране. Можно вывести текст в неактивную в настоящий момент страницу, а затем переключиться на нее, чтобы изображение изменилось мгновенно.

*2.2.2 Считываем положение и размер курсора*

INТ 10h, АН = 03 - Считать положение и размер курсора

|  |  |
| --- | --- |
| Ввод: | АН = 03 ВН = номер страницы |
| Вывод: | DH, DL = строка и столбец текущей позиции курсора СН, CL = первая и последняя строки курсора |

Возвращает текущее состояние курсора на выбранной странице (каждая страница использует собственный независимый курсор).

2.3 Вывод символов на экран

Каждый символ на экране описывается двумя байтами - ASCII-кодом символа и байтом атрибута, указывающим цвет символа и фона, а также является ли символ мигающим.

Атрибут символа:

Бит 7: символ мигает (по умолчанию) или фон яркого цвета (если его действие было переопределено видеофункцией 10h).

Биты 6 - 4: цвет фона. Бит 3: символ яркого цвета (по умолчанию) или фон мигает (если его действие было переопределено видеофункцией 11h).

Биты 2 - 0: цвет символа.

Цвета кодируются в битах, как показано в таблице №2.3 .

Таблица №2.3 Атрибуты символов

|  |  |  |
| --- | --- | --- |
|  | Обычный цвет | Яркий цвет |
| 000b | черный | темно-серый |
| 001b | синий | светло-синий |
| 010b | зеленый | светло-зеленый |
| 011b | голубой | светло-голубой |
| 100b | красный | светло-красный |
| 101b | пурпурный | светло-пурпурный |
| 110b | коричневый | желтый |
| 111b | светло-серый | белый |

*2.3.1 Считываем символ и атрибут символа в текущей позиции курсора.*

INT 10h, АН = 08 - Считать символ и атрибут символа в текущей позиции курсора

|  |  |
| --- | --- |
| Ввод: | АН = 08 ВН = номер страницы |
| Вывод: | АН = атрибут символа AL = ASCII-код символа |

*2.3.1 Выводим символ с заданным атрибутом на экран*

INT 10h, АН = 09 - Вывести символ с заданным атрибутом на экран

|  |  |
| --- | --- |
| Ввод: | АН = 09 ВН = номер страницы AL = ASCII-код символа BL = атрибут символа СХ = число повторений символа |

С помощью этой функции можно вывести на экран любой символ, включая даже символы CR и LF, которые обычно интерпретируются как конец строки. В графических режимах СХ не должен превышать число позиций, оставшееся до правого края экрана.

*Пример работы.*

Пример № 2.2.1

|  |  |
| --- | --- |
| . model tiny | ; модель памяти, в которой сегменты кода, данных и стека объединены. |
| . code | ; сегмент кода, который содержит данные. |
| org 100h | ; начало СОМ-файла |
| begin: | ; метка начала кода программы |
| mov ah,09 | ; помещаем номер функции DOS "вывод строки (9)" в регистр АН. |
| mov bh,0 | ; устанавливаем номер страницы |
| mov al, < ASCII-код символа > | ; строка 12; заносим в регистр al - любой ASCII-код символа |
| mov bl, 00011111b | ; атрибут символа (белый на голубом) |
| mov cx,555 | ; устанавливаем в счетчик кол-во выводимых символов |
| int 10h | ; вызов прерывания DOS - вызов символа; |
| ret | ; функция DOS "завершить программу" |
| end begin | ; метка окончания кода программы |

*2.3.2 Выводим символ с текущим атрибутом на экран*

INT 10h, АН = 0Ah - Вывести

|  |  |
| --- | --- |
| Ввод: | АН = 0Ah ВН = номер страницы AL = ASCII-код символа СХ = число повторений символа |

Эта функция также выводит любой символ на экран, но в качестве атрибута символа используется атрибут, который имел символ, находившийся ранее в этой позиции.

*2.3.3 Выводим символ в режиме телетайпа*

INT 10h, АН = 0Eh - Вывести символ в режиме телетайпа

|  |  |
| --- | --- |
| Ввод: | АН = 0Eh ВН = номер страницы AL = ASCII-код символа |

Символы CR (0Dh), LF (0Ah), BEL (7) интерпретируются как управляющие символы. Если текст при записи выходит за пределы нижней строки, экран прокручивается вверх. В качестве атрибута используется атрибут символа, находившегося в этой позиции.

*2.3.4 Выводим строку символов с заданными атрибутами*

INT 10h, AH = 13h - Вывести строку символов с заданными атрибутами

|  |  |
| --- | --- |
| Ввод: | АН = 13h AL = режим вывода:  Бит 0 - переместить курсор в конец строки после вывода бит 1 - строка содержит не только символы, но также и атрибуты, так что каждый символ описывается двумя байтами: ASCII-код и атрибут биты 2 - 7 зарезервированы  СХ = длина строки (только число символов) BL = атрибут, если строка содержит только символы DH,DL = строка и столбец, начиная с которых будет выводиться строки ES: BP = адрес начала строки в памяти |

Функция 13h выводит на экран строку символов, интерпретируя управляющие символы CR (0Dh), LF (0Ah), BS (08) и BEL (07). Если строка подготовлена в формате символ, атрибут - гораздо быстрее просто скопировать ее в видеопамять.

Функции BIOS удобны для переключения и настройки видеорежимов, но часто оказывается, что вывод текста на экран гораздо быстрее и проще выполнять просто копированием изображения в видеопамять.

3. Прямая работа с видеопамятью

Все, что изображено на мониторе - и графика, и текст, одновременно присутствует в памяти, встроенной в видеоадаптер. Для того чтобы изображение появилось на мониторе, оно должно быть записано в память видеоадаптера. Для этого отводится специальная область памяти, начинающаяся с абсолютного адреса 0B800h: 0000h (для текстовых режимов) и заканчивающаяся на 0B800h: FFFFh. Все, что программы пишут в эту область памяти, немедленно пересылается в память видеоадаптера. В текстовых режимах для хранения каждого изображенного символа используются два байта: байт с ASCII-кодом символа и байт с его атрибутом, так что по адресу 0B800h: 0000h лежит байт с кодом символа, находящимся в верхнем левом углу экрана; по адресу 0B800h: 0001h лежит атрибут этого символа; по адресу 0B800h: 0002h лежит код второго символа в верхней строке экрана и т.д.

Таким образом, любая программа может вывести текст на экран, простой командой пересылки данных, не прибегая ни к каким специальным функциям DOS или BIOS.

*Пример работ с видеопамятью.*

Пример № 3.1

|  |  |
| --- | --- |
| . model tiny | ; модель памяти, в которой сегменты кода, данных и стека объединены. |
| . code | ; сегмент кода, который содержит данные. |
| org 100h | ; начало СОМ-файла |
| begin: | ; метка начала кода программы |
| mov ax,0003h | ; видеорежим 3 (очистка экрана) |
| int 10h | ; прерывание DOS - очистка экрана; |
| mov ax,0B800h | ; загружаем в сегментный регистр ES число 0B800h |
| mov es,ax |
| mov di,0 | ; загружаем в регистр DI нуль |
| mov ah,31 | ; заносим в регистр ah - атрибут символа |
| mov al, < ASCII-код символа > | заносим в регистр al - любой ASCII-код символа |
| mov es: [di],ax | ; заносим по адресу 0B800: 0000h атрибут и ASCII-код символа |
| mov ah,10h | ; вызываем функцию 10h - чтобы можно было остановить программу до нажатия любой клавиши |
| int 16h | ; вызываем прерывание 16h - сервис работы с клавиатурой BIOS |
| ret | ; функция DOS "завершить программу" |
| end begin | ; метка окончания кода программы |

При подготовке данных для копирования в видеопамять в этой программе в строках (7) и (8) загружаем в сегментный регистр ES число 0B800h, которое соответствует сегменту дисплея в текстовом режиме. В строке (9) загружаем в регистр DI нуль. Это будет смещение относительно сегмента 0B800h. В строках (10) и (11) в регистр AH заносится атрибут символа (31 - ярко-белый символ на синем фоне) и в AL - ASCII-код символа (01 - рожица) соответственно.

В строке (12) заносим по адресу 0B800: 0000h (т.е. первый символ в первой строке дисплея - верхний левый угол) атрибут и ASCII-код символа (31 и 01 соответственно).

4. Задание для выполнения.

4.1 С помощью редактора эмулятора EMU 8086 напишите программы примеры, которых приведены в данной лабораторной работе.

4.2 Создайте файлы типа MZ и \*.com.

4.3 Изучите структуру откомпилированных программ.

4.4 Получите задание у преподавателя (один из пяти вариантов табл. №1) напишите программу вывода на экран строки ‘Hello’.

4.5 Напишите программу работы переключения SuperVGA-видеорежимов (согласно вариантов табл. №2)

4.6 Подготовьте отчет, который должен содержать тексты программ, а также укажите описание работы команд программ.

4.7 В отчете должны содержаться ответы на следующие вопросы.

5. Контрольные вопросы

5.1 Перечислите функции вывода на экран средствами операционной системы DOS?

5.2 Принцип работы функции DOS 02h?

5.3 Укажите основные управляющие символы вывода на экран?

5.4 Каким образом осуществить вывод программы в файл?

5.5 Укажите отличие функции DOS 02h от 06h?

5.6 Прерывание int 29h. Преимущества использования?

5.7 С помощью каких функций можно установить нужный видеорежим (текстовый, цветной, монохромный)?

5.8 Отметьте основные моменты установки super VGA-видеорежимов?

5.9 Укажите функции и прерывания управления положением курсора.

5.10 Перечислите функции считывания положения и размера курсора?

5.11 Вывод символов на экран средствами BIOS. Функции?

5.12 Прямая работа с видеопамятью. Принципы работы с видеопамятью?

5.13 Укажите преимущества вывода на экран с помощью непосредственной работы с видеопамятью?

5.14 Область памяти видеоадаптера?

5.15 Укажите код третьего символа в верхней строке экрана для работы с видеопамятью?

5.16 Если в примере № 1.2 длину строку указать большую, чем указанная что в данном случае будет выводиться на экрана

Табл. №1

|  |  |  |  |
| --- | --- | --- | --- |
| №  вар. | Функция вывода (DOS) | Функция вывода (BIOS) | Видеопамять |
|  | 02h | Ah=02h | ‘Hello’ |
|  | 06h | Ah=08h |
|  | 09h | Ah=09h |
|  | 40h | Ah=0Ah |
|  | 29h | Ah=13h |

Примечание: В примерах, в которых возможно задание различных параметров вывода (цвет символа, фона; номер строки, столбца, страницы и т.д.) выводите на экран слово "hello" с параметрами отличными от стандартных.

Табл. №2

|  |  |
| --- | --- |
| №  вар. | *SuperVGA-видеорежим* |
|  | 108h |
|  | 109h |
|  | 10Ah |
|  | 10Bh |
|  | 10Ch |

## Лабораторная работа № 4

КОМАНДЫ ЛОГИЧЕСКИХ ОПЕРАЦИЙ

Цель работы: *ознакомиться с работой команд логических операций: and, or, xor, test, not и реализацией их работы на практике*.

Краткие теоретические сведения.

Логические операции являются важным элементом в проектировании микросхем и имеют много общего в логике программирования. Команды AND, OR, XOR и TEST - являются командами логических операций. Эти команды используются для сброса и установки бит и для арифметических операций в коде ASCII. Все эти команды обрабатывают один байт или одно слово в регистре или в памяти, и устанавливают флаги CF, OF, PF, SF, ZF.

Команда AND.

Команда AND (Логическое И) осуществляет логическое (побитовое) умножение первого операнда на второй. Исходное значение первого операнда (приемника) теряется, замещаясь результатом умножения. В качестве первого операнда команды and можно указывать регистр (кроме сегментного) или ячейку памяти, в качестве второго - регистр (кроме сегментного), ячейку памяти или непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами. Команда воздействует на флаги SF, ZF и PF.

Правила побитового умножения:

|  |  |
| --- | --- |
| Первый операнд-бит 0101 | Бит результата 0001 |
| Второй операнд-бит 0011 |

*Пример 1*

mov AX,0FFEh

and AX,5555h; AX=0554h

*Пример 2*

mov ax,00101001b

add ax,11110111b ; ax=00100001b

Команда OR

Команда OR (Логическое ВКЛЮЧАЮЩЕЕ ИЛИ) выполняет операцию логического (побитового) сложения двух операндов. Результат замещает первый операнд (приемник); второй операнд (источник) не изменяется. В качестве первого операнда можно указывать регистр (кроме сегментного) или ячейку памяти, в качестве второго - регистр (кроме сегментного), ячейку памяти или непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды команды OR могут быть байтами или словами. Команда воздействует на флаги OF, SF, ZF, PF и CF, при этом флаги CF и OF всегда сбрасываются в 0.

Правила побитового сложения:

|  |  |
| --- | --- |
| Первый операнд-бит 0101 | Бит результата 0111 |
| Второй операнд-бит 0011 |

*Пример 1*

mov AX,000Fh

mov BX,00F0h

or AX,BX; AX=00FFh, BX=00F0h

*Пример 2*

mov AX,00101001b

mov BX,11110111b

or AX,BX ; mov dx,11111111b

*Пример 3*

mov AX,000Fh

or AX,8001h ; AX=800Fh

Команда XOR.

Команда ХОR (Логическое ИСКЛЮЧАЮЩЕЕ ИЛИ) выполняет операцию логического (побитового) ИСКЛЮЧАЮЩЕГО ИЛИ над своими двумя операндами. Результат операции замещает первый операнд; второй операнд не изменяется. Каждый бит результата устанавливается в 1, если соответствующие биты операндов различны, и сбрасывается в 0, если соответствующие биты операндов совпадают.

В качестве первого операнда команды ХОR можно указывать регистр (кроме сегментного) или ячейку памяти, в качестве второго - регистр (кроме сегментного), ячейку памяти или непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами. Команда воздействует на флаги OF, SF, ZF, PF и CF, причем флаги OF и CF всегда сбрасываются, а остальные флаги устанавливаются в зависимости от результата.

Правила побитового исключающего или:

|  |  |
| --- | --- |
| Первый операнд-бит 0101 | Бит результата 0110 |
| Второй операнд-бит 0011 |

*Пример 1*

mov AX,0Fh

хог AX,0FFFFh; AX=FFF0h

*Пример 2*

mov AX,00101001b

mov BX,11110111b

xor ax,bx; 11011110b

*Пример 3*

mov SI,0AAAAh

mov BX,5555h

xor SI,BX ; SI=FFFFh,BX=5555h

*Пример 4*

хог ВХ, ВХ ; Обнуление ВХ

Команда TEST.

Команда TEST (Логическое сравнение) выполняет операцию логического умножения И над двумя операндами и, в зависимости от результата, устанавливает флаги SF, ZF и PF. Флаги OF и CF сбрасываются, a AF имеет неопределенное значение. Состояние флагов можно затем проанализировать командами условных переходов. *Команда TEST не изменяет ни один из операндов.*

В качестве первого операнда команды TEST можно указывать регистр (кроме сегментного) или ячейку памяти, в качестве второго - регистр (кроме сегментного), ячейку памяти или непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами и представлять числа со знаком или без знака.

Правила побитового умножения:

|  |  |
| --- | --- |
| Первый операнд-бит 0101 | Бит результата 0001 |
| Второй операнд-бит 0011 |

Флаг SF устанавливается в 1, если в результате выполнения команды образовалось число с установленным знаковым битом.

Флаг ZF устанавливается в 1, если в результате выполнения команды образовалось число, состоящее из одних двоичных нулей.

Флаг PF устанавливается в 1, если в результате выполнения команды образовалось число с четным количеством двоичных единиц в его битах.

*Пример 1*

test AX,1

jne label2: ; Переход, если бит 0 в АХ установлен

je label1: ; Переход, если бит 0 в АХ сброшен

Пример №1.4.1.1

|  |  |
| --- | --- |
| . model tiny | ; модель памяти в которой сегменты кода, данных и стека объединены. |
| . code | ; сегмент кода, который содержит данные. |
| org 100h | ; начало СОМ-файла |
| begin: | ; метка начала кода программы |
| mov СX,<число1 > | ; загружаем в CX число1 <любое число1> |
| mov BX,<число2> | ; загружаем в BX число2 <любое число2> |
| test cx,bx | ; логически сравниваем числа в регистрах cx с bx |
| jne label2 | ; если одно из значений не равно 0 то переходим на метку label2 |
| je label1 | ; если одно из значений равно 0 то переходим на метку label1 |
| ret | ; функция DOS "завершить программу" (не выполняется) |
| label1: | ; начало блока метки Label1 |
| mov ah,9 | ; помещаем номер функции DOS "вывод строки (9)" в регистр АН. |
| mov dx,offset string | помещает в регистр DX смещение метки String относительно начала сегмента данных |
| int 21h | ; функция DOS "вывод строки" |
| ret | ; функция DOS "завершить программу" |
| String db 'одно из чисел равно 0$' | ; cтрока с содержащая выводимые данные. |
| label2: | ; начало блока метки Label2 |
| mov ah,9 | ; помещаем номер функции DOS "вывод строки (9)" в регистр АН. |
| mov dx,offset string1 | помещает в регистр DX смещение метки String1 относительно начала сегмента данных |
| int 21h | ; функция DOS "вывод строки" |
| ret | ; функция DOS "завершить программу" |
| string1 db 'не равны 0$' | ; cтрока с содержащая выводимые данные. |
| end begin | ; метка окончания кода программы |

Данный пример сравнивает два значения (строка (6)), если одно из двух значений равно нулю тогда переходим на метку label1 (строка (10)) далее выполняются команды, следующие после этой метки, в случае если одно из двух значений равно нулю тогда переходим на метку label2

*Пример 2*

test SI,8

jne bityes ; Переход, если бит 3 в SI установлен

je bitno ; Переход, если бит 0 в АХ сброшен

*Пример 3*

test DX,0FFFFh

jz null ; Переход, если DX=0

jnz smth; Переход, если DX не 0

Команда NOT.

Команда NOT (NOT Инверсия, дополнение до 1, логическое отрицание) выполняет инверсию битов указанного операнда, заменяя 0 на 1 и наоборот. В качестве операнда можно указывать регистр (кроме сегментного) или ячейку памяти размером как в байт, так и в слово. Не допускается использовать в качестве операнда непосредственное значение. Команда не воздействует на флаги процессора.

Правила побитовой инверсии:

|  |  |
| --- | --- |
| Операнд-бит 0 1 | Бит результата 1 0 |

*Пример 1*

mov AX,0FFFFh

not AX; AX=0000h

*Пример 2*

mov SI,5551h

not SI; SI=AAAEh

Характерные примеры работы команд логических операций.

Для следующих несвязанных примеров, предположим, что:

AL содержит 1100 0101

BH содержит 0101 1100:

1. AND AL,BH; Устанавливает в AL 0100 0100

2. OR BH,AL; Устанавливает в BH 1101 1101

3. XOR AL,AL; Устанавливает в AL 0000 0000

4. AND AL,00; Устанавливает в AL 0000 0000

5. AND AL,0FH; Устанавливает в AL 0000 0101

6. OR CL,CL; Устанавливает флаги SF и ZF

Примеры 3 и 4 демонстрируют способ очистки регистра.

В примере 5 обнуляются левые четыре бита регистра AL.

Можно применить команду OR для следующих целей:

1. OR CX,CX; Проверка CX на нуль

JZ; Переход, если нуль

2. OR CX,CX; Проверка знака в CX

JS; Переход, если отрицательно

3. Задание для выполнения.

Запустить эмулятор EMU8086.

Пользуясь правилами оформления ассемблерных программ, наберите код примера №1.4.1.1, запустите код на выполнение.

Проанализируйте работу кода примера №1.4.1.1

С помощью справки эмулятора EMU8086 выясните работу команд (jne, je, js, jz)

Получите задание у преподавателя (один из четырех вариантов (команды (and, or, xor, test, not))) и, пользуясь правилами оформления ассемблерных программ, напишите три программы характеризующие (показывающие) работу их с числами (двоичной, десятеричной, шестнадцатеричной систем счисления) согласно перечислению приведенных примеров.

Результаты работы продемонстрируйте преподавателю.

4. Контрольные вопросы

4.1 Назначение команд логических операций?

4.2 Команда and основное назначение?

4.3 Команда or основное назначение?

4.4 Команда xor основное назначение?

4.5 Команда test основное назначение?

4.6 Команда not основное назначение?

4.7 Альтернативная работа команд (test, xor, and)?

4.8 Допустим что будит

4.9 В чем заключается работа связки команд jne, je?

4.10 В чем заключается работа связки команд js, jz?

4.11 Что произойдет, если в примере №1.4.1.1 мы изменим строку (8) на следующую (jne label1)?

ЛАБОРАТОРНАЯ РАБОТА №8

ОЗНАКОМЛЕНИЕ С РАБОТОЙ ЦИКЛОВ

Цель работы: *ознакомиться со структурой и реализацией циклов в программе.*

Краткие теоретические сведения.

Циклы, позволяющие выполнить некоторый участок программы многократно, в любом языке являются одной из наиболее употребительных конструкций. В системе команд МП 86 циклы реализуются, главным образом, с помощью команды loop (петля), хотя имеются и другие способы организации циклов. В большинстве случаях число шагов в цикле определяется содержимым регистра СХ, поэтому максимальное число шагов составляет 64 К.

Организация циклических переходов, как на языках высокого уровня, так и на языке assembler представляет собой замечательное средство, позволяющее значительно снизить код исполняемой программы.

В общем виде любой цикл записывается в ассемблере как условный переход.

Организация цикла с помощью команды LOOP (Первый способ).

Команда loop (анг. петля) выполняет декремент содержимого регистра СХ (счетчик), и если оно не равно 0, осуществляет переход на указанную метку вперед или назад в том же программном сегменте в диапазоне - 128... + 127 байт. Обычно метка помещается перед первым предложением тела цикла, а команда loop является последней командой цикла. Содержимое регистра СХ рассматривается как целое число без знака, поэтому максимальное число повторений группы включенных в цикл команд составляет 65536 (если перед входом в цикл СХ=0). Команда не воздействует на флаги процессора.

|  |  |  |
| --- | --- | --- |
| Команда | Назначение | Процессор |
| LOOP метка | Организация циклов | 8086 |

*Простейший пример организации циклического перехода (со счетчиком в регистре cx) на языке Assembler:*

|  |  |
| --- | --- |
| . model tiny | ; модель памяти, в которой сегменты кода, данных и стека объединены. |
| . code | ; сегмент кода, который содержит и данные. |
| org 100h | ; начало СОМ-файла |
| begin: | ; метка начала кода программы |
| mov cx,10 | ; загружаем в (регистр-счетчик) CX количество повторов (отсчет будет идти от 10 до 0) |
| Label1: | ; создаем метку (Label - метка). |
| mov ah,9 | ; помещаем номер функции DOS "вывод строки (9)" в регистр АН. |
| mov dx,offset String | помещает в регистр DX смещение метки String относительно начала сегмента данных |
| int 21h | ; функция DOS "вывод строки" |
| loop Label1 | ; оператор loop уменьшает на единицу CX и, если он не равен нулю, переходит на метку Label1 (строка 6) |
| ret | ; функция DOS "завершить программу" |
| string db 'privet $' | ; cтрока с содержащая выводимые данные. |
| end begin | ; метка окончания кода программы |

В строке (5) загружаем в CX количество повторов (отсчет будет идти от 10 до 0). В строке (6) создаем метку (Label - метка). Далее (строки (7) - (9)) выводим сообщение. И в строке (10) оператор loop уменьшает на единицу CX и, если он не равен нулю, переходит на метку Label1 (строка (6)). Таким образом, строка будет выведена на экран десять раз. Когда программа перейдет на строку (11), регистр CX будет равен нулю.

Организация цикла с помощью команды JMP (Второй способ).

Команда jmp передает управление в указанную точку того же или другого программного сегмента. Адрес возврата не сохраняется. Команда не воздействует на флаги процессора.

Команда jmp имеет пять разновидностей:

переход прямой короткий (в пределах - 128... + 127 байтов);

переход прямой ближний (в пределах текущего программного сегмента);

переход прямой дальний (в другой программный сегмент);

переход косвенный ближний;

переход косвенный дальний.

Все разновидности переходов имеют одну и ту же мнемонику jmp, хотя и различающиеся коды операций. Во многих случаях транслятор может определить вид перехода по контексту, в тех же случаях, когда это невозможно, следует использовать атрибутные операторы (short - прямой короткий переход; near ptr - прямой ближний переход; far ptr - прямой дальний переход; word ptr - косвенный ближний переход; dword ptr - косвенный дальний переход).

|  |  |  |
| --- | --- | --- |
| Команда | Назначение | Процессор |
| JMP метка | Безусловный переход | 8086 |

|  |  |
| --- | --- |
| . model tiny | ; модель памяти, в которой сегменты кода, данных и стека объединены. |
| . code | ; сегмент кода, который содержит данные. |
| org 100h | ; начало СОМ-файла |
| begin: | ; метка начала кода программы |
| label1: | ; создаем метку |
| mov ah,9 | ; помещаем номер функции DOS "вывод строки (9)" в регистр АН. |
| mov dx,offset String | помещает в регистр DX смещение метки String относительно начала сегмента данных |
| int 21h | ; функция DOS "вывод строки" |
| jmp Label1 | ; переход на строку с меткой Label1 |
| add cx,12 | ; прибавить к значению регистра cx число 12  (данная команда не выполняется) |
| dec cx | ; уменьшить значение регистра cx на 1  (данная команда не выполняется) |
| ret | ; функция DOS "завершить программу" |
| string db "PRIVET",13,10,'$' | ; cтрока с содержащая выводимые данные. |
| end begin | ; метка окончания кода программы |

В результате работы программы будет зациклен блок строк (6) - (10) (Вывод строки PRIVET многочисленное количество раз) Строки (10) - (11).

Организация цикла с помощью команд DEC и JNZ (Третий способ).

С помощь этих операторов можно создавать циклы, которые будут работать быстрее оператора Loop. Комбинированная работа команд DEC и JNZ уменьшает содержимое регистра CX на 1 и выполняет переход на метку, если в CX не равен нулю.

Команда DEC, кроме того, устанавливает флаг нуля во флаговом регистре в состояние 0 или 1. Команда JNZ затем проверяет эту установку.

Аналогично командам JMP и LOOP операнд в команде JNZ содержит значение расстояния между концом команды JNZ и адресом перехода (Label1), которое прибавляется к командному указателю. Это расстояние должно быть в пределах от - 128 до +127 байт.

Следующий пример будет работать так же, как и Пример №1.1, только быстрее.

|  |  |
| --- | --- |
| . model tiny | ; модель памяти, в которой сегменты кода, данных и стека объединены. |
| . code | ; сегмент кода, который содержит данные. |
| org 100h | ; начало СОМ-файла |
| begin: | ; метка начала кода программы |
| mov cx,10 | ; загружаем в (регистр-счетчик) CX количество повторов (отсчет будет идти от 10 до 0) |
| Label1: | ; создаем метку (Label - метка). |
| mov ah,9 | ; помещаем номер функции DOS "вывод строки (9)" в регистр АН. |
| mov dx,offset String | помещает в регистр DX смещение метки String относительно начала сегмента данных |
| int 21h | ; функция DOS "вывод строки" |
| dec cx | ; оператор DEC уменьшает на единицу CX и, если он не равен нулю, переходит на метку Label1 |
| jnz Label1 | ; условный переход на строку с меткой Label1 |
| ret | ; функция DOS "завершить программу" |
| string db 'priver ',13,10, '$' | ; cтрока с содержащая выводимые данные |
| end begin | ; метка окончания кода программы |

2. Программа для практики.

Напишем программу, выводящую на экран все ASCII-символы (16 строк по 16 символов в строке).

|  |  |
| --- | --- |
| . model tiny | ; модель памяти в которой сегменты кода, данных и стека объединены. |
| . code | ; сегмент кода, который содержит данные. |
| org 100h | ; начало СОМ-файла |
| begin: | ; метка начала кода программы |
| mov cx,256 | ; задаем значение счетчика (256 символов) |
| mov dl,0 | ; первый символ - с кодом 00 |
| mov ah,2 | ; номер функции DOS "вывод символа" |
| cloop: int 21h | ; вызов DOS |
| inc dl | ; увеличение DL на 1 - следующий символ |
| test dl,0Fh | ; если DL не кратен 16 |
| jnz continue\_loop; | ; продолжить цикл, |
| push dx | ; иначе: сохранить текущий символ |
| mov dl,0Dh | ; вывести CR |
| int 21h | ; вызов DOS |
| mov dl,0Ah | ; вывести LF |
| int 21h | ; вызов DOS |
| pop dx | ; восстановить текущий символ |
| continue\_loop: | ; метка |
| loop cloop | ; продолжить цикл |
| ret | ; завершение СОМ-файла |
| end begin | ; метка окончания кода программы |

Здесь с помощью команды LOOP оформляется цикл, выполняющийся 256 раз (значение регистра СХ в начале цикла). Регистр DL содержит код символа, который равен нулю в начале цикла и увеличивается каждый раз на 1 командой INC DL. Если значение DL сразу после увеличения на 1 кратно 16, оно временно сохраняется в стеке и на экран выводятся символы CR и LF, выполняющие переход на начало новой строки. Проверка выполняется командой TEST DL,0Fh - результат операции AND над DL и 0Fh будет нулем, только если младшие четыре бита DL равны нулю, что и соответствует кратности шестнадцати.

3. Содержание отчета.

3.1 Титульный лист.

3.2 Индивидуальный вариант задания.

3.3 Тестовые наборы данных и предполагаемые результаты.

3.4 Текст программы до отладки.

3.5 Список ошибок, обнаруженных при отладке.

3.6. Результаты выполнения тестов.

3.7. Распечатка листинга компиляции отлаженной программы с указанием работы каждой строки.

4. Задание для выполнения.

4.1 Выполните все примеры, что содержатся в описании данной лабораторной работы.

4.2 Проанализируйте работу программы примера для практики.

4.3 Изучить условия организации циклических переходовна языке Ассемблера.

4.4 Напишите программу, выводящую на экран слово "!!!!!!!!!! Hello!!!!!!!!!!" используя команды циклических переходов (3 варианта).

4.5 Получите задание у преподавателя (один из пяти вариантов табл. №1) и, пользуясь правилами оформления ассемблерных программ, создайте программу, выводящую на экран слово, D число раз.

4.6 Программу ассемблируйте в файл типа \*.com или \*. exe (на выбор);

5. Контрольные вопросы

5.1 Организация цикла с помощью команды loop?

5.2 Значимость регистра cx?

5.3 Максимальное число повторений команд цикла определяемого регистром сх?

5.4 Организация цикла с помощью команды jmp?

5.5. Разновидности команды jmp?

5.6. Организация цикла с помощью команд dec и jnz?

Табл. №1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| №  вар. | Выводимые данные | Формула расчета | А | B | С |
|  | Циклический переход | D=A+B+C | 101 | 345 | 121 |
|  | Hello world | D=A-B+C | 578 | 152 | 149 |
|  | Good Bye | D=A+B-C | 333 | 223 | 16 |
|  | Группа | D=A-B+C | 1502 | 834 | 1 |
|  | Лабораторная работа | D=A-B-C | 1056 | 33 | 125 |

ЛАБОРАТОРНАЯ РАБОТА №9

СПОСОБЫ И МЕТОДЫ ВЫВОДА ЧИСЕЛ

Цель работы: Освоить методы вывода чисел в двоичном, **шестнадцатеричном и** **десятичном** коде.

1. Вывод двоичного кода числа, записанного в регистр DH.

Методика выполнения.

Нужно последовательно проанализировать биты числа. В данной работе ограничимся байтом, который будем хранить в DH. Если бит нулевой (сброшен), то нужно вывести '0', если установлен, то '1'.

*Алгоритм решения задачи:*

проанализировать значение одного бита;

вывести значение бита;

перейти к следующему биту. И так 8 раз (БАЙТ!) - ЦИКЛ

Анализ бита: **При анализе** значения программисты обычно используют команду TEST, но в данной лабораторной работе мы будет использовать следующую команду SHL

Команда SHL осуществляет сдвиг влево всех битов операнда. Старший бит операнда поступает в флаг CF. Если команда записана в формате

SHL операнд, 1

сдвиг осуществляется на 1 бит. В младший бит операнда загружается 0. Если команда записана в формате

SHL операнд,CL

сдвиг осуществляется на число битов, указанное в регистре-счетчике CL, при этом в процессе последовательных сдвигов старшие биты операнда, пройдя через флаг CF, теряются, а младшие заполняются нулями

В качестве операнда команды SHL можно указывать любой регистр (кроме сегментного) или ячейку памяти размером, как в байт, так и в слово. Не допускается использовать в качестве операнда непосредственное значение.

Каждый сдвиг влево эквивалентен умножению знакового числа на 2, поэтому команду SHL удобно использовать для возведения операнда в степень 2.

Команда воздействует на флаги OF, SF, ZF, PF и CF.

Прореагировать на значение флага можно с помощью команды

JNC <метка>

Осуществляется переход на метку, если флаг CF равен нулю, иначе выполняется команда, непосредственно идущая после команды.

Пример №1.1

|  |  |
| --- | --- |
| . model tiny | ; модель памяти, в которой сегменты кода, данных и стека объединены. |
| . code | ; сегмент кода, который содержит данные. |
| org 100h | ; начало СОМ-файла |
| begin: | ; метка начала кода программы |
| mov DH,<55> | ; заносим в регистр dh - любой ASCII-код символа |
| mov AH,2 | ; помещаем номер функции DOS "вывод строки (2)" в регистр АН. |
| mov CX,8 | ; инициализация переменной цикла |
| @1: mov DL,'0' | ; заносим в DL код символа '0' |
| shl DH, 1 | ; сдвиг на 1 бит |
| jnc @2 | ; переход, если '0' |
| inc DL | ; используем тот факт, что код символа '1' на единицу больше кода символа '0' |
| @2: int 21h | ; вызов прерывания DOS - вызов символа; |
| LOOP @1 | ; переходим на метку @1 |
| int 21h | ; вызов прерывания DOS - вызов символа; |
| end begin | ; метка окончания кода программы |

Замечание: в описанном способе анализа значение исходного числа теряется. Иначе следует использовать команду ROL, но 8 раз для байта.

2. Вывод значения байта в шестнадцатеричной системе счисления

2.1 Методика выполнения.

*Алгоритм решения задачи:*

Допустим, что байт, значение которого нужно вывести, находится в регистре DH, и имеется таблица символов "0123456789ABCDEF". Байт состоит из двух шестнадцатеричных цифр. С учетом этого задачу можно решить так: нужно вывести на экран два символа из этой таблицы. Сначала - символ с номером, равным старшему полубайту числа, а потом с номером, равным младшему полубайту.

Для решения задачи нужно решить две небольшие проблемы:

Записать в AL символ с нужным номером. Воспользуемся регистровым косвенным режимом адресации со смещением. Для этого значение каждого полубайта следует записывать в BX;

Записать в BX значение полубайта.

Простейший способ решения задачи.

Пример №2.1

|  |  |
| --- | --- |
| . model tiny | ; модель памяти, в которой сегменты кода, данных и стека объединены. |
| . code | ; сегмент кода, который содержит данные. |
| org 100h | ; начало СОМ-файла |
| begin: | ; метка начала кода программы |
| mov dh, 10 | ; заносим в регистр dh число 10 |
| mov bl, dh | ; заносим в регистр bl число 10 |
| xor bh, bh | ; Обнуление вх |
| and bl, 0F0h | ; осуществляем логическое (побитовое) умножение dh на 0f0h. |
| shr bl, 4 | ; сдвиг в право на 4 бита |
| mov al, table [bx] | ; заносим в регистр al значение строки данных |
| int 29h | ; вызов прерывания DOS - вызов символа |
| mov bl, dh | ; заносим в регистр bl значение регистра dh |
| and bl, 0Fh | ; осуществляем логическое (побитовое) умножение dh на 0fh. |
| mov al, table [bx] | ; заносим в регистр al значение строки данных |
| int 29h | ; вызов прерывания DOS - вызов символа; |
| mov al, 13 | ; заносим в регистр al число 13 |
| int 29h | ; вызов прерывания DOS - вызов символа; |
| mov al, 10 | ; заносим в регистр al число 10 |
| int 29h | ; вызов прерывания DOS - вызов символа |
| ret | ; функция DOS "завершить программу" |
| table db '0123456789ABCDEF' | ; cтрока с содержащая выводимые данные. |
| end begin | ; метка окончания кода программы |

## 3. Вывод значения байта в десятеричной системе счисления

## 3.1 Методика выполнения

*Алгоритм решения задачи:*

**Будем считать,** что байт, значение которого нужно вывести, находится в регистре DH. Однако теперь применим другой способ вывода символа цифры на экран: используем тот факт, что коды символов, обозначающих цифры, отличаются от них на 30h. Но проблема здесь другая: заранее не известно, сколько цифр нужно отобразить, одну или три. Байт может принимать значение от 0 до 255. И есть еще одна проблема. При записи числа с применением позиционной системы записи в некоторой системе счисления поступают следующим образом: вычисляют и записывают остатки от деления числа на основание системы. Так поступают до тех пор, пока частное от деления не станет равным нулю. Затем остатки выписывают в порядке, обратном тому, как они получены.

Пример:

число = 251.

Делим на 10. Частное 25, остаток "1".

Делим на 10. Частное 2, остаток "5".

Делим на 10. Частное 0, остаток "2".

Нужно вывести на экран "2", "5", "1".

Задача решается с использованием стека программы. Остатки будем помещать в стек программы с помощью оператора PUSH. Одновременно будем подсчитывать число остатков, помещенных в стек. Счетчик - CX. Потом его используем для организации цикла, в котором будем извлекать остатки из стека оператором POP. Стек организован таким образом, что оператор POP извлекает последнее слово, которое было помещено туда оператором PUSH. Отметим, что оператор PUSH помещает в стек слово (WORD) или двойное слово (DWORD). Аналогично работает и оператор POP

3.1.1 Команда PUSH (занесение операнда в стек).

Команда push уменьшает на 2 содержимое указателя стека SP и заносит на эту новую вершину двухбайтовый операнд-источник (проталкивает в стек новое данное). Проталкивать в стек можно только целые слова (не байты). Программа должна строго следить за тем, чтобы каждой команде проталкивания в стек push отвечала обратная команда выталкивания из стека pop. Если стек используется для временного хранения некоторых данных, то извлекать эти данные из стека следует в порядке, обратном их сохранению.

В качестве операнда-источника может использоваться любой 16-разрядный регистр (включая сегментный) или ячейка памяти. Команда push не воздействует на флаги процессора.

Пара команд push - pop часто используется для пересылки данного из регистра в регистр (особенно, в сегментный) через стек.

**Пример 1**

push ES: mem**; Сохранение содержимого**

push DS**; слова памяти mem из**

push BP**; дополнительного сегмента, а также регистров DS и ВР**

pop PP**; Восстановление из стека**

pop DS**; трех операндов**

pop ES: mem**; в обратном порядке**

**Пример 2**

push DS**; Пересылка DS через стек**

pop ES**; Теперь ES=DS**

Простейший способ решения задачи вывода значения байта в десятеричной системе счисления

Пример № 3.1

|  |  |
| --- | --- |
| . model tiny | ; модель памяти, в которой сегменты кода, данных и стека объединены. |
| . code | ; сегмент кода, который содержит данные. |
| org 100h | ; начало СОМ-файла |
| begin: | ; метка начала кода программы |
| mov dh, 33 | ; заносим в регистр dh число 33 |
| mov al, dh | ; заносим в регистр al число 33 |
| xor ah, ah | ; обнуление ah |
| mov bl, 10 | ; заносим в регистр bl число 10 |
| xor cx, cx | ; обнуление сх (счетчик) |
| @1: | ; устанавливаем метку @1 |
| div bl | ; делим целое число (регистр ax) без знака, на число (регистр bl) |
| push ax | ; **пересылаем** ax **через стек** |
| inc cx | ; увеличиваем на 1 cx (счетчик) |
| xor ah, ah | ; обнуление ah |
| or ax, ax | ; логического (побитового) сложения само на себя |
| jnz @1 | ; если флаг zf не ноль, то переходим на метку @1 |
| @2: | ; вызов прерывания DOS - вызов символа; |
| pop ax | ; заносим в регистр al число 10 |
| xchg ah, al | ; обмен данными между операндами ah и al |
| add al, 30h | ; прибавляем к al число 30h |
| int 29h | ; вызов прерывания DOS - вызов символа |
| loop @2 | ; реализация цикла - переходим на метку @2 |
| mov al, 13 | ; заносим в регистр al число 13 |
| int 29h | ; вызов прерывания DOS - вызов символа |
| mov al, 10 | ; заносим в регистр al число 10 |
| int 29h | ; вызов прерывания DOS - вызов символа |
| ret | ; функция DOS "завершить программу" |
| end begin | ; метка окончания кода программы |

Задание для выполнения.

3.1 C помощью редактора эмулятора EMU 8086 напишите программы, исходный текст которых приводится в примерах данной лабораторной работы.

3.2 Создайте исполняемые файлы типа \*.com.

3.3 Изучите работу полученных программ.

3.4 Напишите программу для вывода на экран содержимого регистра DS (на основе примера №2.1). Сравните результат работы своей программы и того, что показывает отладчик.

3.5 Опишите работу команд DIV, PUSH, POP, SHL, TEST.

3.6 Установите (найдите адреса и запишите), где находятся числа, помещенные в стек.

3.7 Напишите программу для вывода на экран содержимого регистра СS (на основе примера №3.1).

3.8 Предложите другие способы решения поставленных задач.

5. Контрольные вопросы

Преимущества использования команды SHL вместо TEST (пример №1.1)?

Чем отличаются команды

SHL dx,1

*и*

SHL dx, cl

Как переслать содержимое X в стек и получить обратно?

Опишите методику вывода значения байта в десятеричной системе счисления?

Опишите методику вывода значения байта в шестнадцатеричной системе счисления?

Опишите методику вывода двоичного кода числа, записанного в регистр X

Стек. Принцип работы. Команды работы со стеком.

**Укажите отличия в работе тандема команд**

push DS

pop ES

*от*

push DS

pop ES

## Лабораторная работа № 5, 6

КОМАНДЫ, ОБСЛУЖИВАЮЩИЕ РАБОТУ С КЛАВИАТУРОЙ

Цель работы: Освоить команды **считывания данных и управления клавиатурой**. Изучить способы работы с процедурами.

1. Средства BIOS

Так же как и для вывода на экран, BIOS предоставляет больше возможностей по сравнению с DOS для считывания данных и управления клавиатурой. Например, функциями DOS нельзя определить нажатие комбинаций клавиш типа Ctrl-Alt-Enter или нажатие двух клавиш Shift одновременно, DOS не может определить момент отпускания нажатой клавиши, и наконец, в DOS нет аналога функции С ungetch (), помещающей символ в буфер клавиатуры, как если бы его ввел пользователь. Все это можно осуществить, используя различные функции прерывания 16h и операции с байтами состояния клавиатуры.

INT 16h, АН **= 0, 10h, 20h - Чтение символа с ожиданием**

|  |  |
| --- | --- |
| Ввод: | АН = 00h (83/84-key), 10h (101/102-key), 20h (122-key) |
| Вывод: | AL = ASCII-код символа, 0 или префикс скан-кода  АН = скан-код нажатой клавиши или расширенный ASCII-код |

Каждой клавише на клавиатуре соответствует так называемый скан-код (см. приложение 1), соответствующий только этой клавише. Этот код посылается клавиатурой при каждом нажатии и отпускании клавиши и обрабатывается BIOS (обработчиком прерывания INT 9). Прерывание 16h дает возможность получить код нажатия, не перехватывая, этот обработчик. Если нажатой клавише соответствует ASCII-символ, то в АН возвращается код этого символа, а в AL - скан-код клавиши. Если нажатой клавише соответствует расширенный ASCII-код, в AL возвращается префикс скан-кода (например, Е0 для серых клавиш) или 0, если префикса нет, а в АН - расширенный ASCII-код. Функция 00Н обрабатывает только комбинации, использующие клавиши 84-клавишной клавиатуры, l0h обрабатывает все 101 - 105-клавишные комбинации, 20h - 122-клавишные. Тип клавиатуры можно определить с помощью функции 09h прерывания 16h, если она поддерживается BIOS (поддерживается ли эта функция, можно узнать с помощью функции C0h прерывания 15h).

INT 16h, АН = 1, 11h, 21h **- Проверка символа**

|  |  |
| --- | --- |
| Ввод: | АН = 01h (83/84-key), 11h (101/102-key), 21h (122-key) |
| Вывод: | ZF = 1, если буфер пуст  ZF = 0, если в буфере присутствует символ, в этом случае  AL = ASCII-код символа, 0 или префикс скан-кода  АН = скан-код нажатой клавиши или расширенный ASCII-код |

Символ остается в буфере клавиатуры, хотя некоторые BIOS удаляют символ из буфера при обработке функции 01h, если он соответствует расширенному ASCII-коду, отсутствующему на 84-клавишных клавиатурах.

INT 16h, АН = 05h **- Поместить символ в буфер клавиатуры**

|  |  |
| --- | --- |
| Ввод: | АН = 05h  СН = скан-код  CL = ASCII-код |
| Вывод: | AL = 00, если операция выполнена успешно  AL = 01h, если буфер клавиатуры переполнен  АН модифицируется многими BIOS |

Обычно можно поместить 0 вместо скан-кода в СН, если функция, которая будет выполнять чтение из буфера, будет использовать именно ASCII-код.

Например, следующая программа при запуске из DOS вызывает команду DIR (но при запуске из некоторых оболочек, например FAR, этого не произойдет).

**Пример №1.1**

|  |  |
| --- | --- |
| . model tiny | ; модель памяти, в которой сегменты кода, данных и стека объединены. |
| . code | ; сегмент кода, который содержит данные. |
| org 100h | ; начало СОМ-файла |
| begin: | ; метка начала кода программы |
| mov cl,'d' | ; заносим в регистр cl - ASCII-код буквы "d" |
| call ungetch | ; переходим на метку ungetch: (вызов подпрограммы) |
| mov cl,'i' | ; заносим в регистр cl - ASCII-код буквы "i" |
| call ungetch | ; вызываем подпрограмму |
| mov cl,'r' | ; заносим в регистр cl - ASCII-код буквы "r" |
| call ungetch | ; вызываем подпрограмму |
| mov cl,0Dh | ; перевод строки |
| Ungetch proc | ; метка начала подпрограммы |
| mov ah,05h | ; AH = номер функции |
| mov ch,0 | ; CH = 0 (скан-код неважен) |
| int 16h | ; вызов DOS (прерывание) |
| ret | ; функция DOS "завершить работу процедуры" |
| Ungetch endp | ; окончание подпрограммы |
| end begin | ; метка окончания кода программы |

**Оформление процедур (подпрограмм).**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Call ungetch

Ungetch **proc**

Ungetch **endp**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Ungetch **- название процедуры**

Call - **вызов подпрограммы**

Proc **- procedure - процедура**

endp **- end procedure - конец процедуры**

Стоит еще отметить, что из одной подпрограммы можно вызывать другие. Из других - третьи. Чаще всего с целью упрощения программы и, тем самым, уменьшения ее размера программисты используют простую связку команд:

Call метка

метка:

**Любой вызов подпрограмм заканчивается возращение в блок кода командой ret (**ret **- return - возврат).**

INT 16h, AH = 02h, 12h, 22h **- Считать состояние клавиатуры**

|  |  |
| --- | --- |
| Ввод: | АН = 02h (83/84-key), 12h (101/102-key), 22h (122-key) |
| Вывод: | AL = байт состояния клавиатуры 1  АН = байт состояния клавиатуры 2 (только для функций 12h и 22h) |

Байт состояния клавиатуры 1 (этот байт всегда расположен в памяти по адресу 0000h: 0417h или 0040h: 0017h):

Бит 7: Ins включена

Бит 6: CapsLock включена

Бит 5: NumLock включена

Бит 4: ScrollLock включена

Бит 3: Alt нажата (любая Alt для функции 02h, часто только левая Alt для 12h/22h)

Бит 2: Ctrl нажата (любая Ctrl)

Бит 1: Левая Shift нажата

Бит 0: Правая Shift нажата

Байт состояния клавиатуры 2 (этот байт всегда расположен в памяти по адресу 0000h: 0418h или 0040h: 0018h):

Бит 7: SysRq нажата

Бит 6: CapsLock нажата

Бит 5: NumLock нажата

Бит 4: ScrollLock нажата

Бит 3: Правая Alt нажата

Бит 2: Правая Ctrl нажата

Бит 1: Левая Alt нажата

Бит 0: Левая Ctrl нажата

Оба этих байта постоянно располагаются в памяти, так что вместо вызова прерывания часто удобнее просто считывать значения напрямую. Более того, в эти байты можно записывать новые значения, и BIOS изменит состояние клавиатуры соответственно:

Помимо этих двух байт BIOS хранит в своей области данных и весь клавиатурный буфер, к которому также можно обращаться напрямую. Буфер занимает 16 слов с 0h: 041Eh no 0h: 043Dh включительно, причем по адресу 0h: 041Ah лежит адрес (ближний) начала буфера, то есть адрес, по которому располагается следующий введенный символ, а по адресу 0h: 041Ch лежит адрес конца буфера, так что если эти два адреса равны, буфер пуст. Буфер действует как кольцо: если начало буфера - 043Ch, а конец - 0420h, то в буфере находятся три символа по адресам 043Ch, 041Eh и 0420h. Каждый символ хранится в виде слова - того же самого, которое возвращает функция 10h прерывания INT 16h. В некоторых случаях (если) буфер размещается по другим адресам, тогда адрес его начала хранится в области данных BIOS по адресу 0480h, а конца - по адресу 0482h. Прямой доступ к буферу клавиатуры лишь немногим быстрее, чем вызов соответствующих функций BIOS, и для приложений, требующих максимальной скорости, таких как игры или демо-программы, используют управление клавиатурой на уровне портов ввода-вывода.

2. Средства DOS

Как и в случае вывода на экран, DOS предоставляет набор функций для чтения данных с клавиатуры, которые используют стандартное устройство ввода STDIN, так что можно использовать в качестве источника данных файл или стандартный вывод другой программы.

Функция DOS 0Ah - Считать строку символов из STDIN в буфер

|  |  |
| --- | --- |
| Ввод: | АН = 0Ah  DS: DX = адрес буфера |
| Вывод: | Буфер содержит введенную строку |

Для вызова этой функции надо подготовить буфер, первый байт которого содержит максимальное число символов для ввода (1-254), а содержимое, если оно задано, может использоваться как подсказка для ввода. При наборе строки обрабатываются клавиши Esc, F3, F5, BS, Ctrl-C/Ctrl-Break и т.д., как при наборе команд DOS (то есть Esc начинает ввод сначала, F3 восстанавливает подсказку для ввода, F5 запоминает текущую строку как подсказку, Backspace стирает предыдущий символ). После нажатия клавиши Enter строка (включая последний символ CR (0Dh)) записывается в буфер, начиная с третьего байта. Во второй байт записывается длина реально введенной строки без учета последнего CR.

Рассмотрим пример программы, выполняющей преобразование десятичного числа в шестнадцатеричное.

Пример №2.1

|  |  |
| --- | --- |
| . model tiny | ; модель памяти, в которой сегменты кода, данных и стека объединены. |
| . code | ; сегмент кода, который содержит данные. |
| org 100h | ; начало СОМ-файла |
| start: | ; метка начала кода программы |
| mov dx,offset messagel | ; DS: DX - адрес строки |
| mov ah,9 | ; номер функции DOS в АН |
| int 21h | ; вывести приглашение ко вводу message1 |
| mov dx,offset buffer | ; DS: DX - адрес строки |
| mov ah,0Ah | ; номер функции DOS в АН |
| int 21h | ; считать строку символов в буфер |
| mov dx,offset crlf | ; DS: DX - адрес строки |
| mov ah,9 | ; номер функции DOS в АН |
| int 21h | ; перевод строки |
| xor di,di | ; DI = 0 - номер байта в буфере |
| xor ax,ax | ; АХ = 0 - текущее значение результата |
| mov cl,blength |  |
| xor ch,ch | ; обнуляем регистр ch |
| xor bx,bx | ; обнуляем регистр bx |
| mov si,cx | ; SI - длина буфера |
| mov cl,10 | ; CL = 10, множитель для MUL |
| asc2hex: | ; метка начала блока asc2hex: |
| mov bl,byte ptr bcontents [di] |  |
| sub bl,'0' | ; цифра = код цифры - код символа "0", |
| jb asc\_error | ; если код символа был меньше, чем код "0", |
| cmp bl,9 | ; или больше, чем "9", |
| ja asc\_error | ; выйти из программы с сообщением об ошибке, |
| mul cx | ; иначе: умножить текущий результат на 10, |
| add ax,bx | ; добавить к нему новую цифру, |
| inc di | ; увеличить счетчик |
| cmp di,si | ; если счетчик+1 меньше числа символов - |
| jb asc2hex | ; продолжить (счетчик считается от 0) |
| push ax | ; сохранить результат преобразования |
| mov ah,9 | ; номер функции DOS в АН |
| mov dx,offset message2 | ; DS: DX - адрес строки |
| int 21h | ; вывести приглашение ко вводу message2 |
| pop ax | ; считать из стека |
| push ax | ; записать в стек |
| xchg ah,al | ; поместить в AL старший байт |
| call print\_al | ; вывести его на экран |
| pop ax | ; восстановить в AL младший байт |
| call print\_al | ; вывести его на экран |
| ret | ; завершение СОМ-файла |
| asc\_error: | ; начало блока asc\_error: |
| mov dx,offset err\_msg | ; DS: DX - адрес строки |
| mov ah,9 | ; номер функции DOS в АН |
| int 21h | ; вывести сообщение об ошибке |
| ret | ; завершить программу |
| print\_al: | ; метка начала блока print\_al: |
| mov dh,al | ; заносим в dh значение регистра al |
| and dh,0Fh | ; DH - младшие 4 бита |
| shr al,4 | ; AL - старшие |
| call print\_nibble | ; вывести старшую цифру |
| mov al,dh | ; теперь AL содержит младшие 4 бита |
| print\_nibble: | ; процедура вывода 4 бит (шестнадцатеричной цифры) |
| cmp al,10 | ; три команды, переводящие цифру в AL |
| sbb al,69h | ; в соответствующий ASCII-код |
| das | ; (см. описание команды DAS) |
| mov dl,al | ; код символа в DL |
| mov ah,2 | ; номер функции DOS в АН |
| int 21h | ; вывод символа |
| ret | ; этот RET работает два раза - один раз для возврата из процедуры print\_nibble, вызванной для старшей цифры и второй раз - для возврата из print\_al |
| messagel db "Десятичное число: $" | ; cтрока с содержащая выводимые данные. |
| message2 db "Шестнадцатеричное число: $" | ; cтрока с содержащая выводимые данные. |
| err\_msg db "Ошибка ввода" | ; cтрока с содержащая выводимые данные. |
| crlf db 0Dh,0Ah, '$' | ; cтрока с содержащая выводимые данные. |
| Buffer db 6 | ; максимальный размер буфера ввода |
| blength db? | ; размер буфера после считывания |
| bcontents: | ; содержимое буфера располагается за концом СОМ-файла |
| end start | ; метка окончания кода программы |

Функция 0Ah предоставляет удобный, но ограниченный способ ввода данных. Чаще всего используют функции посимвольного ввода, позволяющие контролировать отображение символов на экране, реакцию программы на функциональные и управляющие клавиши и т.д.

Функция DOS 01h - Считать символ из STDIN с эхом, ожиданием и проверкой на Ctrl-Break

|  |  |
| --- | --- |
| Ввод: | АН = 01h |
| Вывод: | AL = ASCII-код символа или 0. Если AL = 0, второй вызов этой функции возвратит в AL расширенный ASCII-код символа |

При чтении с помощью этой функции введенный символ автоматически немедленно отображается на экране (посылается в устройство STDOUT - так что его можно перенаправить в файл). При нажатии Ctrl-C или Ctrl-Break выполняется команда INT 23h. Если нажата клавиша, не соответствующая какому-нибудь символу (стрелки, функциональные клавиши Ins, Del и т.д.), то в AL возвращается 0 и функцию надо вызвать еще один раз, чтобы получить расширенный ASCII-код (см. приложение 1).

В трех следующих вариантах этой функции код символа возвращается в AL по такому же принципу.

Функция DOS 08h - Считать символ из STDIN без эха, с ожиданием и проверкой на Ctrl-Break

|  |  |
| --- | --- |
| Ввод: | АН = 08h |
| Вывод: | AL = код символа |

Функция DOS 07h - Считать символ из STDIN без эха, с ожиданием и без проверки на Ctrl-Break

|  |  |
| --- | --- |
| Ввод: | АН = 07h |
| Вывод: | AL = код символа |

Функция DOS 06h - Считать символ из STDIN без эха, без ожидания и без проверки на Ctrl-Break

|  |  |
| --- | --- |
| Ввод: | АН = 07h  DL = 0FFh |
| Вывод: | ZF = 1, если не была нажата клавиша, и AL = 00  ZF = 0, если клавиша была нажата. В этом случае AL = код символа |

2.2 служебные функции DOS для работы с клавиатурой.

Кроме перечисленных функций используются некоторые служебные функции DOS для работы с клавиатурой.

Функция DOS 0Bh - Проверить состояние клавиатуры

|  |  |
| --- | --- |
| Ввод: | АН = 0Bh |
| Вывод: | AL = 0, если не была нажата клавиша  AL = 0FFh, если была нажата клавиша |

Эту функцию удобно использовать перед функциями 01, 07 и 08, чтобы не ждать нажатия клавиши. Кроме того, вызов этой функции позволяет проверить, не считывая символ с клавиатуры, была ли нажата комбинация клавиш Ctrl-Break; если это произошло, выполнится прерывание 23h.

Функция DOS 0Ch - Очистить буфер и считать символ

|  |  |
| --- | --- |
| Ввод: | АН = 0Ch  AL = Номер функции DOS (01, 06, 07, 08, 0Ah) |
| Вывод: | Зависит от вызванной функции |

Функция 0Ch очищает буфер клавиатуры, так что следующая функция чтения символа будет ждать ввода с клавиатуры, а не использовать нажатый ранее и еще не обработанный символ. Например, именно эта функция используется для считывания ответа на вопрос "Уверен ли пользователь в том, что он хочет отформатировать диск?".

Функции посимвольного ввода без эха можно использовать для интерактивного управления программой.

|  |  |
| --- | --- |
| . model tiny | ; модель памяти, в которой сегменты кода, данных и стека объединены. |
| . code | ; сегмент кода, который содержит данные. |
| org 100h | ; начало СОМ-файла |
| Begin: | ; метка начала кода программы |
| call Wait\_key | ; вызываем подпрограмму |
| cmp al,27 | ; сравниваем значение в al с кодом 27 (ESC)  ; если да - то на метку Quit\_prog |
| je Quit\_prog |
| cmp al,0 | ; код клавиши расширенный? (F1-F12 и т.п.) |
| je Begin | ; да - повторим запрос... |
| call Out\_char | ; вызываем процедуру вывода нажатой клавиши на экран |
| jmp Begin | ; ждем дальше... |
| Quit\_prog: | ; метка, на которую придет программа в случае нажатия ESC |
| mov al,32 | ; помещаем в AL <пробел> |
| call Out\_char | ; вызываем процедуру вывода символа в AL |
| int 20h | ; выходим... |
| Wait\_key proc | ; процедура ожидания клавиши от пользователя |
| mov ah,10h | ; окончание подпрограммы |
| int 16h | ; прерывание DOS |
| ret | ; функция DOS "завершить работу процедуры" |
| Wait\_key endp | ; окончание процедуры Wait\_key |
| Out\_char proc | ; начало процедуры out\_char |
| push cx | ; сохраним все регистры, которые будут изменены подпрограммой... |
| push ax | ;... сделаем это для того, чтобы в последствии не было путаницы |
| push es | ; сохраним сегментный регистр |
| push ax | ; сохраним AX, т.к в нем код нажатой клавиши... |
| mov ax,0B800h | ; установим ES на сегмент видеобуфера |
| mov es,ax |
| mov di,0 | ; DI - первый символ первой строки |
| mov cx, 2000 | ; выводим 2000 символов (80 символов в строке 25 строк) |
| pop ax | ; восстановим код клавиши |
| mov ah,31 | ; цвет символа |
| Next\_sym: | ; метка для цикла |
| mov es: [di],ax | ; заносим код клавиши и ее цвет (цвет всегда 31) |
| inc di | ; увеличиваем указатель на 2 (первый байт - символ, второй байт - цвет) |
| inc di |
| loop Next\_sym | ; обработка следующего символа |
| pop es | ; восстановим сохраненные регистры и выровним стек |
| pop ax |
| pop cx |
| ret | ; вернемся из процедуры |
| Out\_char endp | ; окончание процедуры Out\_char |
| end Begin | ; метка окончания кода программы |

Программа делает следующее:

Ждет от пользователя клавиши;

если это расширенный ASCII (F1-F12, стрелки), то игнорирует ее;

если это не расширенный ASCII (A-Z, 0-9 и т.п.) - заполнить экран данным символом;

если нажимаем ESC (27 или 1Bh), то заполнить экран пробелами (mov al,32) и выйти.

Задание для выполнения.

3.1 C помощью редактора эмулятора EMU 8086 напишите программы, исходный текст которых приводится в примерах данной лабораторной работы.

3.2 Создайте исполняемые файлы типа \*.com.

3.3 Изучите работу полученных программ.

3.4 Напишите программу для вывода на экран содержимого регистра DS (на основе примера №2.1). Сравните результат работы своей программы и того, что показывает отладчик.

3.5 Опишите работу команд DIV, PUSH, POP, SHL, TEST.

3.6 Установите (найдите адреса и запишите), где находятся числа, помещенные в стек.

3.7 Напишите программу для вывода на экран содержимого регистра СS (на основе примера №3.1).

3.8 Предложите другие способы решения поставленных задач.

5. Контрольные вопросы

Преимущества использования команды SHL вместо TEST (пример №1.1)?

Чем отличаются команды

SHL dx,1

*и*

SHL dx, cl

Как переслать содержимое X в стек и получить обратно?

Опишите методику вывода значения байта в десятеричной системе счисления?

Опишите методику вывода значения байта в шестнадцатеричной системе счисления?

Опишите методику вывода двоичного кода числа, записанного в регистр X

Стек. Принцип работы. Команды работы со стеком.

**Укажите отличия в работе тандема команд.**

push DS

pop ES

**от**

push DS

pop ES

Приложение №1

**Основные Скан-Коды клавиш клавиатуры.**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Клавиша | Код | Клавиша | Код | Клавиша | Код | Клавиша | Код |
| Esc | 01h | Enter | 1Ch | K\* | 37h | Ins | 52h |
| 1! | 02h | Ctrl | 1Dh | Alt | 38h | Del | 53h |
| 2 @ | 03h | A | 1Eh | SP | 39h | SysRq | 54h |
| 3 # | 04h | S | 1Fh | Caps | 3Ah | Macro | 56h |
| 4 $ | 05h | D | 20h | F1 | 3Bh | F11 | 57h |
| 5% | 06h | F | 21h | F2 | 3Ch | F12 | 58h |
| 6 ^ | 07h | G | 22h | F3 | 3Dh | PA1 | 5Ah |
| 7 & | 08h | H | 23h | F4 | 3Eh | F13/LWin | 5Bh |
| 8 \* | 09h | J | 24h | F5 | 3Fh | F14/RWin | 5Ch |
| 9 ( | 0Ah | K | 25h | F6 | 40h | F15/Menu | 5Dh |
| 0) | 0Bh | L | 26h | F7 | 41h | F16 | 63h |
| - \_ | 0Ch | ;: | 27h | F8 | 42h | F17 | 64h |
| = + | 0Dh | ' " | 28h | F9 | 43h | F18 | 65h |
| BS | 0Eh | ` ~ | 29h | F10 | 44h | F19 | 66h |
| Tab | 0Fh | LShift | 2Ah | Num | 45h | F20 | 67h |
| Q | 10h | \ | | 2Bh | Scroll | 46h | F21 | 68h |
| W | 11h | Z | 2Ch | Home | 47h | F22 | 69h |
| E | 12h | X | 2Dh | - | 48h | F23 | 6Ah |
| R | 13h | C | 3Eh | PgUp | 49h | F24 | 6Bh |
| T | 14h | V | 2Fh | K- | 4Ah | EraseEOF | 6Dh |
| Y | 15h | B | 30h |  | 4Bh | Copy/Play | 6Fh |
| U | 16h | N | 31h | K5 | 4Ch | CrSel | 72h |
| I | 17h | M | 32h | ® | 4Dh | Delta | 73h |
| O | 18h | , < | 33h | K+ | 4Eh | ExSel | 74h |
| P | 19h | . > | 34h | End | 4Fh | Clear | 76h |
| [{ | 1Ah | /? | 35h | I | 50h |  |  |
| ] } | 1Bh | RShift | 36h | PgDn | 51h |  |  |

3. Создание лабораторного практикума "Операционная система"

Операционная система состоит из следующих компонентов:

"Собственно ядро"

Драйвера устройств

Системные вызовы

В зависимости от организации внутренних взаимодействий, ядра подразделяются на "микроядра" (microkernel) и монолитные ядра.

Системы с "микроядром" строятся по модульному принципу, имеют обособленное ядро, и механизм взаимодействия между драйверами устройств и процессами. По такому принципу строятся системы реального времени. Примерно так сделан QNX или HURD.

Монолитное ядро имеет более жесткую внутреннюю структуру. Все установленные драйвера жестко связываются между собой, обычно прямыми вызовами. По таким принципам строятся обыкновенные операционные системы типа Linux, FreeBSD.

Естественно, не все так четко, идеального монолитного или "микроядра" нет, наверное, ни в одной системе, просто системы приближаются к тому или иному типу ядра.

Один, отдельно взятый, процессор, в один момент времени, может исполнять только одну программу. Но к компьютерам предъявляются более широкие требования. Мало кто, в настоящее время, удовлетворился однозадачной операционной системой (к каким относился DOS, например). В связи с этим разработчики процессоров предусмотрели мультизадачные возможности.

Возможность эта заключается в том, что процессор выполняет какую-то одну программу (их еще называют процессами или задачами). Затем, по истечении некоторого времени (обычно это время меряется микросекундами), операционная система переключает процессор на другую программу. При этом все регистры текущей программы сохраняются. Это необходимо для того, чтобы через некоторое время вновь передать управление этой программе. Программа при этом не замечает каких либо изменений, для нее процесс переключения остается незаметен.

Для того чтобы программа не могла, каким либо образом, нарушить работоспособность системы или других программ, разработчики процессоров предусмотрели механизмы защиты.

Процессор предоставляет 4 "кольца защиты" (уровня привилегий), можно было бы использовать все, но это связано со сложностями взаимодействия программ разного уровня защиты. Поэтому в большинстве существующих систем используют два уровня.0 - привилегированный уровень (ядро) и 3 - непривилегированный (пользовательские программы).

Всем этим обеспечивается надежное функционирование системы и независимость программ друг от друга.

На "Собственно ядро" возлагаются функции менеджера памяти и процессов. Переключение процессов - это основной момент нормального функционирования системы. Драйвера не должны "тормозить", а тем более блокировать работу ядра. Windows - наглядный пример того, что этого нельзя допустить!

Теперь о драйверах. Драйвера - это специальные программы, обеспечивающие работу устройств компьютера. В существующих системах (во FreeBSD это точно есть, про Linux не уверен) предусматриваются механизмы прерывания работы драйверов по истечении какого-то времени. Правда, все зависит от того, как написан драйвер. Можно написать драйвер под FreeBSD или Linux, который полностью блокирует работу системы.

Избежать этого при двухуровневой защите не представляется возможным, поэтому драйвера надо будет тщательно программировать. В нашей работе драйверам мы уделим очень много внимания, поскольку от этого в основном зависит общая производительность системы.

Системные вызовы - это интерфейс между процессами и ядром (читайте-железом). Никаких других методов взаимодействия процессов с устройствами компьютера быть не должно. Системных вызовов достаточно много, на Linux их 190, на FreeBSD их порядка 350, причем большей частью они совпадают, соответствуя стандарту POSIX (стандарт, описывающий системные вызовы в UNIX). Разница заключается в передаче параметров, что легко будет предусмотреть. Естественно, мы не сможем сделать ядро, работающее одновременно на Linux и на FreeBSD, но по отдельности совместимость вполне реализуема.

Прикладным программам абсолютно безразлично, как системные вызовы реализуются в ядре. Это облегчает для нас обеспечение совместимости с существующими системами.

Ядро системы при распределении памяти оперирует 4-х килобайтными страницами.

Страницы могут использоваться самим ядром, для нужд драйверов (кэширование, например), или для процессов.

Программа или процесс состоит из следующих частей:

Сегмент кода. Может только выполняться, сама программа его не прочитать, не переписать не может! Использовать для этого сегмента swap не нужно, при необходимости код считывается прямо из файла;

Сегмент данных состоит из трех частей:

Константные данные, их тоже можно загружать из файла, так как они не меняются при работе программы;

Инициализированные данные. Участвует в процессе свопинга;

Не инициализированные данные. Так же участвует в свопинге;

Сегмент стека. Так же участвует в свопинге.

Но, обычно, системы делят сегмент данных на две части: инициализированные данные и не инициализированные данные.

Все сегменты разбиваются на страницы. Сегмент кода имеет постоянный размер. Сегмент данных может увеличиваться в сторону больших адресов. Сегмент стека, поскольку растет вниз, увеличивается в сторону уменьшения адресов. Страницы памяти для дополнительных данных или стека выделяются системой по мере необходимости.

Очень интересный момент:

При выполнении программы операционная система делает следующие действия:

Готовит для программы локальную таблицу дескрипторов;

Готовит для программы каталог страниц, все страницы помечаются как не присутствующие в памяти.

Все.

При передаче управления этой программе процессор генерирует исключение по отсутствию страницы, в котором нужная страница загружается из файла или инициализируется.

Еще один интересный момент:

Когда в системе загружается две или более одинаковых программы - нет необходимости для каждой из них выделять место для кодового сегмента, они спокойно могут использовать один код на всех.

Лабораторная работа № 1.

Загрузка операционной системы

Цель работы: Создать загрузочный диск для операционной системы.

Теоретические понятия.

Какие функции выполняет операционная система?

Операционная система должна выполняет следующие функции:

Обеспечивать загрузку пользовательских программ в оперативную память и их исполнение;

Обеспечивать управление памятью. В простейшем случае это указание единственной загруженной программе адреса, на котором заканчивается память, доступная для использования, и начинается память, занятая системой.

Обеспечивать работу с устройствами долговременной памяти, такими как магнитные диски, ленты, оптические диски и т.д.

Предоставлять более или менее стандартизированный доступ к различным периферийным устройствам.

Предоставлять некоторый пользовательский интерфейс.

Что такое загрузочный монитор и каково его предназначение?

Загрузочным монитором называется записанная в ПЗУ программа, которая находиться по тому адресу, по которому процессор передает управление в момент включения питания и производит первичную инициализацию процессора, тестирование памяти и обязательного периферийного оборудования, и, наконец, начинает загрузку системы

В чем отличие загрузочного монитора от консольного монитора?

В отличие от загрузочного монитора, консольный монитор позволяет просматривать содержимое памяти по заданному адресу, записывать туда данные, запускать какую-то область памяти как программу. На консольном мониторе можно даже писать программы, почти с таким же успехом, как на ассемблере.

Приведите определение первичного загрузчика.

Первичным загрузчиком (загрузочный сектор, boot-сектор) - это содержимое нулевого сектора нулевой дорожки диска, с которого производится загрузка. Первичный загрузчик, пользуясь сервисами загрузочного монитора, ищет на диске начало файловой системы своей родной ОС, находит в этой файловой системе файл с определенным именем, считывает его в память и передает этому файлу управление.

Что такое бутстрап?

Бустрап - это последовательное исполнение втягивающих друг друга загрузчиков возрастающей сложности.

Как происходит загрузка операционной системы?

Загрузка операционной системы может происходить по-разному. В простейшем случае, первичный загрузчик, пользуясь сервисами загрузочного монитора, ищет на диске начало файловой системы своей родной ОС, находит в этой файловой системе файл с определенным именем (ядро ОС), считывает его в память и передает этому файлу управление. Если файловая система имеет сложную структуру, то первичный загрузчик не в состоянии самостоятельно произвести загрузку ОС в связи с ограничениями его размера. Поэтому приходится считывать вторичный загрузчик,размер которого может быть намного больше. Из-за большего размера этот загрузчик намного умнее и в состоянии разобраться в структурах файловой системы. В некоторых случаях используются и третичные загрузчики.

Возможен вариант загрузки по сети: ПЗУ, установленное на сетевой карте, посылает в сеть пакет стандартного содержания, который содержит запрос к серверу удаленной загрузки. Этот сервер передает по сети вторичный загрузчик и т.д.

В чем преимущество модульного программирования ОС?

Преимущества модульного программирования ОС заключаются в возможности динамической сборки ядра операционной системы при загрузке, при которой дополнительные модули подгружаются уже после старта самого ядра. Это допускает подгрузку модулей по запросу, при этом подсистемы, нужные только иногда, могут не загрузиться вообще. Даже те модули, которые нужны всегда, могут проинициализироваться, только когда станут нужны, уменьшив тем самым время от начала загрузки до старта некоторых сервисов. Второе преимущество состоит в возможности реконфигурировать систему без перезагрузки, что особенно полезно для систем коллективного пользования. И, наконец, возможность выгрузки модулей ядра иногда (но не всегда, а лишь если поломка не мешает драйверу корректно освободить ресурсы) позволяет корректировать работу отдельных подсистем без перезагрузки всей ОС и пользовательских приложений.

Загрузчик boot. asm

kernel\_segequ1000h

org 100h

start:

jmp begin

begin:

mov ax,kernel\_seg

mov es,ax

mov bx,100h

call Write

exit:

call far kernel\_seg: 0100h

ret

procwrite

mov al,1h

mov dl,0

xor ch,ch

mov cl,2h

mov dh,0

mov ah,02h

int 13h

ret

endp Write

end start

Ядро kernel. asm

code segment para public 'code'

assume cs: code, ds: code, ss: code,es: code

org 100h

main proc

jmp EndData

string db " Operatsionnaea Sisteam studenta Makarova Anatoliea, grupa TI-065"

string\_len equ $-string

EndData:

xor ax,ax

mov bh,07h

xor cx,cx

mov dx,0184fh

mov ah,06h

int 10h

mov ah,13h

xor al,al

xor bh,bh

mov cx,string\_len

mov bl,06h

mov bp,offset string

mov dl,10

mov dh,10

int 10h

mov ah,00h

int 16h

main endp

code ends

end main

Вывод:

Я научился реализовывать загрузку операционной системы. Изучил основные моменты и правила вызова адресов памяти.

Лабораторная работа № 2.

Меню операционной системы

Цель работы: Создать меню для операционной системы и перехват нажатия клавиш.

Задание: Создать меню для операционной системы, каждое меню содержит свою клавишу, которую надо будет отлавливать программно, выполнить обновление экрана.

Новые добавления:

В процедуре PresKey происходит обработка нажатия клавиш, с последующим выводом сообщения на экран через процедуру print\_string.

|  |  |
| --- | --- |
| ; - -------------------------------------------  PresKeyPROC  Again:  mov ah,00h  int 16h  cmp al,0  jne Again  cmp ah,3Bh; Pres F1?  je F1  cmp ah,3Ch; Pres F2?  je F2  cmp ah,3Dh; Pres F3?  je F3  cmp ah,3Eh; Pres F4?  je F4  cmp ah,3Fh; Pres F5?  je F5  cmp ah,40h; Pres F6?  je F6  jmp Again  F1:  leasi, mes1  callprint\_string  jmp Again | F2:  leasi, mes2  callprint\_string  jmp Again  F3:  leasi, mes3  callprint\_string  jmp Again  F4:  call clear\_screen  jmp Again  F5:  leasi, mes5  callprint\_string  jmp Again  F6:  leasi, mes6  callprint\_string  hlt  ret  ENDPPresKey |

За вывод сообщения отвечает процедура print\_string. Печать происходит по символьное, похожее на teletype.

|  |  |
| --- | --- |
| print\_string proc near  push ax; store registers...  push si;  next\_char:  mov al, [si]  cmp al, 0  jz printed  inc si | mov ah, 0eh; teletype function.  int 10h  jmp next\_char  printed:  pop si; re-store registers...  pop ax;  ret  print\_string endp |

В ОС предусмотрено обновление экрана, необходима для корректного отображения меню. Предотвращает его смещение.

|  |  |
| --- | --- |
| clear\_screen proc near  push ax; store registers...  push ds;  push bx;  push cx;  push di;  mov ax, 40h  mov ds, ax; for getting screen parameters.  mov ah, 06h; scroll up function id.  mov al, 0; scroll all lines!  mov bh, 10011111b; attribute for new lines.  mov ch, 0; upper row.  mov cl, 0; upper col.  mov di, 84h; rows on screen - 1,mov dh, [di] ; lower row (byte).  mov di, 4ah; columns on screen,  mov dl, [di]  dec dl; lower col.  int 10h  ; set cursor position to top  ; of the screen:  mov bh, 0; current page.  mov dl, 0; col.  mov dh, 0; row.  mov ah, 02 | int 10h  pop di; re-store registers...  pop cx;  pop bx;  pop ds;  pop ax;  mov ah,13h  xor al,al  xor bx,bx  xor dx,dx  mov cx,menu\_len  mov bl,11  mov bp,offset menu  mov dl,10  mov dh,18h  int 10h  mov ah,13h  xor al,al  xor bh,bh  mov cx,string\_len  mov bl, 19h  mov bp,offset string  mov dl,10  mov dh,0  int 10h  ret  clear\_screen endp |

Код рабочей программы:

. model small

CSEG segment

assume cs: CSEG, ds: CSEG, es: CSEG, ss: CSEG

; - --------------------------------------------- -

Start:

jmp EndData

mes1DB13,10,' Pres <F1>', 0Dh,0Ah,0

mes2DB13,10,' Pres <F2>', 0Dh,0Ah,0

mes3DB13,10,' Pres <F3>', 0Dh,0Ah,0

mes5DB13,10,' Pres <F5>', 0Dh,0Ah,0

mes6DB13,10,' Pres <F6>', 0Dh,0Ah,0

string db " Operatsionnaea Sisteam studenta Macarova Anatoliea, grupa TI-065"

string\_len equ $-string

menu DB"F1 Help F2 Command F3 Calculator F4 ClearScreen F5 About F6 Exit"

menu\_len equ $-menu

EndData:

mov ax,cs

mov es,ax

mov ds,ax

xor di,di

xor si,si

xor dx,dx

xor bx,bx

xor cx,cx

mov ss,ax

mov sp,0FFFEh

call clear\_screen

call PresKey

Exit:

mov ah,00h

int 16h

hlt

; - -----------------------------------------------

PresKeyPROC

Again:

mov ah,00h

int 16h

cmp al,0

jne Again

cmp ah,3Bh; Pres F1?

je F1

cmp ah,3Ch; Pres F2?

je F2

cmp ah,3Dh; Pres F3?

je F3

cmp ah,3Eh; Pres F4?

je F4

cmp ah,3Fh; Pres F5?

je F5

cmp ah,40h; Pres F6?

je F6

jmp Again

F1:

leasi, mes1

callprint\_string

jmp Again

F2:

leasi, mes2

callprint\_string

jmp Again

F3:

leasi, mes3

callprint\_string

jmp Again

F4:

call clear\_screen

jmp Again

F5:

leasi, mes5

callprint\_string

jmp Again

F6:

leasi, mes6

callprint\_string

hlt

ret

ENDPPresKey

; print a null terminated string at current cursor position,

; string address: ds: si

print\_string proc near

push ax; store registers...

push si;

next\_char:

mov al, [si]

cmp al, 0

jz printed

inc si

mov ah, 0eh; teletype function.

int 10h

jmp next\_char

printed:

pop si; re-store registers...

pop ax;

ret

print\_string endp

; clear the screen by scrolling entire screen window,

; and set cursor position on top.

; default attribute is set to white on blue.

clear\_screen proc near

push ax; store registers...

push ds;

push bx;

push cx;

push di;

mov ax, 40h

mov ds, ax; for getting screen parameters.

mov ah, 06h; scroll up function id.

mov al, 0; scroll all lines!

mov bh, 10011111b; attribute for new lines.

mov ch, 0; upper row.

mov cl, 0; upper col.

mov di, 84h; rows on screen - 1,mov dh, [di] ; lower row (byte).

mov di, 4ah; columns on screen,

mov dl, [di]

dec dl; lower col.

int 10h

; set cursor position to top

; of the screen:

mov bh, 0; current page.

mov dl, 0; col.

mov dh, 0; row.

mov ah, 02

int 10h

pop di; re-store registers...

pop cx;

pop bx;

pop ds;

pop ax;

mov ah,13h

xor al,al

xor bx,bx

xor dx,dx

mov cx,menu\_len

mov bl,11

mov bp,offset menu

mov dl,10

mov dh,18h

int 10h

mov ah,13h

xor al,al

xor bh,bh

mov cx,string\_len

mov bl, 19h

mov bp,offset string

mov dl,10

mov dh,0

int 10h

ret

clear\_screen endp

CSEG ends

end Start

Выводы:

Данное меню выполнено на простом выводе на экран teletype. При использование меню происходит постоянное обновление, предотвращая сдвига верхней и нижней строки. Каждая клавиша выводит на экран сообщение о нажатой клавише. Строки меню выводятся по определенным координатам и окрашены особенным цветом.

Лабораторная работа № 3.

Окна операционной системы, реализация меню

Цель работы: Создать окна для операционной системы и перехват нажатия клавиш для меню и их реализация.

Задание: Создать окна для операционной системы, для примера окно About, так же произвести перезагрузку компьютера, задействуя кнопку F6.

Новые добавления:

В Data segment определяю значение, которые будут использованы для отображения меню: Нижнее главное меню и верхнее название ОС. Так же и для окна About, которое уже составлено в этом значение из символов. Данное окно и меню будет прорисовываться посимвольно с помощью процедуры print\_string.

string db " Operatsionnaea Sistema studenta Macarova Anatoliea, grupa TI-065"

string\_len equ $-string

menuText DB"F1 Save F2 Read F3 New Write F4 Close "

menuText\_len equ $-menuText

menu DB"F1 Bloknot F2 Change PAss F3 Convertor F4 ClearScreen F5 About F6 Exit"

menu\_len equ $-menu

About DB 13,10

db ' +------------------------------------------------+', 0Dh,0Ah

db ' |XXXXXXXXXXXXXXXX About Makar OS XXXXXXXXXXXXXXXX|', 0Dh,0Ah

db ' +------------------------------------------------|', 0Dh,0Ah

db ' | |', 0Dh,0Ah

db ' | +----------------------------------------+ |', 0Dh,0Ah

db ' | | | |', 0Dh,0Ah

db ' | | @@@@ Makar OS A 1.5 | |', 0Dh,0Ah

db ' | | @@@@ | |', 0Dh,0Ah

db ' | | Copiright (C) 2008 | |', 0Dh,0Ah

db ' | | Author Macarov Anatoli TI-065 | |', 0Dh,0Ah

db ' | | All Rights Reserved | |', 0Dh,0Ah

db ' | +----------------------------------------+ |', 0Dh,0Ah

db ' | |', 0Dh,0Ah

db ' | +-----------+ |', 0Dh,0Ah

db ' | |####OK#####| |', 0Dh,0Ah

db ' | +-----------+ |', 0Dh,0Ah

db ' +------------------------------------------------+', 0Dh,0Ah,0

About\_len equ $-About

Процедура print\_string обеспечивает посимвольный вывод. Передача в регистр si данные и после вызвать процедуру, вы получите результат.

print\_string proc near

push ax; store registers...

push si;

next\_char:

mov al, [si]

cmp al, 0

jz printed

inc si

mov ah, 0eh; teletype function.

int 10h

jmp next\_char

printed:

pop si; re-store registers...

pop ax;

ret

print\_string endp

Данный код программы реализует перезагрузку ОС, для активации необходимо нажать F6. После этого последует сообщение о том, что надо вытащить флоппи дискету и нажать любую кнопку и тогда произойдет перезагрузка.

; +++ 'quit', 'exit', 'reboot' +++

reboot\_command:

call clear\_screen

leasi, Please

callprint\_string

mov ax, 0; wait for any key... .

int 16h

; store magic value at 0040h: 0072h:

; 0000h - cold boot.

; 1234h - warm boot.

; mov ax, 0040h

; mov ds, ax

; mov w. [0072h], 0000h; cold boot.

; jmp 0ffffh: 0000h; reboot!

int 19h

Выводы:

Выполнив данную лабораторную работу, я изучил несколько способов перезагрузки ОС. Познал азы формирования окон на assembler'е. Узнал какие используются прерывания для BIOS.

Лабораторная работа № 4.

Конвертор из символа в ASCII код

Цель работы: Дополнить меню ОС.

Задание: Создать конвертор, обеспечивающий получить из символа ASCII код, задействуя кнопку F3.

Новые добавления:

Алгоритм конвертирования простой: пока частное не равно 0, делим его, делим и еще раз делим на 10d, запихивая остатки в стек. Потом - извлекаем из стека. Вот и вся конвертация из HEX в BIN. Это если в двух словах.

А если подробно, то вот что получается:

1 - подготавливаем делимое. Как уже говорилось, оно у нас задается неявно - обязательно через AX. А параметр у нас - через DX процедуре передается. Вот и перемещаем.

2 - это, собственно, делитель.

3 - очищаем CX. Он у нас будет в качестве счетчика.

4 - очищаем DX. Если не очистим, то мы не 1234h какое-нибудь на 10 делить будем, а 12341234h. Первое 1234 нам надо - очищать.

5 - делим. Частное - в AX, остаток - в DX.

6 - заносим остаток DX в стек.

7 - CX=CX+1. Это мы считаем сколько раз заносили остаток в стек, по кругу (прыжок на метку non\_zero), пока AX не равно 0 (8). То есть делим, делим AX, пока он не окажется таким, что делить, собственно, нечего.

Деление закончено, число, которое мы поделили AX до его полного обнуления, хранится в CX.

Дальше все просто. Нам нужно такое же количество раз CX извлечь значение DX из стека. И это будет "HEX", переведенный в DEC. (Оно же: число в двоичном коде, разобранное на последовательность десятичных цифр).

Цикл, в теле которого извлечь цифру (9) и напечатать цифру (10). Столько же раз, сколько мы и делили наше исходное шестнадцатеричное число.

write\_decimal proc

push ax

push cx

push dx

push bx

mov ax,dx; (1)

mov bx,10d; (2)

xor cx,cx; (3)

non\_zero:

xor dx,dx; (4)

div bx; (5)

push dx; (6)

inc cx; (7)

cmp ax,0; (8)

jne non\_zero

write\_digit\_loop:

pop dx; (9)

call write\_hex\_digit; (10)

loop write\_digit\_loop

pop bx

pop dx

pop cx

pop ax

ret

write\_decimal endp

WRITE\_HEX\_DIGIT proc

push DX

xor dh, dh

cmp DL,0Ah

jae HEX\_LETTER

add DL,30h

JMP WRITE\_DIGIT

HEX\_LETTER:

add DL,37h

WRITE\_DIGIT:

call WRITE\_CHAR

pop DX

ret

WRITE\_HEX\_DIGIT endp

WRITE\_CHAR proc

push AX

push BX

push CX

mov AH,9

xor BH,BH

mov BL,00000111b

mov CX,1

mov AL,DL

int 10h

call CURSOR\_RIGHT

pop CX

pop BX

pop AX

ret

WRITE\_CHAR endp

CURSOR\_RIGHT proc

push DX

call CURSOR\_READ

inc DL

call CURSOR\_SET

pop DX

ret

CURSOR\_RIGHT endp

CURSOR\_READ proc

push AX

push BX

push CX

mov AH,3

xor BH,BH

int 10h

pop CX

pop BX

pop AX

ret

CURSOR\_READ endp

CURSOR\_SET proc

push AX

push BX

push CX

mov AH,2

xor BH,BH

int 10h

pop CX

pop BX

pop AX

ret

CURSOR\_SET endp

Выводы:

Выполнив данную лабораторную работу, я изучил алгоритм преобразования шестнадцатеричного числа в десятичное. Так как символ в регистре хранится в HEX формате.

Лабораторная работа № 5.

Динамический пароль для входа в ОС

Цель работы: Дополнить меню ОС.

Задание: Создать паролирование ОС, задействуя кнопку F2.

Новые добавления:

В меню программы есть возможность изменить текущий пароль, нажав F2. За это действие отвечает процедура changepass, которое сохраняет введенную строку в временный буфер и после чего с помощью процедуры writesec записываю в сектор.

writ db 13,10,'Write sector', 0Dh,0Ah,0

sect db 1; sector number (1. .18).

cyld db 10; cylinder number (0. .79).

head db 0; head number (0. .1).

drive db 0; drive number (0. .3); A: =0, B: =1...

BUFFERDB 512 dup (0); prosto izmenili razmer do 512

Passworddb512 dup (0)

changepass proc

push ax

push bx

push cx

push si

call clear\_screen

lea si,PassMsg

call print\_string

xor bx,bx

lpss:

mov ah,00h

int16h

cmpal, 13

jelConv1

mov [Password + bx], al

movah, 0eh

int10h

incbx

jmplpss

lConv1:

call writesec

call readsec; scitivau dlea proverki

lea si,BUFFER

call print\_string

pop si

pop cx

pop bx

pop ax

ret

changepass ENDP

Процедура writesec обеспечивает запись данных в 10 цилиндр, в 1 сектор, с 0 головки. Заранее заполненный буфер Password, будет помещен в регистр bx для записи.

writesec proc; zapisivau dannie

push ax

push bx

push cx

push dx

lea si,writ

call print\_string

wr: mov ah, 03h

mov al, 1; write 1 sector (512 bytes).

mov cl, sect; sector (1. .18)

mov ch, cyld; cylinder (0. .79)

mov dh, head; head (0. .1)

mov dl, drive; always 0 (A:)

mov bx, offset Password

mov [Password + 6],0

int 13h

jc er

jmp e2e

er: lea dx, e2

mov ah, 9

int 21h

jmp e2e

e2 db " i/o error... ",0Dh,0Ah,'$'

e2e:

pop dx

pop cx

pop bx

pop ax

ret

ENDP writesec

Процедура readsec обеспечивает считывание данных из прежних секторов в буфер BUFFER. Дальше этот буфер можно использовать в сравнение с веденным паролем или просто вывести на экран.

readsec proc; scitivau dannie

push ax

push bx

push cx

push dx

mov ah, 02h

mov al, 1; write 1 sector (512 bytes).

mov cl, sect; sector (1. .18)

mov ch, cyld; cylinder (0. .79)

mov dh, head; head (0. .1)

mov dl, drive; always 0 (A:)

mov bx, offset BUFFER

int 13h

mov [BUFFER+6],0

pop dx

pop cx

pop bx

pop ax

ret

ENDP readsec

Процедура EnterPass проводит сравнение веденного пароля с текущем зарегистрированным паролем, что записан в сектор. Сравнение происходит с помощью команды repe cmpsb. При вводе пароля предусмотрено скрывание символов под звездочкой.

EnterPassPROC

push ax

push bx

push cx

push si

xorbx, bx

lPass:

mov ah,00h; vvoju paroli

int16h

cmpal, 13

jelCompare

mov [Password + bx], al

movah, 0eh

moval, '\*'

int10h

incbx

jmplPass

lCompare:

call readsec

leasi, Password

leadi, BUFFER

cld

movcx, 6

repecmpsb; sravnenie

jnelErr

jmplNext

lErr:; owibka

lea si, ENTERR

call print\_string

leasi, ErrPassMsg

callprint\_string

call Exit

lNext:; Paroli OK

lea si, ENTERR

call print\_string

leasi, OkMsg

callprint\_string

pop si

pop cx

pop bx

pop ax

; call PresKey

ret

ENDPEnterPass

; - -----------------------------------------------

Выводы:

Выполнив данную лабораторную работу, я изучил команды прерывания ввода/вывода биоса. Научился записывать на флоппи диск в определенный цилиндр и сектор. Получил опыт в сравнение двух строк.

Лабораторная работа № 6.

Блокнот для ОС

Цель работы: Блокнот для ОС.

Задание: Создать блокнот для записи данных на флоппи диск для последующего прочтения, задействуя кнопку F1.

Новые добавления:

В данной ОС предусмотрен блокнот, он обеспечивает запись данных на флоппи диск. Для активации блокнота вам надо нажать F1. После этого последует меню самого блокнота в котором вы можете прочитать текущие данные, вести данные для записи, выйти. Запись и чтение данных происходит так же как и в 5 лабораторной работе с помощью команд биоса, только здесь задействован 11 цилиндр и 1 сектор. Данных рассчитано на 512 байт.д.анная процедура редактировать данные не может.

; - ----------------------------------------------

EditText proc

push ax

push bx

push cx

push dx

mov ah,13h

xor al,al

xor bx,bx

xor dx,dx

mov cx,menuText\_len

mov bl,11

mov bp,offset menuText

mov dl,8

mov dh,18h

int 10h

mov ah,02h

mov bh,0

mov dh,2

mov dl,1

int 10h

AgainTe:

mov ah,00h

int 16h

cmp al,0

jne AgainTe

cmp ah,3Ch; Pres F2?

je F2t

cmp ah,3Dh; Pres F3?

je F3t

cmp ah,3Eh; Pres F4?

je F4t

jmp AgainTe

F2t:

jmp readtext

F3t:

call clear\_screen

jmp writetext

F4t:

call clear\_screen

jmp the\_end

writetext:

mov ah,13h

xor al,al

xor bx,bx

xor dx,dx

mov cx,menuText\_len

mov bl,11

mov bp,offset menuText

mov dl,8

mov dh,18h

int 10h

mov ah,02h

mov bh,0

mov dh,2

mov dl,1

int 10h

xor bx,bx

lp:

mov ah,00h

int16h

cmp ah,3Bh

jelC

cmpah,3Eh

jeF4t

mov [Editor + bx], al

movah, 0eh

int10h

incbx

jmplp

lC:

mov ah, 03h

mov al, 1; write 1 sector (512 bytes).

mov cl, sect; sector (1. .18)

mov ch, 11; cylinder (0. .79)

mov dh, head; head (0. .1)

mov dl, drive; always 0 (A:)

mov bx, offset Editor

int 13h

jmp the\_end

readtext:

mov ah, 02h

mov al, 1; write 1 sector (512 bytes).

mov cl, sect; sector (1. .18)

mov ch, 11; cylinder (0. .79)

mov dh, head; head (0. .1)

mov dl, drive; always 0 (A:)

mov bx, offset BUFFER

int 13h

mov [BUFFER+512],0

call clear\_screen

lea si,BUFFER

call print\_string

mov ah,13h

xor al,al

xor bx,bx

xor dx,dx

mov cx,menuText\_len

mov bl,11

mov bp,offset menuText

mov dl,8

mov dh,18h

int 10h

mov ah,02h

mov bh,0

mov dh,2

mov dl,1

int 10h

jmp AgainTe

the\_end:

pop dx

pop cx

pop bx

pop ax

ret

endp EditText

Код программы ОС:

. model small

CSEG segment

assume cs: CSEG, ds: CSEG, es: CSEG, ss: CSEG

; - --------------------------------------------- -

Start:

jmp EndData

mes1DB13,10,' Pres <F1>', 0Dh,0Ah,0

mes2DB13,10,'PASSS is... ', 0Dh,0Ah,0

writ db 13,10,'Write sector', 0Dh,0Ah,0

sect db 1; sector number (1. .18).

cyld db 10; cylinder number (0. .79).

head db 0; head number (0. .1).

drive db 0; drive number (0. .3); A: =0, B: =1...

BUFFERDB 512 dup (0); prosto izmenili razmer do 512

Passworddb512 dup (0)

EditorDB512 dup (0)

PassMsgdb13,10,'Enter password: ', 0Dh,0Ah,0

OkMsgdb13,10,' - > OK', 0Dh,0Ah,0

ErrPassMsg db13,10,'Password incorrect', 0Dh,0Ah,0

ConvEnterSy db13,10,' Convertor symbol in ASCII v1.0', 0Dh,0Ah

db13,10,'Enter symbol: ',0

ConResultAdb' Result ASCII: ',0

string db " Operatsionnaea Sistema studenta Macarova Anatoliea, grupa TI-065"

string\_len equ $-string

menuText DB"F1 Save F2 Read F3 New Write F4 Close "

menuText\_len equ $-menuText

menu DB"F1 Bloknot F2 Change PAss F3 Convertor F4 ClearScreen F5 About F6 Exit"

menu\_len equ $-menu

About DB 13,10

db ' +------------------------------------------------+', 0Dh,0Ah

db ' |XXXXXXXXXXXXXXXX About Makar OS XXXXXXXXXXXXXXXX|', 0Dh,0Ah

db ' +------------------------------------------------|', 0Dh,0Ah

db ' | |', 0Dh,0Ah

db ' | +----------------------------------------+ |', 0Dh,0Ah

db ' | | | |', 0Dh,0Ah

db ' | | @@@@ Makar OS A 1.5 | |', 0Dh,0Ah

db ' | | @@@@ | |', 0Dh,0Ah

db ' | | Copiright (C) 2008 | |', 0Dh,0Ah

db ' | | Author Macarov Anatoli TI-065 | |', 0Dh,0Ah

db ' | | All Rights Reserved | |', 0Dh,0Ah

db ' | +----------------------------------------+ |', 0Dh,0Ah

db ' | |', 0Dh,0Ah

db ' | +-----------+ |', 0Dh,0Ah

db ' | |####OK#####| |', 0Dh,0Ah

db ' | +-----------+ |', 0Dh,0Ah

db ' +------------------------------------------------+', 0Dh,0Ah,0

About\_len equ $-About

Please db 13,10," please eject any floppy disks "

db 13,10," and press any key to reboot... ",0

ENTERR db 13,10,"",0

EndData:

mov ax,cs

mov es,ax

mov ds,ax

xor di,di

xor si,si

xor dx,dx

xor bx,bx

xor cx,cx

mov ss,ax

mov sp,0FFFEh

; blinking disabled for compatibility with dos/bios,

; emulator and windows prompt never blink.

mov ax, 1003h

mov bx, 0; disable blinking.

int 10h

leasi, PassMsg

callprint\_string

callEnterPass

call clear\_screen

call PresKey

Exit:

mov ah,00h

int 16h

hlt

; - -----------------------------------------------

PresKeyPROC

Again:

mov ah,00h

int 16h

cmp al,0

jne Again

cmp ah,3Bh; Pres F1?

je F1

cmp ah,3Ch; Pres F2?

je F2

cmp ah,3Dh; Pres F3?

je F3

cmp ah,3Eh; Pres F4?

je F4

cmp ah,3Fh; Pres F5?

je F5

cmp ah,40h; Pres F6?

je F6

jmp Again

F1:

call EditText

jmp Again

F2:

call changepass

jmp Again

F3:

callconvert

jmp Again

F4:

call clear\_screen

jmp Again

F5:

call clear\_screen

lea si, About

call print\_string

jmp Again

F6:

callreboot\_command

hlt

ret

ENDPPresKey

; - --------------------------------------------- -

; +++ 'quit', 'exit', 'reboot' +++

reboot\_command:

call clear\_screen

leasi, Please

callprint\_string

mov ax, 0; wait for any key... .

int 16h

; store magic value at 0040h: 0072h:

; 0000h - cold boot.

; 1234h - warm boot.

; mov ax, 0040h

; mov ds, ax

; mov w. [0072h], 0000h; cold boot.

; jmp 0ffffh: 0000h; reboot!

int 19h

; ++++++++++++++++++++++++++

; - -----------------------------------------------

; print a null terminated string at current cursor position,

; string address: ds: si

print\_string proc near

push ax; store registers...

push si;

next\_char:

mov al, [si]

cmp al, 0

jz printed

inc si

mov ah, 0eh; teletype function.

int 10h

jmp next\_char

printed:

pop si; re-store registers...

pop ax;

ret

print\_string endp

; - ----------------------------------------------

EditText proc

push ax

push bx

push cx

push dx

mov ah,13h

xor al,al

xor bx,bx

xor dx,dx

mov cx,menuText\_len

mov bl,11

mov bp,offset menuText

mov dl,8

mov dh,18h

int 10h

mov ah,02h

mov bh,0

mov dh,2

mov dl,1

int 10h

AgainTe:

mov ah,00h

int 16h

cmp al,0

jne AgainTe

cmp ah,3Ch; Pres F2?

je F2t

cmp ah,3Dh; Pres F3?

je F3t

cmp ah,3Eh; Pres F4?

je F4t

jmp AgainTe

F2t:

jmp readtext

F3t:

call clear\_screen

jmp writetext

F4t:

call clear\_screen

jmp the\_end

writetext:

mov ah,13h

xor al,al

xor bx,bx

xor dx,dx

mov cx,menuText\_len

mov bl,11

mov bp,offset menuText

mov dl,8

mov dh,18h

int 10h

mov ah,02h

mov bh,0

mov dh,2

mov dl,1

int 10h

xor bx,bx

lp:

mov ah,00h

int16h

cmp ah,3Bh

jelC

cmpah,3Eh

jeF4t

mov [Editor + bx], al

movah, 0eh

int10h

incbx

jmplp

lC:

mov ah, 03h

mov al, 1; write 1 sector (512 bytes).

mov cl, sect; sector (1. .18)

mov ch, 11; cylinder (0. .79)

mov dh, head; head (0. .1)

mov dl, drive; always 0 (A:)

mov bx, offset Editor

int 13h

jmp the\_end

readtext:

mov ah, 02h

mov al, 1; write 1 sector (512 bytes).

mov cl, sect; sector (1. .18)

mov ch, 11; cylinder (0. .79)

mov dh, head; head (0. .1)

mov dl, drive; always 0 (A:)

mov bx, offset BUFFER

int 13h

mov [BUFFER+512],0

call clear\_screen

lea si,BUFFER

call print\_string

mov ah,13h

xor al,al

xor bx,bx

xor dx,dx

mov cx,menuText\_len

mov bl,11

mov bp,offset menuText

mov dl,8

mov dh,18h

int 10h

mov ah,02h

mov bh,0

mov dh,2

mov dl,1

int 10h

jmp AgainTe

the\_end:

pop dx

pop cx

pop bx

pop ax

ret

endp EditText

; - -----------------------------------------------

changepass proc

push ax

push bx

push cx

push si

call clear\_screen

lea si,PassMsg

call print\_string

xor bx,bx

lpss:

mov ah,00h

int16h

cmpal, 13

jelConv1

mov [Password + bx], al

movah, 0eh

int10h

incbx

jmplpss

lConv1:

call writesec

call readsec; scitivau dlea proverki

lea si,BUFFER

call print\_string

pop si

pop cx

pop bx

pop ax

ret

changepass ENDP

; - -----------------------------------------------

writesec proc; zapisivau dannie

push ax

push bx

push cx

push dx

lea si,writ

call print\_string

wr: mov ah, 03h

mov al, 1; write 1 sector (512 bytes).

mov cl, sect; sector (1. .18)

mov ch, cyld; cylinder (0. .79)

mov dh, head; head (0. .1)

mov dl, drive; always 0 (A:)

mov bx, offset Password

mov [Password + 6],0; prosto dobavili v konec 0 - dlea correctnogo vivoda na ekran i sravnenia parolei

int 13h

jc er

jmp e2e

er: lea dx, e2

mov ah, 9

int 21h

jmp e2e

e2 db " i/o error... ",0Dh,0Ah,'$'

e2e:

pop dx

pop cx

pop bx

pop ax

ret

ENDP writesec

; - -----------------------------------------------

readsec proc; scitivau dannie

push ax

push bx

push cx

push dx

mov ah, 02h

mov al, 1; write 1 sector (512 bytes).

mov cl, sect; sector (1. .18)

mov ch, cyld; cylinder (0. .79)

mov dh, head; head (0. .1)

mov dl, drive; always 0 (A:)

mov bx, offset BUFFER

int 13h

mov [BUFFER+6],0; prosto dobavili v konec 0 - dlea correctnogo vivoda na ekran i sravnenia parolei

pop dx

pop cx

pop bx

pop ax

ret

ENDP readsec

; - -----------------------------------------------

EnterPassPROC

push ax

push bx

push cx

push si

xorbx, bx

lPass:

mov ah,00h; vvoju paroli

int16h

cmpal, 13

jelCompare

mov [Password + bx], al

movah, 0eh

moval, '\*'

int10h

incbx

jmplPass

lCompare:

call readsec

leasi, Password

leadi, BUFFER

cld

movcx, 6

repecmpsb; sravnenie

jnelErr

jmplNext

lErr:; owibka

lea si, ENTERR

call print\_string

leasi, ErrPassMsg

callprint\_string

call Exit

lNext:; Paroli OK

lea si, ENTERR

call print\_string

leasi, OkMsg

callprint\_string

pop si

pop cx

pop bx

pop ax

; call PresKey

ret

ENDPEnterPass

; - -----------------------------------------------

; - -----------------------------------------------

convert PROC

push ax

push bx

push cx

push si

CAgain:

call clear\_screen

lea si,ConvEnterSy

call print\_string

xor bx,bx

mov ah,00h

int 16h

mov dl, al

mov ah, 0eh

int 10h

lConv:

lea si,ConResultA

call print\_string

call write\_decimal

CheEnter:

mov ah,00h

int 16h

cmp al, 13

je CAgain

cmp al, 1Bh

je ESCB

jmp CheEnter

ESCB:

call clear\_screen

call PresKey

pop si

pop cx

pop bx

pop ax

ret

ENDP convert

; - -----------------------------------------------

; - -----------------------------------------------

; clear the screen by scrolling entire screen window,

; and set cursor position on top.

; default attribute is set to white on blue.

clear\_screen proc near

push ax; store registers...

push ds;

push bx;

push cx;

push di;

mov ax, 40h

mov ds, ax; for getting screen parameters.

mov ah, 06h; scroll up function id.

mov al, 0; scroll all lines!

mov bh, 10011111b; attribute for new lines.

mov ch, 0; upper row.

mov cl, 0; upper col.

mov di, 84h; rows on screen - 1,mov dh, [di] ; lower row (byte).

mov di, 4ah; columns on screen,

mov dl, [di]

dec dl; lower col.

int 10h

; set cursor position to top

; of the screen:

mov bh, 0; current page.

mov dl, 0; col.

mov dh, 0; row.

mov ah, 02

int 10h

pop di; re-store registers...

pop cx;

pop bx;

pop ds;

pop ax;

mov ah,13h

xor al,al

xor bx,bx

xor dx,dx

mov cx,menu\_len

mov bl,11

mov bp,offset menu

mov dl,8

mov dh,18h

int 10h

mov ah,13h

xor al,al

xor bh,bh

mov cx,string\_len

mov bl, 19h

mov bp,offset string

mov dl,10

mov dh,0

int 10h

lea si, ENTERR

call print\_string

ret

clear\_screen endp

; - ------------------------------------------------------

; - -----------------------------------------------

write\_decimal proc

push ax

push cx

push dx

push bx

mov ax,dx; (1)

mov bx,10d; (2)

xor cx,cx; (3)

non\_zero:

xor dx,dx; (4)

div bx; (5)

push dx; (6)

inc cx; (7)

cmp ax,0; (8)

jne non\_zero

write\_digit\_loop:

pop dx; (9)

call write\_hex\_digit; (10)

loop write\_digit\_loop

pop bx

pop dx

pop cx

pop ax

ret

write\_decimal endp

WRITE\_HEX\_DIGIT proc

push DX

xor dh, dh

cmp DL,0Ah

jae HEX\_LETTER

add DL,30h

JMP WRITE\_DIGIT

HEX\_LETTER:

add DL,37h

WRITE\_DIGIT:

call WRITE\_CHAR

pop DX

ret

WRITE\_HEX\_DIGIT endp

WRITE\_CHAR proc

push AX

push BX

push CX

mov AH,9

xor BH,BH

mov BL,00000111b

mov CX,1

mov AL,DL

int 10h

call CURSOR\_RIGHT

pop CX

pop BX

pop AX

ret

WRITE\_CHAR endp

CURSOR\_RIGHT proc

push DX

call CURSOR\_READ

inc DL

call CURSOR\_SET

pop DX

ret

CURSOR\_RIGHT endp

CURSOR\_READ proc

push AX

push BX

push CX

mov AH,3

xor BH,BH

int 10h

pop CX

pop BX

pop AX

ret

CURSOR\_READ endp

CURSOR\_SET proc

push AX

push BX

push CX

mov AH,2

xor BH,BH

int 10h

pop CX

pop BX

pop AX

ret

CURSOR\_SET endp

; - ----------------------------------------------------- -

CSEG ends

end Start

Выводы:

Выполнив данную лабораторную работу, я изучил команды прерывания ввода/вывода биоса. Научился записывать на флоппи диск большое количество данных. Данный курс дал мне основные понятия по операционным системам.

## Заключение

Лабораторный практикум является обязательным компонентом обучения для всех инженерных дисциплин, применяемых в дистанционном обучении. Во время практикума студенты закрепляют теоретические знания практической работой с электронными схемами, учатся работать с контрольно-измерительной аппаратурой, приобретают исследовательские навыки. В связи с динамическим изменением элементной базы электроники, измерительной аппаратуры, электронный практикум должен своевременно обновляться и совершенствоваться. Дело это трудоемкое и достаточно дорогое, особенно в нынешних условиях.

При всех несомненных достоинствах существующего практикума имеется довольно много замечаний, которые в силу объективных и субъективных трудностей практической реализации не решены на сегодня:

Одно из таких направлений рассмотрено в данной работе - использование в лабораторном практикуме компьютерного моделирования на базе программного пакета Emu8086.

Достоинств в этом пакете больше, чем перечислено и о них еще будет говориться в процессе разработки лабораторного практикума. Однако то, что перечислено, позволило выбрать именно Emu8086, как наиболее подходящий для использования в лабораторном практикуме.

До настоящего времени, студенты выполняли лабораторные работы на Лабораторной установке, которая состоит из учебного микропроцессорного комплекта (УМК), выпускаемого рижским заводом "ВЭФ", набора модулей, подключаемых к ее системной шине и различных периферийных устройств УМК представляет собой учебную микро-ЭВМ, предназначенную для изучения программирования, пооектирования и настройки микропроцессорных устройств и систем, выполненных на МП KP580.

Как уже понятно, рижский завод "ВЭФ", уже находиться в другом государстве, морально и физически устарел (уже не выпускается). Микропроцессор МП KP580В уже давно снят с производства, не выпускается, весь мир давно перешел на микропроцессоры фирмы Intel, AMD, Motorolla и т.д.

Последние пять лет - лабораторный практикум выполнялся на эмуляторе МП086, но он был выполнен под операционную систему ДОС, еще современные операционные системы поддерживают эмуляцию ДОС, но следующие ОС перестают поддерживать ДОС.

Поэтому возникла потребность выбора программного продукта, работающего под Windows.

К тому же, необходимой литературы не имелось, нашел ссылки, но они только в общем рассказывают про работу эмулятора Emu8086. Это своего рода - первая работа про данный продукт. Может применяться для проведения лабораторных работ по дисциплине "Основы микропроцессорной техники", "Организация ЭВМ" и т.д.

Представленная работа - это реальный лабораторный практикум для основы работы с микропроцессором, с использованием методов виртуальной электронной лаборатории.

Основные результаты работы следующие:

Методически такая практическая работа прекрасно вписывается в программу курсов, принимаемых в обучении.

Программа имеет интуитивный интерфейс, достаточно проста и практически не требует специального времени на освоение.

Разработан лабораторный практикум, позволяющего быстро освоить работу с программой.

Настоящий лабораторный практикум не является окончательным и закрытым его всегда можно расширить и модифицировать. Для этого не нужны специальные навыки и знания (как например при попытках дополнить программы моделирующие лабораторные работы и написанные на языках программирования) интерфейс Emu8086 прост и выразителен.

Кроме того, тематика лабораторных работ (а возможно и курсовых) выполняемых в этой виртуальной лаборатории может быть очень широка.

Данная разработка демонстрирует лишь очень небольшую часть возможностей Emu8086.

Сравнение виртуального лабораторного практикума с традиционным оборудованием - лабораторными стендами. Итог этого обоснования следующий:

один компьютер способен заменить несколько стендов с разной тематикой работ;

использование компьютерного моделирования позволяет высвободить часть персонала занятого ранее ремонтом стендов;

качество обучения также повысится за счёт большей чем у стендов наглядности, за счёт того, что перестанут выходить из строя исследуемые и вспомогательные компоненты, и за счет приобретения студентами дополнительных навыков работы на компьютере.

## Список использованных источников

1. Алексенко А. Г, Шагурин И.И. “Микросхемотехника”. Москва, изд. “Радио и связь", 1982г.

2. Ан П. Сопряжение ПК с внешними устройствами, ДМК.

3. Балашов Е.П., Григорьев В.Л., Петров Г.А. Микро - и миниЭВМ. Л.: Энергоатомиздат, 1984.376 с.

4. Басманов А.С., Широков Ю.Ф. Микропроцессоры и однокристальные микроЭВМ: Номенклатура и функциональные возможности / Под ред.В.Г. Домрачева. М.: Энергоатомиздат, 1988.127 с.

5. Байцер Б. Архитектура вычислительных комплексов. М.: Мир, 1974. Т.1,2.

6. Васильев Н. П, Горовой В.Р. Микропроцессоры: Аппаратурно - программные средства отладки. /Под ред. Проф.Л.Н. Преснухина. - М.: Высшая школа, 1984.

7. Влах, Кишор, Сингхал. “Машинные методы анализа и проектирования электронных схем” Москва, изд. “Радио и связь", 1988г.

8. Водовозов В.М., Осипов В.О., Пожидаев А.К. Практическое введение в информационные системы / ГЭТУ. СПб, 1995.

9. Вуд А. Микропроцессоры в вопросах и ответах / Пер. с англ. М.: Энергоатомиздат, 1985.185с.

10. Гибсон Г., Лю Ю. - Ч. Аппаратные и программные средства микроЭВМ / Пер. с англ.В.Л. Григорьева, Под ред.В. В. Сташина. М.: Финансы и статистика, 1983.255с.

11. Гивоне Д, Россер Р. Микропроцессоры и микрокомпьютеры: Вводный курс / Пер. с англ. М.: Мир, 1983.463 с.

12. Гук М. Аппаратные средства IBM PC. Энциклопедия, Питер.

13. Дао Л. Программирование микропроцессора 8088. - 1988.

14. Дебновецкий С.В. “Основы автоматизированного проектирования электронных приборов” Киев, Вища школа, 1987г.

15. “Измерения параметров цифровых интегральных микросхем" (под ред. Эйдукаса Д.Ю., Орлова Б. В). Москва, “Радио и связь", 1982г.

16. Казагачев В.Н. Методические указания к лабораторной работе №1 " Правила перевода чисел из одной системы счисления в другую" по дисциплине "Основы микропроцессорной техники". АФ КазАТК, Актобе-2006, 40с.

17. Казагачев В.Н. Методические указания к лабораторной работе №2 " Проверка таблиц истинности базовых вентилей" по дисциплине "Основы микропроцессорной техники". АФ КазАТК, Актобе-2006, 22с.

18. Казагачев В.Н. Основы микропроцессорной техники. Учебное пособие. Актобе: - 2007г, 350с.

19. Карлов Н.В. "Вселенная образования: давно осознанная необходимость и склонность к учению". Электронный журнал "Исследовано в России": 2-62, 2001.

20. Карлащук В.И. Электронная лаборатория на IBM PC. Программа Electronics Workbench и ее применение. Издание 2-е, дополненное и переработанное. Издательство "Солон-Р", Москва, 2001 - 736с.

21. Корнеев В. В, Киселёв А.В. “Современные микропроцессоры" Москва, изд. “Нолидж”, 1998г.

22. Лазер И. М, Шубарев В.А. “Устойчивость цифровых микроэлектронных устройств" Москва, “Радио и связь", 1983г.

23. Лю Ю-Чжен, Гибсон Г. Микропроцессоры семейства 8086/8088. Архитектура, программирование и проектирование микрокомпьютерных систем. - 1987.

24. Лысиков Б.Г. “Арифметические и логические основы цифровых автоматов" Минск, “Вышэйшая школа", 1980г.

25. Микропроцессоры: В 3-х кн. / Под ред. Преснухина. М.: Высшая школа, 1986. Кн.1.495 с. Кн.2.383 с. Кн.3.351с.

26. МикроЭВМ / Пер. с англ., Под ред.А. Дирксена. М.: Энергоиздат, 1982.328 с.

27. Мячев А.А., Иванов В.В. Интерфейсы вычислительных систем на базе мини - и микроЭВМ / Под ред. Наумова Б.Н. М.: Радио и связь, 1986.248 с.

28. Майоров С.А., Кириллов В.В., Приблуда А.А. Введение в микроЭВМ. М.: Машиностроение, Ленингр. отд., 1988.303с.

29. Морисита И. Аппаратные средства микроЭВМ / Пер. с япон. М.: Мир, 1988.279с.

30. Мюллер Скотт. Модернизация и ремонт компьютера.14-издание.: Пер. с англ. - М.: Издательский дом "Вильямс", 2003. - 1184с. Ил. - Парал. тит. англ.

31. Новиков Ю.В., Скоробогатов П.К. Основы микропроцессорной техники. Интернет-университет информационных технологий - ИНТУИТ. ру. 2004г.

32. Новиков Ю.В. Кондратенко С.В. Локальные сети: архитектура, алгоритмы, проектирование, ЭКОМ.

33. Нестеров П.В. Микропроцессоры. Архитектура и ее оценка. М.: Высшая школа, 1984.104с.

34. Нефедов А. В, Савченко А.М., Феоктистов Ю.Ф. “Зарубежные интегральные микросхемы для электронной аппаратуры." Москва, Энергоатомиздат, 1989г.

35. Ногов Ю.Р. “Математические модели элементов интегральной электроники" Москва, “Современное радио", 1976г.

36. Интерфейсы систем обработки данных: Справочник / Под ред.А. А. Мячева. М.: Радио и связь, 1989.

37. Пухальский Г. И, Новосельцева Т.Я. “Цифровые устройства” Санкт-Петербург, изд. “Политехника”1996г.

38. Предко М. Справочник по PIC-микроконтроллерам, ДМК, Додэка.

39. Пономарёв С.М., Ховричева М.Л. "Особенности лабораторного эксперимента в преподавании естественнонаучных дисциплин". М.: МИФИ, Материалы // Конференция-выставка "Информационные технологии в образовании", ИТО-2000.

40. Ремизевич Т.В. Микроконтроллеры для встраиваемых приложений: от общих подходов - к семействам НС05 и НС08 фирмы Motorola, Додэка

41. Соучек Б. Микропроцессоры и микроЭВМ / Пер. с англ. Под ред.А.И. Петренко. М.: Сов. радио, 1979.517 с.

42. Сысоев В.В. “Структурные и алгоритмические модели автоматизированного проектирования производства изделий электронной техники" Воронеж, Воронежский технологический институт, 1993г.

43. Сборник задач по вычислительным машинам и программированию. Учебное пособие. М., "Статистика", 1975г. - 280 с, с ил.

44. Стецюк А.Е., Кейно М.Ю. Микропроцессорные системы управления: Методические указания на выполнение лабораторных работ.д.альневосточный государственный университет путей сообщения, Хабаровск - 1998

45. СТ РК 34.017-2005. Информационные технологии. Электронное издание. Электронное учебное издание

46. Токхайм Р. “Основы цифровой электроники" Москва, изд. “Мир”, 1988г.

47. Токхайм Р. Микропроцессоры: Курс и упражнения /Пер. с англ. Под ред. Грасевича. М.: Энергоатомиздат, 1987.338с.

48. Тавернье К. PIC-микроконтроллеры. Практика применения, ДМК

49. Чахмахсазян Е.А., Мозговой Г.П., “Математическое моделирование и макромоделирование биполярных элементов электронных схем” Москва, “Радио и связь", 1985г.

50. Уокерли Дж. Архитектура и программирование микроЭВМ: В 2-х кн. / Пер. с англ. М.: Мир, 1984. Кн.1.486 с. Кн.2.359 с.

51. Хосе М. Ангуло. Микропроцессоры: Архитектура, программирование и проектирование систем. Тбилиси: Ганатлеба, 1989.

52. Хоровиц П., Хилл У. Искусство схемотехники. М.: Мир, 1983. Т.2.590 с.

53. Хамакер К, З. Вранешич, С. Заки. Организация ЭВМ, 5-издание. - СПб.: Питер; Киев: Издательская группа BHV, 2003. - 848с.: ил.

54. Шило В.Л. “Популярные цифровые микросхемы” Москва, Металлургия, 1988г.

55. Эмулятор Emu8086. Справочная система.

56. Якимов О.П. “Моделирование режимов и оценка качества электронных приборов” Москва, “Радио и связь", 1989г.

57. Янсен Й. “Курс цифровой электроники" т.1 Москва, Мир, 1987г.

58. Яценков В.С. Микроконтроллеры Microchip. Практическое руководство, Горячая линия - Телеком