

МИНИСТЕРСТВО СЕЛЬСКОГО ХОЗЯЙСТВА РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего профессионального образования

«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ АГРАРНЫЙ УНИВЕРСИТЕТ»

А.В. Параскевов, А.Н. Бардак

МИКРОПРОЦЕССОРЫ

Учебное пособие

Краснодар
КубГАУ
2015

УДК 004.31 (076.5)

П-18

Рецензенты:

Франгиз Гильфанетдинович Хисамов – доктор технических наук, профессор, академик Российской академии естественных наук, ректор Кубанского института информзащиты;

Елена Витальевна Попова – доктор экономических наук, профессор, зав. кафедрой информационных систем КубГАУ.

Под редакцией заслуженного деятеля науки РФ, доктора технических наук, профессора, зав. кафедрой компьютерных технологий и систем Кубанского государственного аграрного университета - Валерия Ивановича Лойко.

Параскевов А.В.

П-18 «Микропроцессоры»: учеб. пособие / А.В. Параскевов, А.Н. Бардак – Краснодар, КубГАУ, 2015. - 160 с.

Учебное пособие по дисциплине «Микропроцессоры» предназначено для направления подготовки 09.03.02 – «Информационные системы и технологии», разработано на кафедре компьютерных технологий и систем (КТС) факультета прикладной информатики (ФПИ) КубГАУ. Оно соответствует требованиям государственного образовательного стандарта (ГОС) высшего образования по направлению подготовки 09.03.02 – «Информационные системы и технологии (уровень бакалавриата)», утвержденного 12.03.2015г. Минобрнауки РФ регистрационный № 219.

УДК 004.31 (076.5)

© Параскевов А.В.,
Бардак А.Н., 2015
© ФГБОУ ВПО «Кубанский
государственный аграрный
университет», 2015

Оглавление

<i>Предисловие.....</i>	<i>5</i>
<i><u>Лекционный курс.....</u></i>	<i>6</i>
<i>1. Архитектура микропроцессора.....</i>	<i>6</i>
<i>2. Особенности реализации микропроцессоров.....</i>	<i>12</i>
<i>3. Организация памяти.....</i>	<i>16</i>
<i>4. Жизненный цикл программы.....</i>	<i>19</i>
<i>5. Система команд микропроцессора.....</i>	<i>22</i>
<i>6. Команды обмена данными.....</i>	<i>26</i>
<i>7. Прерывания.....</i>	<i>32</i>
<i>8. Команды передачи управления.....</i>	<i>36</i>
<i>9. Цепочечные команды.....</i>	<i>39</i>
<i>10. Сложные структуры данных (часть 1).....</i>	<i>43</i>
<i>11. Сложные структуры данных (часть 2).....</i>	<i>47</i>
<i>12. Модульное программирование (часть 1).....</i>	<i>50</i>
<i>13. Модульное программирование (часть 2).....</i>	<i>54</i>
<i>14. Создание Windows – приложений на ассемблере.....</i>	<i>58</i>
<i>15. Защищенный режим работы микропроцессора.....</i>	<i>64</i>
<i>16. Обработка прерываний в защищенном режиме.....</i>	<i>68</i>
<i>17. MMX-технология микропроцессоров.....</i>	<i>70</i>
<i>18. Макросредства языка ассемблера.....</i>	<i>76</i>
<i>19. Архитектура сопроцессора (часть 1).....</i>	<i>80</i>
<i>20. Архитектура сопроцессора (часть 2).....</i>	<i>83</i>
<i><u>Лабораторный практикум.....</u></i>	<i>86</i>
<i>Лабораторная работа №1.....</i>	<i>87</i>
<i>Лабораторная работа №2.....</i>	<i>91</i>
<i>Лабораторная работа №3.....</i>	<i>96</i>

<i>Лабораторная работа №4.....</i>	<i>100</i>
<i>Лабораторная работа №5.....</i>	<i>106</i>
<i>Лабораторная работа №6.....</i>	<i>111</i>
<i>Лабораторная работа №7.....</i>	<i>114</i>
<i>Лабораторная работа №8.....</i>	<i>118</i>
<i>Лабораторная работа №9.....</i>	<i>121</i>
<i>Лабораторная работа №10.....</i>	<i>123</i>
<i>Курсовое проектирование</i>	<i>124</i>
<i>Тематика курсовых работ (примерная).....</i>	<i>125</i>
<i>Структура и объем пояснительной записки курсовой работы.....</i>	<i>128</i>
<i>Оформление курсовой работы.....</i>	<i>130</i>
<i>Критерии оценки курсовой работы.....</i>	<i>134</i>
<i>Фонд оценочных средств по дисциплине</i>	<i>135</i>
<i>Паспорт фонда оценочных средств... ..</i>	<i>136</i>
<i>Перечень вопросов к экзамену (примерный).....</i>	<i>136</i>
<i>Вопросы к тестовым заданиям (примерные).....</i>	<i>140</i>
<i>Глоссарий.....</i>	<i>154</i>
<i>Список использованных источников.....</i>	<i>159</i>

Предисловие

Несмотря на общее название, язык ассемблера для каждого типа компьютеров свой. Это касается и внешнего вида программ, написанных на ассемблере и идей, отражением которых этот язык является. Не случайно практически все компиляторы языков высокого уровня содержат средства связи своих модулей с модулями на ассемблере, либо поддерживают выход на ассемблерный уровень программирования. Есть нечто общее в базовой подготовке всех программистов, своего рода фундамент, - это знание принципов работы компьютера, его архитектуры и язык ассемблера, отражающего устройство компьютера. Без рассмотрения данных вопросов невозможно любое сколько-нибудь серьезное компьютерное образование.

Так как ассемблер является символическим представлением машинного языка, то он неразрывно связан с архитектурой микропроцессора. По мере внесения изменений в его архитектуру совершенствуется и язык ассемблера.

Процесс изучения дисциплины направлен на формирование следующих компетенций:

профессиональные (ПК):

- способностью проводить техническое проектирование (ПК-2);
- способностью проводить рабочее проектирование (ПК-3).

Ваши замечания и предложения отправляйте по адресу электронной почты **paraskevov.alexander@yandex.ru**.

ЛЕКЦИОННЫЙ КУРС

1. Архитектура ЭВМ, характеристики микропроцессоров

Архитектура ЭВМ – это абстрактное представление ЭВМ, которое отражает ее структурную, схемотехническую и логическую организацию. Понятие архитектуры ЭВМ является комплексным и включает в себя:

- структурную схему;
- средства и способы доступа к элементам структурной схемы;
- организацию и разрядность интерфейсов;
- набор и доступность регистров;
- набор машинных команд;
- организацию и способы адресации памяти;
- способы представления и форматы данных;
- обработку прерываний;
- форматы машинных команд.

Микропроцессор – это устройство, выполняющее обработку информации на персональных компьютерах, управляет вычислительным процессом, арифметическими и логическими операциями.

Модели процессоров включают следующие, совместно работающие устройства:

Устройство управления (УУ). Осуществляет координацию работы всех остальных устройств, выполняет функции управления устройствами, управляет вычислениями в компьютере.

Арифметико-логическое устройство (АЛУ). Так называется устройство для целочисленных операций. Арифметические операции, такие как сложение, умножение и деление, а также логические операции обрабатываются при помощи АЛУ. Эти операции составляют подавляющее большинство программного кода в большинстве программ. Все операции в АЛУ производятся в регистрах - специально отведенных ячейках АЛУ. В процессоре может быть несколько АЛУ. Каждое способно исполнять арифметические или логические операции независимо от других, что позволяет выполнять несколько операций одновременно. Арифметико-логическое устройство выполняет арифметические и логические действия. Логические операции делятся на две простые операции: «Да» и «Нет» («1» и «0»). Обычно эти два устройства выделяются чисто условно, конструктивно они не разделены.

AGU (Address Generation Unit) - устройство генерации адресов. Это устройство не менее важное, чем АЛУ, т.к. оно отвечает за корректную адресацию при загрузке или сохранении данных. Абсолютная адресация в программах используется только в редких исключениях. Как только берутся массивы данных, в программном коде используется косвенная адресация, заставляющая работать **AGU**.

Математический сопроцессор. Процессор может содержать несколько математических сопроцессоров. Каждый из них способен выполнять, по меньшей мере, одну операцию с плавающей точкой независимо от того, что делают другие АЛУ. Метод конвейерной обработки данных позволяет одному математическому сопроцессору выполнять несколько операций одновременно. Сопроцессор поддерживает высокоточные вычисления как целочисленные, так и с плавающей точкой и, кроме того, содержит набор полезных констант, ускоряющих вычисления.

Сопроцессор работает параллельно с центральным процессором, обеспечивая, таким образом, высокую производительность. Система выполняет команды сопроцессора в том порядке, в котором они появляются в потоке. Математический сопроцессор персонального компьютера **IBM PC** позволяет ему выполнять скоростные арифметические и логарифмические операции, а также тригонометрические функции с высокой точностью.

Дешифратор инструкций (команд). Анализирует инструкции в целях выделения операндов и адресов, по которым размещаются результаты. Затем следует сообщение другому независимому устройству о том, что необходимо сделать для выполнения инструкции. Дешифратор допускает выполнение нескольких инструкций одновременно для загрузки всех исполняющих устройств.

Кэш-память. Особая высокоскоростная память процессора. Кэш используется в качестве буфера для ускорения обмена данными между процессором и оперативной памятью, а также для хранения копий инструкций и данных, которые недавно использовались процессором. Значения из кэш-памяти извлекаются напрямую, без обращения к основной памяти. При изучении особенностей работы программ было обнаружено, что они обращаются к тем или иным областям памяти с различной частотой, а именно: ячейки памяти, к которым программа обращалась недавно, скорее всего, будут использованы вновь. Предположим, что микропроцессор способен хранить копии этих инструкций в своей локальной памяти. В этом случае процессор сможет каждый раз использовать копию этих инструкций на протяжении всего цикла. Доступ к памяти понадобится в самом начале. Для хранения этих инструкций необходим совсем небольшой объем памяти. Если инструкции в процессор поступают достаточно быстро, то

микропроцессор не будет тратить время на ожидание. Таким образом экономиться время на выполнение инструкций. Но для самых быстродействующих микропроцессоров этого недостаточно. Решение данной проблемы заключается в улучшении организации памяти. Память внутри микропроцессора может работать со скоростью самого процессора.

Кэш первого уровня (**L1 cache**). Кэш-память, находящаяся внутри процессора. Она быстрее всех остальных типов памяти, но меньше по объёму. Хранит совсем недавно

использованную информацию, которая может быть использована при выполнении коротких программных циклов.

Кэш второго уровня (**L2 cache**). Также находится внутри процессора. Информация, хранящаяся в ней, используется реже, чем информация, хранящаяся в кэш-памяти первого уровня, но зато по объёму памяти он больше. Также в настоящее время в процессорах используется кэш третьего уровня.

Основная память. Намного больше по объёму, чем кэш-память, и значительно менее быстродействующая.

Многоуровневая кэш-память позволяет снизить требования наиболее производительных микропроцессоров к быстродействию основной динамической памяти. Так, если сократить время доступа к основной памяти на 30%, то производительность хорошо сконструированной кэш-памяти повысится только на 10-15%. Кэш-память, как известно, может достаточно сильно влиять на производительность процессора в зависимости от типа выполняемых операций, однако ее увеличение вовсе не обязательно принесет увеличение общей производительности работы процессора. Все зависит от того, насколько приложение оптимизировано под данную структуру и использует кэш, а также от того, помещаются ли различные сегменты программы в кэш целиком или кусками.

Кэш-память не только повышает быстродействие микропроцессора при операции чтения из памяти, но в ней также могут храниться значения, записываемые процессором в основную память; записать эти значения можно будет позже, когда основная память будет не занята. Такая кэш-память называется «кэш с обратной записью» (**write back cache**). Её возможности и принципы работы заметно отличаются от характеристик кэша со сквозной записью (**write through cache**), который участвует только в операции чтения из памяти.

Шина - это канал пересылки данных, используемый совместно различными блоками системы. Шина может представлять собой набор проводящих линий в печатной плате, провода, припаянные к выводам разъемов, в которые вставляются печатные платы, либо плоский кабель. Информация передается по шине в виде групп битов. В состав шины

для каждого бита слова может быть предусмотрена отдельная линия (параллельная шина), или все биты слова могут последовательно во времени использовать одну линию (последовательная шина). К шине может быть подключено много приемных устройств - получателей. Обычно данные на шине предназначаются только для одного из них. Сочетание управляющих и адресных сигналов, определяет для кого именно. Управляющая логика возбуждает специальные стробирующие сигналы, чтобы указать получателю, когда ему следует принимать данные. Получатели и отправители могут быть однонаправленными (т.е. осуществлять только либо передачу, либо прием) и двунаправленными (осуществлять и то и другое). Однако самая быстрая процессорная шина не сильно поможет, если память не сможет доставлять данные с соответствующей скоростью.

Типы шин

Шина данных. Служит для пересылки данных между процессором и памятью или процессором и устройствами ввода-вывода. Эти данные могут представлять собой как команды микропроцессора, так и информацию, которую он посылает в порты ввода-вывода или принимает оттуда.

Шина адресов. Используется ЦП для выбора требуемой ячейки памяти или устройства ввода-вывода путем установки на шине конкретного адреса, соответствующего одной из ячеек памяти или одного из элементов ввода-вывода, входящих в систему.

Шина управления. По ней передаются управляющие сигналы, предназначенные памяти и устройствам ввода-вывода. Эти сигналы указывают направление передачи данных (в процессор или из него).

Регистры - это внутренняя память процессора. Представляют собой ряд специализированных дополнительных ячеек памяти, а также внутренние носители информации микропроцессора. Регистр является устройством временного хранения данных, числа или команды и используется с целью облегчения арифметических, логических и пересылочных операций. Над содержимым некоторых регистров специальные электронные схемы могут выполнять некоторые манипуляции. Например, «вырезать» отдельные части команды для последующего их использования или выполнять определенные арифметические операции над числами. Основным элементом регистра является электронная схема, называемая триггером, которая способна хранить одну двоичную цифру (разряд). Регистр представляет собой совокупность триггеров, связанных друг с другом определенным образом общей системой управления. Существует несколько типов регистров, отличающихся видом выполняемых операций.

Некоторые важные регистры имеют свои названия, например:

сумматор — регистр АЛУ, участвующий в выполнении каждой операции;

счетчик команд — регистр УУ, содержимое которого соответствует адресу очередной выполняемой команды; служит для автоматической выборки программы из последовательных ячеек памяти;

регистр команд — регистр УУ для хранения кода команды на период времени, необходимый для ее выполнения. Часть его разрядов используется для хранения кода операции, остальные — для хранения кодов адресов операндов.

Понятие содержит в себе практически всю необходимую информацию о компьютере. Все современные ЭВМ обладают как общими, так и индивидуальными свойствами и особенностями архитектуры. Индивидуальные свойства отличают лишь конкретную модель компьютера от остальных. Присутствие общих архитектурных свойств объясняется тем, что большинство типов существующих машин принадлежат к так называемой «фон-неймановской» архитектуре. К числу общих принципов и свойств следует отнести:

1. *Принцип хранимой программы.* Согласно ему код и данные находятся в одном адресном пространстве оперативной памяти.

2. *Принцип микропрограммирования.* Его суть заключается в том, что машинный язык все еще не является той конечной субстанцией, которая физически приводит в действие процессы в машине. В состав процессора входит блок микропрограммного управления, который для каждой машинной команды имеет набор действий-сигналов, которые необходимо сгенерировать для физического выполнения требуемой машинной команды.

3. *Линейное пространство памяти.* Это совокупность ячеек, которым последовательно присваиваются адреса (номера, начиная от 0).

4. *Последовательное выполнение программ.* Строго последовательно выбираются команды из памяти процессором.

5. *Безразличие к целевому назначению данных.* Для машины не имеет значения, какую логическую нагрузку несут обрабатываемые ей данные.

6. *Отсутствие, с точки зрения обработки, принципиальной разницы между данными и командами.* Данные и машинные команды находятся в одном адресном пространстве памяти в виде последовательности нулей и единиц. Процессор, исполняя содержимое последовательных ячеек памяти, пытается трактовать их как коды машинной команды, если это не так, то происходит аварийное завершение программы, содержащей некоторый фрагмент.

Конвейеризация вычислений – специальное устройство, конвейер, реализует такой метод обработки команд внутри микропроцессоров, что исполнение команды разбивается на несколько этапов. Первые конвейеры имели 5 этапов:

1. выборка команды из кэш-памяти или оперативной памяти;
2. декодирование команды;
3. генерация адреса (при ней определяются адреса операндов в памяти);
4. выполнение операции с помощью АЛУ;
5. запись результата (когда будет записан результат - зависит от конкретного алгоритма работы конкретной машинной команды).

2. Особенности реализации микропроцессоров

Исторически разделение процессоров на процессоры и микропроцессоры возникло в начале 70-х годов XX века, с началом производства больших интегральных схем. Собственно микропроцессор отличается от обычных процессоров тем, что располагается на одном кристалле БИС.

Если отслеживать поколения ЭВМ по элементной базе, можно увидеть, что с первого по третье поколение ЭВМ (на лампах, на транзисторах и на малых интегральных схемах) элементы ЭВМ (АЛУ, регистры) занимали много места (по площади платы), и поэтому объединять их в блоки по функциональному признаку не было смысла. Даже в ЭВМ III-го поколения существовали отдельные микросхемы сумматоров, регистров, дешифраторов, и часто было проще заменить отдельную неисправную микросхему, чем весь процессорный блок.

Ситуация стала меняться в конце 60-х годов XX века, когда, во-первых, уменьшились размеры элементов (так появились БИС), а во-вторых, увеличилась надежность и выход готовых микросхем. Тогда стали проводиться исследования по построению процессора на одной микросхеме. От слов «микросхема» («**MICROchip**») и процессор («**PROCESSOR**») возникло понятие «микропроцессор» («**microprocessor**»), а вовсе не от «миниатюрный процессор». Тогда микропроцессор считали чудачеством ученых и инженеров. В конце концов в 1968 году группа ученых во главе с Крэгом Барретом (**Crag Barrett**) выделилась со скандалом из Американского исследовательского центра, занимающегося разработкой технологии интегральных микросхем, в отдельную фирму – **Intel**. Эта группа стала создавать микропроцессоры. В 1969 году из этого же центра выделилась фирма **AMD**.

Объединение элементов процессора на одном кристалле повысила не только надежность, но и скорость его работы. После этого скорость работы процессоров увеличивали повышением его разрядности и частоты, о чем будет сказано ниже. Именно после возникновения микропроцессоров был сформулирован закон Мура – производительность ЭВМ возрастает в целом в два раза каждые девять месяцев.

Hyperthreading - это метод, оптимизирующий возможность процессора с одним ядром переключаться между приложениями. В процессорах с использованием этой технологии каждый физический процессор может хранить состояние сразу двух потоков, что для операционной системы выглядит как наличие двух логических процессоров. Физически у каждого из логических процессоров есть свой набор регистров и контроллер

прерываний, а остальные элементы процессора являются общими. Когда при исполнении потока одним из логических процессоров возникает пауза (в результате кэш-промаха, ошибки предсказания ветвлений, ожидания результата предыдущей инструкции), то управление передается потоку в другом логическом процессоре. Таким образом, пока один процесс ждет, например, данные из памяти, вычислительные ресурсы физического процессора используются для обработки другого процесса. По словам **Intel**, первая реализация потребовала всего 5-процентного увеличения площади кристалла, но позволяла увеличить производительность на 15—30%. Однако даже *hyperthreading* не позволяет выполнять процессы в действительно параллельном режиме.

В отличие от этой технологии, двоядерный процессор позволяет двум приложениям работать параллельно или одновременно. Результатом является повышенная производительность многозадачности внутри операционной среды. Также заметно значительное повышение эффективности на уровне многопоточных приложений - многочисленные процессы приложений могут выполняться одновременно.

Модуль предсказания условных переходов (*англ. Branch Prediction Unit*) — устройство, входящее в состав микропроцессоров, имеющих конвейерную архитектуру, определяющее направление ветвлений (предсказывающее, будет ли выполнен условный переход) в исполняемой программе. Предсказание ветвлений позволяет осуществлять предварительную выборку инструкций и данных из памяти, а также выполнять инструкции, находящиеся после условного перехода, до того, как он будет выполнен.

Параллелизм и многоядерность - наиболее эффективный способ повышения вычислительной мощности процессоров без увеличения энергопотребления. Объединение на одном кристалле нескольких ядер позволяет выполнять большее число операций за единицу времени при меньшем тепловыделении, а **многопоточность** каждого ядра дает возможность одновременно обрабатывать несколько команд.

С точки зрения программиста, микропроцессор, содержит две группы регистров – *пользовательские и системные*.

Пользовательские регистры доступны для использования в программах и включают следующие группы регистров:

1. Регистры общего назначения. Регистры этой группы используются для хранения данных и адресов. (**eax/ax/ah/al, ebx/bx/bh/bl, edx/dx/dh/dl, ecx/cx/ch/cl, ebp/bp, esi/si, edi/di, esp/sp**).
2. Сегментные регистры. Регистры этой группы используются для хранения адресов сегментов в памяти. (**cs, ds, ss, es, fs, gs**).

3. Регистры сопроцессора. Предназначены для написания программ, использующих тип данных с плавающей точкой. (**st(0)-st(7)**).

4. Целочисленные регистры **MMX**-расширения. (**mmx0-mmx7**). **MMX** (Multimedia Extensions — мультимедийные расширения) — коммерческое название дополнительного набора инструкций, выполняющих характерные для процессов кодирования/декодирования потоковых аудио/видео данных действия за одну машинную инструкцию. Процессор **Pentium MMX** отличается от «обычного» **Pentium** по шести основным пунктам:

- 4.1. добавлено 57 новых команд обработки данных;
- 4.2. увеличен в два раза объем внутреннего кэш (16 кб для команд и столько же — для данных);
- 4.3. увеличен объем буфера адресов перехода (Branch Target Buffer — BTB), используемого в системе предсказания переходов (Branch Prediction);
- 4.4. оптимизирована работа конвейера (Pipeline);
- 4.5. увеличено количество буферов записи (Write Buffers);
- 4.6. введено так называемое двойное электропитание процессора. Набор из 57 новых команд и является основным отличием; остальные пять — не более, чем сопутствующие изменения. Хотя, увеличенный объем кэш и внутренних буферов и оптимизированный конвейер несколько ускоряют работу любых приложений, однако основное увеличение производительности — до 60% — возможно только при использовании программ, правильно применяющих технологию **MMX** в обработке данных.

5. Регистры **MMX**-расширения с плавающей точкой. (**xmm0-xmm7**).

6. Регистры состояния и управления - это регистры, содержащие информацию о состоянии микропроцессора, исполняемой программы и позволяющие изменить это состояние. Они делятся на:

- 6.1. Регистр флагов. (**eflags/flags**).
- 6.2. Регистр указатель команды. (**eip/ip**).

Регистры для поддержания различных режимов работы, сервисных функций, а также регистры, специфичные для определенной модели микропроцессора.

Сегмент представляет собой непрерывный блок памяти.

Как правило, использование сегментов однозначно. Программа может содержать три сегмента — кода, данных и стека. Сегментация использовалась в старых микропроцессорах из-за ограничения их адресного пространства.

Ассемблер является символическим аналогом машинного языка. По этой причине программа, написанная на ассемблере, должна отражать все особенности архитектуры микропроцессора.

Программа на ассемблере представляет собой совокупность блоков памяти, которые называются сегментами. Программа может состоять из одного или нескольких таких сегментов. Каждый сегмент содержит совокупность предложений языка. Каждое предложение занимает отдельную строку программы.

Предложения ассемблера бывают четырех типов:

1. команды (инструкции), представляющие собой символические аналоги машинных команд. В процессе трансляции инструкции ассемблера преобразуются в соответствующие команды микропроцессора;
2. макрокоманды - оформляемые определенным образом предложения текста программы, замещаемые во время трансляции другими предложениями;
3. директивы, являющиеся указанием транслятору ассемблера на выполнение некоторых действий. У директив нет аналогов в машинном представлении;
4. строки комментариев, содержащие любые символы, в том числе и буквы русского алфавита. Комментарии игнорируются транслятором.

Допустимыми символами при написании текста программ являются:

1. Все латинские буквы **A-Z**, **a-z**. При этом заглавные и прописные буквы считаются эквивалентными.
2. Цифры от 0 до 9.
3. Знаки **?**, **@**, **\$**, **_**, **&**.
4. Разделители: **,**, **.**, **[**, **]**, **(**, **<**, **>**, **{**, **}**, **+**, **/**, *****, **%**, **!**, **"**, **"**, **?**, ****, **=**, **#**, **^**.

Идентификатор может состоять из одного или нескольких символов. В качестве символов можно использовать буквы латинского алфавита, цифры и некоторые специальные знаки — **_**, **?**, **\$**, **@**. Идентификатор не может начинаться символом цифры. Длина идентификатора может быть до 255 символов, хотя транслятор воспринимает лишь первые 32, а остальные игнорирует.

3. Организация памяти

Физическая память, к которой микропроцессор имеет доступ по шине адреса, называется оперативной памятью. На самом нижнем уровне память компьютера можно рассматривать как массив битов. Для физической реализации битов и работы с ними идеально подходят логические схемы. Но микропроцессору неудобно работать с памятью на уровне битов, поэтому ОЗУ реально организовано как последовательность байтов. Каждому байту соответствует его уникальный адрес (номер), который называется *физическим*. Диапазон значений физических адресов зависит от разрядности шины адреса микропроцессора. Механизм управления памятью полностью аппаратный. Это означает, что программа не может сама сформировать физический адрес памяти на адресной шине. В конечном итоге этот механизм позволяет обеспечить:

1. компактность хранения адреса в машинной команде;
2. гибкость механизма адресации;
3. защиту адресных пространств задач в многозадачной системе;
4. поддержку виртуальной памяти.

Микропроцессор аппаратно поддерживает две модели использования оперативной памяти:

Сегментированную модель. В этой модели программе выделяются непрерывные области памяти (сегменты), а сама программа может обращаться только к данным, которые находятся в этих сегментах.

Страничную. Ее можно рассматривать как надстройку над сегментированной моделью. В случае использования этой модели оперативная память рассматривается как совокупность блоков фиксированного размера. Основное применение этой модели связано с организацией виртуальной памяти, что позволяет операционной системе использовать для работы программ пространство памяти большее, чем объем физической памяти.

Особенности использования и реализации этих моделей зависят от режима работы микропроцессора:

1. *Режим реальных адресов (реальный режим).* Его наличие обусловлено желанием обеспечить в новых моделях микропроцессоров функционирование программ, разработанных для ранних моделей микропроцессоров.

2. *Защищенный режим.* Этот режим позволяет максимально реализовать все архитектурные идеи, заложенные в модели микропроцессоров, начиная с **i80286**.

Программы, разработанные для реального режима, не могут функционировать в защищенном режиме, это связано с особенностями формирования физического адреса в защищенном режиме.

3. *Режим виртуального 8086.* Переход в режим возможен, если микропроцессор уже находится в защищенном режиме. Несмотря на то, что микропроцессор находился в защищенном режиме, в режиме виртуального **i8086** возможна работа программ реального режима.

4. *Режим системного управления.* Он обеспечивает операционную систему механизмом для выполнения машинно-зависимых функций, таких как перевод компьютера в режим пониженного энергопотребления или выполнения действий по защите системы. Для перехода в данный режим микропроцессор должен получить специальный сигнал – SMI – от усовершенствованного программируемого контроллера прерываний APIC (Advanced Programmable Interrupt Controller), при этом сохраняет состояние вычислительной среды микропроцессора. Функционирование микропроцессора в этом режиме подобно его работе в режиме реальных адресов. Возврат из этого режима производится специальной командой микропроцессора.

Сегментированная модель памяти.

Сегментация – это механизм адресации, обеспечивающий существование нескольких независимых адресных пространств как в пределах одной задачи, так и в системе в целом для защиты задач от взаимного влияния.

Сегмент – это независимый, поддерживаемый на аппаратном уровне, блок памяти.

Под *физическим адресом* понимается адрес памяти, выдаваемый на шину адреса микропроцессора. Другое название этого адреса – *линейный адрес*. Эти названия являются синонимами, только при отключении страничного преобразования адреса. Это обусловлено наличием страничной модели организации оперативной памяти, которая является надстройкой над сегментированной моделью.

Каждая программа, в общем случае, может состоять из любого количества сегментов, но непосредственный доступ она имеет только к трем основным сегментам – *кода, данных, стека* и от одного до трех дополнительных сегментов данных. Программа никогда не знает, по каким физическим адресам будут размещены ее сегменты. Этим занимается операционная система. Операционная система размещает сегменты программы в оперативной памяти по определенным физическим адресам, после чего помещает значения этих адресов в определенные места. Куда именно зависит от режима работы микропроцессора. В реальном режиме эти адреса помещаются в соответствующие

сегментные регистры. В защищенном режиме – в элементы системной дескрипторной таблицы.

Различают 3 модели сегментированной организации памяти:

1. сегментированная модель памяти реального режима;
2. сегментированная модель памяти защищенного режима;
3. сплошная модель памяти защищенного режима.

Недостатки сегментной организации памяти:

Сегменты бесконтрольно перемещаются с любого адреса, кратного 16 (так как содержимое сегментного регистра аппаратно смещается на 4 разряда).

Сегменты имеют максимальный размер 64 Кбайта.

Сегменты могут перекрываться другими сегментами.

Для борьбы с этими недостатками в архитектуру был введен защищенный режим (см. лекцию №15).

4. Жизненный цикл программы на ассемблере

Надежность программы, в большинстве случаев, достигается за счет ее грамотного проектирования, а отнюдь не благодаря ее бесконечному тестированию. При применении такого стиля программирования ошибки являются легко локализуемыми и устранимыми.

Жизненный цикл программы состоит из следующих этапов:

1. *Этап постановки и формулировки задачи:*

- 1.1 Изучение предметной области, сбор материала;
- 1.2 Определение назначения программы, выработка требований к ней;
- 1.3 Формулирование требований к представлению исходных данных;
- 1.4 Определение структур входных и выходных данных;
- 1.5 Формирование ограничений и допущений на исходные и выходные данные.

2. *Этап проектирования:*

- 2.1 выбор метода реализации задачи;
- 2.2 разработка алгоритма реализации задачи;
- 2.3 разработка структуры программы в соответствии с выбранной моделью

памяти.

3. *Этап кодирования:*

- 3.1 уточнение структур входных и выходных данных и определение ассемблерного формата их представления;
- 3.2 программирование задачи;
- 3.3 комментирование текста программы и составление предварительного описания текста программы.

4. *Этап отладки и тестирования:*

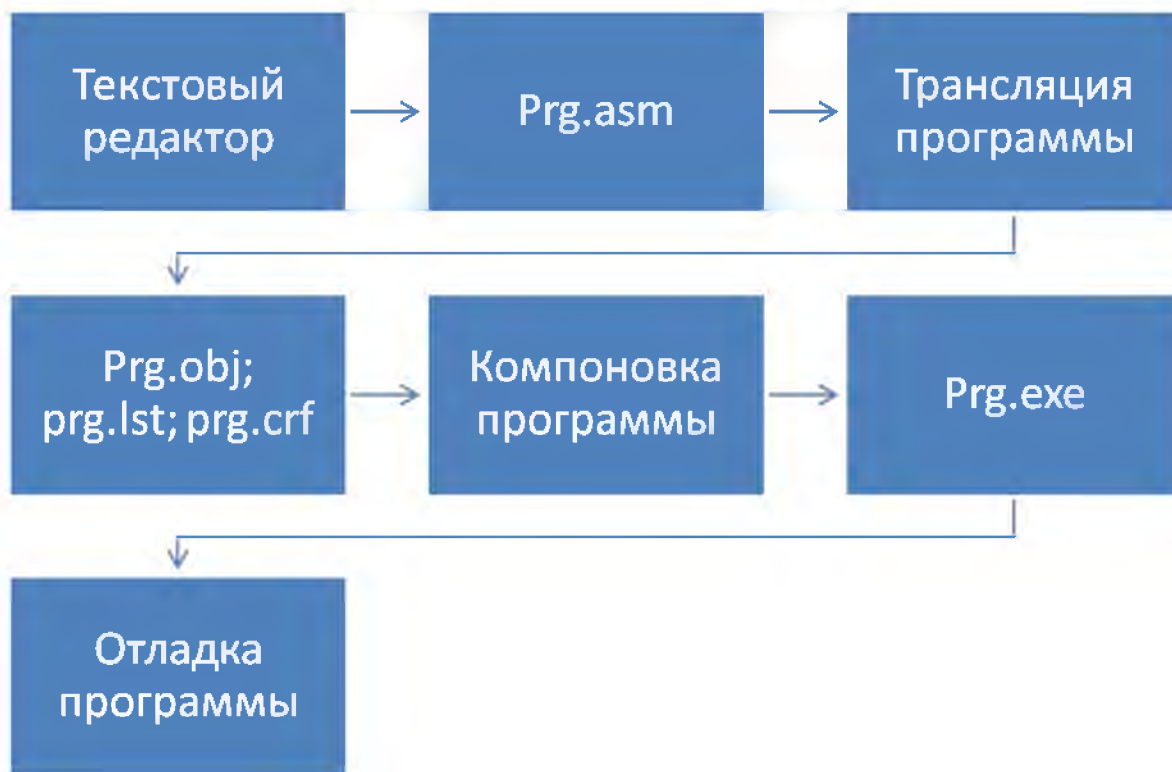
- 4.1 составление тестов для проверки правильности работы программы;
- 4.2 обнаружение, локализация и устранение ошибок, выявленных в тестах;
- 4.3 корректировка кода программы и ее описания.

5. *Этап эксплуатации и сопровождения:*

- 5.1 настройка программы на конкретные условия использования;
- 5.2 обучение пользователей работе с программой;
- 5.3 организация сбора сведений о сбоях в работе программы, ошибках в выходных данных, пожеланиях по улучшению интерфейса;

5.4 модификация программы с целью устранения выявленных ошибок.

Общая схема разработки программы на ассемблере:



Трансляция программы

На этом шаге формируется объектный модуль, который включает в себя представление исходной информации в машинных кодах.

В результате на экране вы получите последовательность строк. Если программа содержит ошибки, то транслятор выдаст на экран строки сообщений начинающихся со слов «**Error**» или «**Warning**» (ошибка или предупреждение). Наличие ошибок говорит о том, что у вас в программе есть недопустимые с точки зрения синтаксиса комбинации символов. Наличие строки предупреждений может послужить источником последующих ошибок. После исправления одной ошибки могут исчезнуть и последующие, так называемые наведенные ошибки.

Компоновка программы – создание исполняемого загрузочного модуля.

Цель этого шага – преобразовать код и данные в объектных файлах в их перемещаемое выполняемое отображение. Разделение процесса создания исполняемого модуля на два шага, трансляцию и компоновку, сделано намеренно, для того чтобы можно

было объединить вместе несколько модулей. Результатом работы компоновщика является создание загрузочного файла с расширением «*.exe».

На этапе отладки, используя описание алгоритма выполняется контроль правильности функционирования как отдельных участков кода, так и все программы в целом.

Отладчик представляет собой оконную среду отладки программ на уровне исходного текста.

Он позволяет решить две главные задачи:

1. Определить место логической ошибки.
2. Определить причину логической ошибки.

Стандартные возможности отладчика:

1. Выполнение трассировки программы в прямом направлении, то есть последовательное исполнение программы, при котором за один шаг выполняется одна машинная инструкция.

2. Выполнение трассировки программы в обратном направлении, то есть выполнение программы по одной команде, но в обратном направлении.

3. Просмотр и изменение состояния аппаратных ресурсов микропроцессора во время покомандного выполнения программы.

Это позволяет найти место и источник ошибки в программе.

5. Система команд микропроцессора

Программирование на уровне машинных команд – это тот минимальный уровень, на котором еще возможно программирование компьютера. Система машинных команд должна быть *достаточной* для того, чтобы реализовать требуемые действия, выдавая указания аппаратуре машины.

Каждая машинная команда состоит из 2-х частей: операционной части, определяющей что делать, и операндной части, определяющей объекты обработки.

В машинную команду явно и неявно входят следующие элементы:

1. *Поле префиксов* - элемент команды, который уточняет, либо модифицирует действие команды в следующих аспектах:
 - a. замена сегмента, если не удовлетворяет сегмент по умолчанию;
 - b. изменение размерности адреса;
 - c. указание на необходимость повторения команды.
2. *Поле кода операции* - определяющее действие команды. Одной и той же команде могут соответствовать несколько кодов операций, в зависимости от ее операндов.
3. *Поле операндов* - содержит от 0 до 2 элементов.

Важной особенностью машинных команд является то, что они не могут манипулировать одновременно двумя операндами, находящимися в оперативной памяти. Это означает, что в команде могут быть использованы: один регистр или регистр и некоторый операнд, который может либо непосредственно находиться в команде, либо располагаться в памяти.

По этой причине возможны только следующие сочетания операндов в команде:

регистр-регистр;

регистр-память;

память-регистр;

непосредственный операнд-регистр;

непосредственный операнд – память.

Префиксы – необязательные элементы машинной команды. Назначение префиксов – модифицировать операцию, выполняемую командой.

Префикс замены сегмента. В явной форме указывает, какой сегментный регистр используется в данной команде для адресации стека или данных.

Префикс разрядности адреса уточняет разрядность адреса – 16 или 32 разрядный адрес.

Префикс разрядности операнда указывает на разрядность операнда, с которым работает команда – 16 или 32 разрядный.

Префикс повторения закидывает команду для обработки всех элементов цепочки. Два типа префиксов: безусловные и условные, которые проверяют некоторые флаги и в результате проверки осуществляют досрочный выход из цикла.

На микропрограммном уровне операнд задается неявно. В этом случае команда явно не содержит операндов, а алгоритм выполнения команды использует некоторые объекты по умолчанию (регистры, флаги **eflags**).

Непосредственный операнд задается в самой команде. Операнд находится в коде команды, то есть является ее частью. Для хранения такого операнда в команде выделяется поле длиной до 32 бит.

Операнд может находиться в одном из регистров. Регистровые операнды указываются именами регистров.

32 разрядные регистры EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP

16 разрядные регистры AX, BX, CX, DX, SI, DI, SP, BP

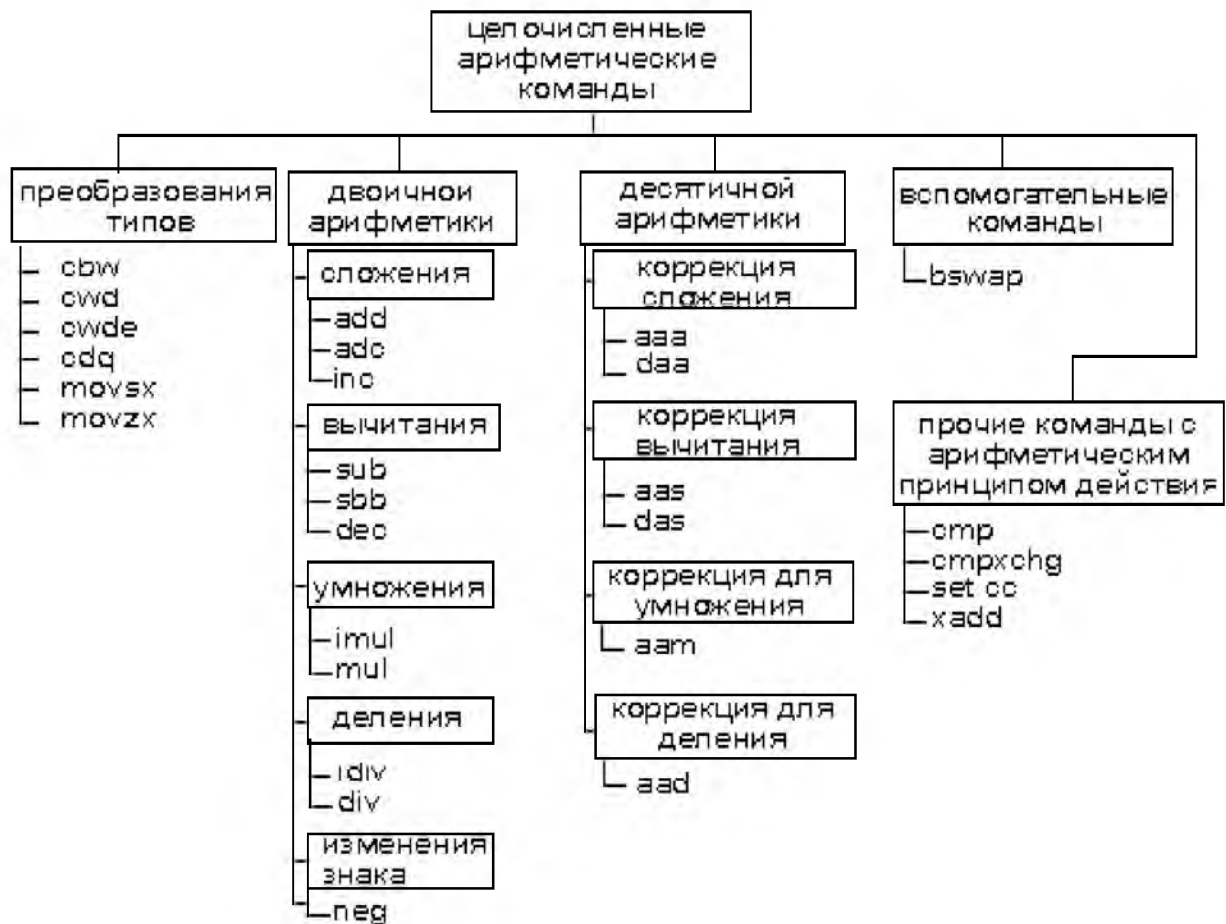
8 разрядные регистры AH, AL, BH, BL, CH, CL, DH, DL

Сегментные регистры CS, DS, SS, ES, FS, GS

Операнд располагается в памяти. Позволяет реализовать два основных вида адресации: прямую и косвенную адресацию.

Операндом является также порт ввода/вывода. Микропроцессор поддерживает адресное пространство ввода/вывода, которое используется для доступа к устройствам ввода/вывода. В качестве источника информации или получателя применяются регистры аккумуляторы **eax, ax, al**.

Целочисленные арифметические команды



Логические команды

1. *Операция логического умножения.* Команда выполняет поразрядно логическую операцию «И», конъюнкцию. Результат записывается на место «операнд_1».

and операнд_1, операнд_2

2. *Операция логического сложения.* Команда выполняет поразрядно логическую операцию «ИЛИ». Результат записывается на место операнд_1.

or операнд_1, операнд_2

3. *Операция исключающего «ИЛИ».* Результат на место операнд_1.

xor операнд_1, операнд_2

4. *Операция «ПРОВЕРИТЬ».* Команда выполняет поразрядно логическую операцию «И» над битами операторов. Состояние операндов остается прежним, изменяются только флаги **zf**, **sf**, **pf**, что дает возможность анализировать состояние отдельных битов операнда без изменения их состояния.

test операнд_1, операнд_2

5. *Операция логического отрицания.* Команда выполняет поразрядное инвертирование каждого бита операнда.

not операнд

Флаги.

Исходя из особенностей использования, флаги регистра **eflags/flags** можно разделить на три группы:

1. *8 флагов состояния.* Эти флаги могут изменяться после выполнения машинных команд. Флаги регистра **eflags** отражают особенности результата исполнения арифметических или логических операций. Это дает возможность анализировать состояние вычислительного процесса и реагировать на него с помощью команд условных переходов и вызова подпрограмм.

2. *1 флаг управления. df (directory flag).* Он используется цепочечными командами. Значение флага определяет направление поэлементной обработки: от начала строки к концу (**df=0**) либо, наоборот - от конца строки к ее началу (**df=1**).

3. *5 системных флагов.* Управляют вводом/выводом, маскируемыми прерываниями, отладкой, переключением между задачами.

6. Команды обмена данными

К группе команд пересылки данных относятся следующие команды:

mov <операнд назначения>, <операнд-источник>
xchg <операнд1>, <операнд2>

MOV — это основная команда пересылки данных. Она реализует самые разнообразные варианты пересылки. Отметим особенности применения этой команды. Командой **MOV** нельзя осуществить пересылку из одной области памяти в другую. Если такая необходимость возникает, то нужно использовать в качестве промежуточного буфера любой доступный в данный момент регистр общего назначения.

Нельзя загрузить в сегментный регистр значение непосредственно из памяти. Для такой загрузки требуется промежуточный объект. Это может быть регистр общего назначения или стек.

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

Но есть и другой, более «красивый» способ выполнения данной операции — использование стека и команд **PUSH** и **POP**:

push ds ; *поместить значение регистра ds в стек*
pop es ; *записать в es число из стека*

Нельзя использовать сегментный регистр **CS** в качестве операнда назначения. Причина здесь простая. Дело в том, что в архитектуре процессора пара **CS:IP** содержит адрес команды, которая должна выполняться следующей. Изменение командой **MOV** содержимого регистра **CS** фактически означало бы операцию перехода, а не пересылки, что недопустимо. В связи с командой **MOV** отметим один тонкий момент. Пусть в регистре **BX** содержится адрес некоторого поля (то есть мы используем косвенную базовую адресацию). Его содержимое нужно переслать в регистр **AX**. Очевидно, что нужно применить команду **MOV** в форме;

mov ax, [bx]

Транслятор задает себе вопрос: что адресует регистр **BX** в памяти — слово или байт? Обычно в этом случае он принимает решение сам, по размеру большего операнда, но может и выдать предупреждающее сообщение о возможном несовпадении типов операндов.

Что адресуется регистром **BX** в памяти — *байт, слово или двойное слово*? Допустим также, что требуется инициализировать поле, адресуемое регистром **BX**, значением 0. Очевидно, что одно из решений — применение команды **MOV**:

mov [bx],0

И опять у транслятора вопрос: *какую машинную команду ему конструировать?*

Для инициализации байта? Для инициализации слова? Для инициализации двойного слова?

Во всех этих случаях необходимо уточнять тип используемых операндов. Для этого существует специальный оператор ассемблера **PTR**. Правильно записать приведенные ранее команды можно следующим образом:

mov ax,word ptr[bx];	<i>если [bx] адресует слово в памяти</i>
inc byte ptr[bx];	<i>если [bx] адресует байт в памяти</i>
dec dword ptr[bx];	<i>если [bx] адресует двойное слово в памяти</i>
mov word ptr[bx],0;	<i>если [bx] адресует слово в памяти</i>

Можно рекомендовать использовать оператор **PTR** во всех сомнительных относительно согласования размеров операндов случаях. Также этот оператор нужно применять, когда требуется принудительно поменять размерность операндов. Несмотря на то, что поле **fp** имеет тип **WORD**, мы сообщаем ассемблеру, что поле нужно трактовать как байтовое, и заставляем вычислить значение эффективного адреса второго операнда как смещение **fp** плюс единица. Тем самым мы получаем доступ ко второму байту поля **fp**. Для двунаправленной пересылки данных применяют команду **XCHG**. Для этой операции можно, конечно, применить последовательность из нескольких команд **MOV**, но из-за того что операция обмена используется довольно часто, разработчики системы команд процессора посчитали нужным ввести отдельную команду обмена — **XCHG**. Естественно, что операнды должны иметь один тип. Не допускается (как и для всех

команд ассемблера) напрямую обменивать между собой содержимое двух ячеек памяти. К примеру:

*обменять содержимое регистров **ax** и **bx***

xchg ax, bx

*обменять содержимое регистра **ax** и слова в памяти по адресу в **[si]***

xchg ax,word ptr [si]

Ввод/вывод в порт

Система ввода-вывода, т.е. комплекс средств обмена информацией с внешними устройствами, является важнейшей частью архитектуры процессора и машины в целом. К системе ввода-вывода можно отнести и способы подключения к системной шине различного оборудования, и процедуры взаимодействия процессора с этим оборудованием, и команды процессора, предназначенные для обмена данными с внешними устройствами. Непрерывное совершенствование микропроцессоров и стремление максимально повысить производительность всей вычислительной системы, привело к существенному усложнению внутренней организации компьютеров: повышению разрядности шин, появлению внутренних быстродействующих магистралей обмена данными, использованию кэш-буферов для ускорения обмена с памятью и дисками и проч. Если, однако, отвлечься от важных с точки зрения производительности, но несущественных для программиста деталей, то логическую схему современного компьютера можно представить традиционным образом, в виде системной шины (магистральной), к которой подключаются сам микропроцессор и все устройства компьютера.

Системная шина представляет собой, в сущности, набор линий - проводов, к которым единообразно подключаются все устройства компьютера. В более широком плане в понятие системной шины следует включить электрические и логические характеристики сигналов, действующих на линиях шины, их назначение, а также правила взаимодействия этих сигналов при выполнении тех или иных операций на шине - то, что обычно называют протоколами обмена информацией. Сигналы, распространяющиеся по шине, доступны всем подключенным к ней устройствам, и в задачу каждого устройства входит выбор предназначенных ему сигналов и обеспечение реакции на них, соответствующей протоколу обмена. Процессор связан с системной шиной большим количеством линий (практически всеми своими выводами), из которых нас будут интересовать только линии трех категорий: набор линий адресов, набор линий данных и

один из сигналов управления. Последний сигнал, строго говоря, имеется только среди выходных сигналов микропроцессора. Процессор, желая записать данные по некоторому адресу в памяти, выставляет на линии адресов требуемый адрес, а на линии данных - данное. Устройство управления памятью расшифровывает поступивший адрес и, если этот адрес принадлежит памяти, принимает с линий данных поступившее данное и заносит его в соответствующую ячейку памяти. Описанная процедура отражает выполнение процессором команды типа:

mov mem, AX

где **mem** - символическое обозначение ячейки памяти, принадлежащей сегменту данных программы. Если процессор, выполняя команду типа:

mov AX, mem

должен прочесть данные из памяти, он выставляет на линии адресов требуемый адрес и ожидает поступления данных. Устройство управления памятью, расшифровав поступивший адрес и убедившись в наличии такого адреса в памяти, отыскивает в памяти требуемую ячейку, считывает из нее данное и выставляет его на линии данных. Процессор снимает данные с шины и отправляет его в указанный в команде операнд (в данном случае в регистр **AX**).

Описанные процедуры записи и чтения справедливы не только по отношению к памяти; для всех остальных устройств компьютера они выглядят точно так же. За каждым устройством закреплена определенная группа адресов, на которые оно должно отзываться. Обнаружив свой адрес на магистрали, устройство, в зависимости от заданного процессором направления передачи данных, либо считывает с магистрали поступившие данные, либо, наоборот, устанавливает имеющиеся в нем данные на магистраль.

Аналогичная ситуация наблюдается и со многими другими устройствами компьютера. Например, контроллер прерываний, служащий для объединения сигналов прерываний от всех устройств компьютера и направления их на единственный вход прерывания микропроцессора, управляется через два адреса. Поскольку в состав машины всегда включают два контроллера, для них выделяются две пары адресов. Во всех компьютерах типа **IBM PC** контроллерам прерываний назначаются адреса 20h-21h и A0h-A1h, которые так же отвечают и некоторым байтам оперативной памяти.

Программное разделение устройств реализуется с помощью двух наборов команд процессора - для памяти и для устройств ввода/вывода.

В *первую группу* команд входят практически все команды процессора, с помощью которых можно обратиться по тому или иному адресу - команды пересылки **mov** и **movs**, арифметических действий **add**, **mul** и **div**, сдвигов **rol**, **ror**, **sal** и **sar**, анализа содержимого байта или слова **test** и многие другие. Фактически, в эту группу команды входит большинство команд процессора.

Вторую группу команд образуют специфические команды ввода-вывода. В МП 86 их всего две - команда ввода **in** и команда вывода **out**. Таким образом, при обращении к памяти и к видеобуферу программист может использовать все подходящие по смыслу команды процессора, при этом, работая, например, с видеобуфером, можно не только засылать в него (или получать из него) данные, но и выполнять прямо в видеобуфере любые арифметические, логические и прочие операции. Обращаться же к контроллерам тех или иных устройств (и, между прочим, к видеоадаптеру), допустимо только с помощью двух команд - **in** и **out**.

Арифметические операции или анализ данных в устройстве невозможен. Необходимо сначала прочитать в процессор данные из внешнего устройства, и лишь затем выполнять над ним требуемую операцию. Наличие двух категорий адресов устройств дает основание говорить о существовании двух адресных пространств:

пространства памяти, куда входит сама память, а также видеобуферы и ПЗУ;
пространства ввода/вывода (пространства портов), куда входят адреса остальной аппаратуры компьютера.

При этом, если объем адресного пространства памяти составляет 1 Мбайт (а в защищенном режиме 4 Гбайт), то адресное пространство портов гораздо меньше - его размер составляет всего 64 Кбайт. Эта величина определяется форматом команд ввода/вывода. Адрес адресуемого порта должен быть записан в регистр **DX** (и ни в какой другой) и, таким образом, максимальное значение этого адреса составляет величину **FFFFh**. Реально из 64 Кбайт адресного пространства портов используется лишь очень малая часть.

Работа с адресами и указателями

При написании программ на ассемблере производится интенсивная работа с адресами операндов, находящимися в памяти. Для поддержки такого рода операций есть специальная группа команд, в которую входят следующие команды:

загрузка эффективного адреса

lea <приемник>, <источник>

загрузка указателя в регистр сегмента данных ds

lds <приемник>, <источник>

загрузка указателя в регистр дополнительного сегмента данных es

les <приемник>, <источник>

загрузка указателя в регистр дополнительного сегмента данных gs

lgs <приемник>, <источник>

Команда **LEA** похожа на команду **MOV** тем, что она также производит пересылку, однако команда **LEA** производит пересылку не данных, а эффективного адреса данных (то есть смещения данных относительно начала сегмента данных) в регистр, указанный операндом <приемник>.

Часто для выполнения некоторых действий в программе недостаточно знать значение одного лишь эффективного адреса данных, а необходимо иметь полный указатель на данные. Вы помните, что полный указатель на данные состоит из сегментной составляющей и смещения. Все остальные команды этой группы позволяют получить в паре регистров такой полный указатель на операнд в памяти. При этом имя сегментного регистра, в который помещается сегментная составляющая адреса, определяется кодом операции. Соответственно, смещение помещается в регистр общего назначения, указанный операндом <приемник>. Но не все так просто с операндом <источник>. На самом деле в команде в качестве источника нельзя указывать непосредственно имя операнда в памяти, на который мы бы хотели получить указатель. Предварительно необходимо получить само значение полного указателя в некоторой области памяти и задать в команде получения полного адреса имя этой области. Для выполнения этого действия необходимо вспомнить директивы резервирования и инициализации памяти. При применении этих директив возможен частный случай, когда в поле операндов указывается имя другой директивы определения данных (фактически, имя переменной). В этом случае в памяти формируется адрес этой переменной. Какой адрес будет сформирован (эффективный или полный), зависит от применяемой директивы. Если это **DW**, то в памяти формируется только 16-разрядное значение эффективного адреса, если же **DD** — в память записывается полный адрес. Размещение этого адреса в памяти следующее: в младшем слове находится смещение, в старшем — 16-разрядная сегментная составляющая адреса.

7. Прерывания

Прерывания – инициируемый определенным образом процесс, временно переключающий микропроцессор на выполнение другой программы с последующим возобновлением прерванной программы. Механизм прерываний позволяет обеспечить наиболее эффективное управление не только внешними устройствами, но и программами.

Прерывания могут быть внешними и внутренними. Внешние прерывания вызываются внешними, по отношению к микропроцессору, событиями. У микропроцессора есть два физических контакта **INTR** и **NMI**.

INTR (Interrupt Request)

NMI (Non Maskable Interrupt)

На них и формируются внешние по отношению к микропроцессору сигналы, возрастающие фронты которых извещают о том, что некоторое внешнее устройство инициирует прерывание. Вход **INTR (Interrupt Request)** предназначен для фиксации запросов от различных периферийных устройств. Вход **NMI (Non Maskable Interrupt)** – немаскируемое прерывание. Используется для того, чтобы сообщить микропроцессору о событии, требующем безотлагательной обработки или критической ошибке. Внешние прерывания относятся к непланируемым прерываниям.

Внутренние прерывания возникают внутри микропроцессора во время вычислительного процесса. К их возбуждению приводит одна из двух причин:

1. ненормальное внутреннее состояние микропроцессора, возникшее при обработке некоторой команды программы. Такие события называют исключительными ситуациями или исключениями. Этот вид прерываний отчасти можно отнести к непланируемым;

2. обработка машинной команды **int xx**. Этот тип прерываний называется программным. Это – планируемые прерывания, так как с их помощью программист обращается в нужное для него время за обслуживанием своих запросов либо к операционной системе, либо к **BIOS**, либо к собственным программам обработки прерываний.

Обработка прерываний в реальном и защищенном режиме осуществляется принципиально разными методами. **Система прерываний** – это совокупность

программных и аппаратных средств, реализующих механизм прерываний. К аппаратным средствам относятся:

1. Выводы микропроцессора:

а. **INTR** – вывод для входного сигнала внешнего прерывания. На этот вход поступает выходной сигнал от микросхемы контроллера прерываний 8259А.

б. **INTA** – вывод микропроцессора для выходного сигнала подтверждения получения сигнала прерывания микропроцессором. Этот выходной сигнал поступает на одноименный вход **INTA** микросхемы контроллера прерываний 8259А.

с. **NMI** – вывод микропроцессора для входного сигнала немаскируемого прерывания.

2. Микросхема программируемого контроллера прерываний 8259А. Она предназначена для фиксирования сигналов прерываний от восьми различных внешних устройств.

3. Внешние устройства: клавиатура, таймер, диски и т.д.

К программным средствам системы прерываний реального режима относятся:

1. Таблица векторов прерываний. В этой таблице в определенном формате, который зависит от режима работы микропроцессора, содержатся указатели на процедуры обработки соответствующих прерываний.

2. Следующие флаги в регистре флагов **flags\eflags**:

а. **IF (Interrupt Flag)** – флаг прерывания. Предназначен для маскирования (запрещения) аппаратных прерываний, то есть прерываний по входу **INTR**. На обработку прерываний остальных типов флаг **IF** влияния не оказывает. Если **IF=1** микропроцессор обрабатывает внешние прерывания, если **IF=0** микропроцессор игнорирует сигналы на входе **INTR**.

б. **TF (Trace Flag)** – флаг трассировки. Единичное состояние флага **TF** переводит микропроцессор в режим покомандной работы. В режиме покомандной работы после выполнения каждой машинной команды в микропроцессоре генерируется внутреннее прерывание с номером 1, и далее следуют действия в соответствии с алгоритмом обработки данного прерывания.

3. Машинные команды микропроцессора **int**, **into**, **iret**, **cli**, **sti**.

Функции, выполняемые микросхемой контроллера прерываний:

- фиксирование запросов на обработку прерывания от восьми источников, формирование единого запроса на прерывание и подача его на вход **INTR** микропроцессора;

- формирование номера вектора прерывания и выдача его на шину данных;
- организация приоритетной обработки прерываний;
- запрещение (маскирование) прерываний с определенными номерами.

Важное свойство контроллера i8259A – возможность его программирования, что позволяет достаточно гибко изменять алгоритмы обработки аппаратных прерываний.

Исходя из этого, микросхема имеет два состояния:

1. состояние настройки параметров обслуживания прерываний, во время которого, путем посылки в определенном порядке так называемых управляющих слов, производится инициализация контроллера;
2. состояние работы – это обычное состояние контроллера в котором производится фиксация запросов на прерывание и формирование управляющей информации для микропроцессора в соответствии с параметрами настройки.

Назначение основных структурных компонентов контроллера прерываний:

1. регистр запросов на прерывания **IRR (interrupt request register)** – восьмиразрядный регистр фиксирующий поступление сигнала на один из входов **irq0-irq7**. Фиксация выражается в установке соответствующего бита в единичное состояние;
2. регистр маскирования прерываний **IMR (interrupt mask register)** – восьмиразрядный регистр, с помощью которого можно запретить обработку запросов на прерывания, поступающих на соответствующие входы **irq0-irq7**. Для запрещения определенных уровней прерываний необходимо установить соответствующие биты регистра **IMR**. Эта операция осуществляется путем программирования порта 21h;
3. Регистр обслуживаемых прерываний **ISR (interrupt service register)** – восьмиразрядный регистр, единичное состояние разрядов которого показывает, прерывания каких уровней обрабатываются в данный момент в микропроцессоре;
4. арбитр приоритетов **PR (priority resolver)** – функцией данного блока является разрешение конфликта при одновременном поступлении запросов на входы;
5. блок управления – основной функцией данного блока является организация информационного обмена контроллера прерываний и микропроцессора через шину данных.

При поступлении сигнала на вход **INTR** в микропроцессоре происходят следующие процессы:

1. анализируется флаг **IF**;
2. если прерывания запрещены, то запрос на прерывания повисает до момента установки флага **IF** в единицу;

3. если прерывания разрешены, то:
 - a. сбрасывает флаг **IF** в ноль;
 - b. формирует сигнал подтверждения прерывания на входе микропроцессора **INTA**.

В процессе загрузки и работы компьютера контроллер прерываний настраивается на работу в одном из четырех режимов:

1. **FNM** – режим вложенных прерываний. В этом режиме каждому входу присваивается фиксированное значение приоритета, причем уровень **irq0** имеет наивысший приоритет, а **irq7** - наименьший. Приоритетность прерываний определяет их право на прерывание обработки менее приоритетного прерывания более приоритетным.

2. **ARM** – режим циклической обработки прерываний. В этом режиме значения приоритетов уровней прерываний также линейно упорядочены, но уже не фиксированным способом, а изменяются после обработки очередного прерывания по следующему принципу: значению приоритета последнего обслуженного прерывания присваивается наименьшее значение. Следующий по порядку уровень получает наивысшее значение и поэтому, при одновременном приходе запросов на прерывания от нескольких источников, преимущество будет иметь этот уровень. Это дает возможность обеспечить равноправие при обработке прерываний.

8. Команды передачи управления

Обычно в программе есть точки, в которых нужно принять решение о том, какая команда будет выполняться следующей. Это решение может быть:

1. *безусловным* – в данной точке необходимо передать управление не той команде, которая идет следующей, а другой, которая находится на некотором удалении от текущей команды;

2. *условным* – решение о том, какая команда будет выполняться следующей, принимается на основе анализа некоторых условий или данных.

То, какая команда будет выполняться следующей, микропроцессор узнает по содержимому пары регистров **cs:(e)ip**, в которой:

cs – сегментный регистр кода, в котором находится физический (базовый) адрес текущего сегмента кода;

еip/ip – регистр указателя команды в котором находится значение, представляющее собой смещение в памяти следующей команды подлежащей выполнению, относительно начала текущего сегмента кода.

Таким образом, команды передачи управления изменяют содержимое регистров, в результате чего микропроцессор выбирает для выполнения не следующую по порядку команду программы, а команду в некотором другом участке программы. Конвейер внутри микропроцессора при этом сбрасывается.

По принципу действия команды микропроцессора, обеспечивающие организацию переходов в программе можно разделить на три группы:

1. Команды безусловной передачи управления:

1.1 команда безусловного перехода;

1.2 вызов процедуры и возврат из процедуры;

1.3 вызов программных прерываний и возврат из программных прерываний.

2. Команды условной передачи управления:

2.1 команды перехода по результату команды сравнения;

2.2 команды перехода по состоянию определенного флага;

2.3 команды перехода по содержимому регистра есх/сх.

3. Команды управления циклом:

3.1 команда организации цикла со счетчиком есх/сх;

3.2 команда организации цикла со счетчиком есх/сх с возможностью досрочного выхода из цикла по дополнительному условию.

Возникает вопрос о том, каким образом обозначается то место, куда необходимо передать управление.

В языке ассемблера это делается с помощью меток.

Метка – это символическое имя обозначающее определенную ячейку памяти, предназначенное для использования в качестве операнда в командах передачи управления.

Транслятор ассемблера присваивает метке три атрибута:

1. имя сегмента кода, где эта метка описана;
2. смещение – расстояние в байтах от начала сегмента кода, в котором описана метка;

3. тип метки или атрибут расстояния, который может принимать два значения:

- 3.1 **near** – переход на эту метку возможен только в пределах сегмента кода где эта метка описана. Физически это означает что для перехода на метку достаточно изменить только содержимое регистра **еір/ір**;

- 3.2 **far** – переход на метку возможен только в результате межсегментной передачи управления для осуществления которой требуется изменение содержимого как регистра **еір/ір** так и регистра **сs**.

Метку можно определить двумя способами:

1. *оператором двоеточие* – с помощью этого способа можно определить метку только ближнего типа. Символическое имя в программе может быть определено только один раз. Определенную таким образом метку можно использовать в качестве операнда в командах условного перехода **jcc** и безусловного перехода **jmp, call**. Эти команды естественно должны быть в сегменте кода где определена метка;

2. *директивой label* – обычно директиву **label** используют для определения идентификатора заданного типа. Другой часто встречающийся случай использования директивы **label** – это организация доступа к одной области памяти как к области содержащей данные разных типов.

Метка может быть только одного типа – либо **near**, либо **far**.

Существует еще одно важное понятие ассемблера, имеющее прямое отношение к меткам – счетчик адреса команд.

Транслятор ассемблера обрабатывает исходную программу последовательно – команда за командой. При этом он ведет счетчик адреса команд, который для первой исполняемой команды равен нулю, а далее, по ходу обработки очередной команды

транслятором он увеличивается на длину этой команды. По сути, счетчик адреса команд – это смещение конкретной команды относительно начала сегмента кода. Таким образом, каждая команда во время трансляции имеет адрес, равный значению счетчика адреса команд.

Транслятор ассемблера обеспечивает две возможности работы с этим счетчиком:

1. использование меток, атрибуту смещения которых транслятор присваивает значение счетчика адреса той команды, перед которой они появились;
2. применение специального символа **\$** для обозначения счетчика адреса команд. Этот символ позволяет в любом месте программы использовать численное значение счетчика адреса.

Безусловные переходы

Что именно должно подвергнуться модификации зависит:

1. от типа операнда в команде безусловного перехода (ближний или дальний);
2. от указания в команде перехода перед адресом перехода модификатора. При этом сам адрес перехода может находиться либо непосредственно в команде (прямой переход), либо в регистре или ячейке памяти (косвенный переход).

Модификатор может принимать следующие значения:

Near ptr – прямой переход на метку внутри текущего сегмента кода.

Far ptr – прямой переход на метку в другом сегменте кода. Адрес перехода задается в виде непосредственного операнда или адреса

Word ptr – косвенный переход на метку внутри текущего сегмента кода.

Dword ptr – косвенный переход на метку в другом сегменте кода.

Команда безусловного перехода *Jmp*

Jmp [модификатор] адрес_перехода – безусловный переход без сохранения информации о точке возврата. **Адрес_перехода** представляет собой адрес в виде метки либо адрес области памяти, в которой находится указатель перехода. Дальность перехода определяется местоположением операнда **адрес_перехода**. Внутрисегментный переход предполагает, что изменяется содержимое только регистра **еір/ір**. Можно выделить три варианта внутрисегментного использования команды **jmp**:

1. прямой короткий
2. прямой
3. косвенный

9. Цепочечные команды

Элементарные или базовые типы данных:

1. Непосредственные данные, представляющие собой числовые или символьные значения, являющиеся частью команды.
2. Данные, описываемые с помощью ограниченного набора директив резервирования памяти, позволяющие выполнить самые элементарные операции по размещению и инициализации числовой и символьной информации.

Работа с ними поддерживается на уровне системы команд микропроцессора. Используя данные этих типов можно формализовать и запрограммировать практически любую задачу, с известной долей труда.

С целью облегчения разработки программ в язык ассемблера была введена возможность использования нескольких сложных типов данных: сканирование цепочки, загрузка элемента из цепочки, сохранение элемента в цепочке, получение элементов цепочки из порта ввода-вывода, вывод элементов цепочки в порт ввода-вывода. Они строятся на сравнении байт, слов, двойных слов.

Сканирование цепочки

Scas адрес_приемника

scasb

scasw

scasd

Загрузка элемента цепочки

lods адрес_источника

lodsb

lodsw

lodsd

Сохранение элемента в цепочке

stos адрес_приемника

stosb

stosw

stosd

Получение элементов цепочки из порта ввода/вывода

ins адрес_приемника, номер порта

insb

insw

insd

Вывод элементов цепочки в порт ввода/вывода

outs номер порта, адрес_источника

outsb

outsw

outsd

- **cmpsb**
- **cmpsw**
- **cmpsd**

Приведенные 3 команды сравнивают байт, слово, двойное слово из памяти по адресу **esi** с байтом, словом, двойным словом по адресу **edi** и устанавливают флаги. При этом **esi** и **edi** продвигаются.

rep

(repeat)

repe или **repz**

(repeat while equal or zero)

repne или **repnz**

(repeat while not equal or zero)

Логически к этим командам можно отнести и так называемые префиксы повторения. Они предназначены для использования цепочечными командами. Префиксы имеют свои обозначения. Эти префиксы повторения указываются перед нужной цепочечной командой в поле метки. Цепочечная команда без префикса выполняется один раз. Размещение префикса перед цепочечной командой заставляет ее выполняться в цикле. Отличия префиксов в том, на каком основании принимается решение о циклическом выполнении цепочечной команды: либо по состоянию регистра **ecx/cx**, либо по флагу нуля **zf**.

Префикс повторения **rep**. Этот префикс используется с командами, реализующими операции-примитивы пересылки и сохранения элементов цепочек – **movs**, **stos**. Префикс заставляет данные команды выполняться, пока содержимое в **ecx/cx** не

станет равным 0. При этом цепочечная команда, перед которой стоит префикс, автоматически уменьшает содержимое **есх/сх** на единицу. Та же команда, но без префикса этого не делает.

Префиксы повторения *gere* или *gerz*

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

Префиксы повторения *gerne* или *geriz*

Эти префиксы также синонимы. Они заставляют цепочечную команду циклически выполняться до тех пор, пока содержимое **есх/сх** не равно нулю или флаг **zf** равен нулю. При невыполнении одного из этих условий работа команды прекращается. Данные команды также можно использовать с командами **cmps** и **scas**, но для поиска совпадающих элементов цепочек.

Цепочка_источник, адресуемая операндом *адрес_источника* может находиться в текущем сегменте данных, определяемом регистром **ds**. *Цепочка_приемник*, адресуемая операндом *адрес_приемника*, должна быть в дополнительном сегменте данных, адресуемом сегментным регистром **es**. Допускается замена только регистра **ds**. Регистр **es** подменять нельзя. Вторые части адресов – смещения цепочек – также находятся в строго определенных местах. Для *цепочки_источника* это **esi/si** (индексный регистр источника). Для *цепочки_получателя* это регистр **edi/di** (индексный регистр приемника).

На самом деле набор команд микропроцессора имеет соответствующие машинные команды только для цепочечных команд ассемблера без операндов. Команды с операндами транслятор ассемблера использует только для определения типов операндов. После того как выяснен тип элементов цепочек по их описанию в памяти, генерируется одна из трех машинных команд для каждой из цепочечных операций.

По этой причине все регистры, содержащие адреса ячеек, должны быть инициализированы заранее, в том числе и для команд, допускающих явное указание операндов. Правильная загрузка регистров указателями обязательно требуется до выдачи любой цепочечной команды.

Есть две возможности направления обработки цепочек:

1. От начала цепочки к концу, то есть в направлении возрастания адресов.
2. От конца цепочки к началу, то есть в направлении убывания адресов.

Цепочечные команды сами выполняют модификацию регистров, адресующих операнды, обеспечивая тем самым автоматическое продвижение по цепочке. Количество байт, на которые эта модификация осуществляется, определяется кодом команды. Знак модификации определяется значением флага направления **df**.

Если **df=0**, то значения индексных регистров **esi/si** и **edi/di** будут автоматически увеличиваться цепочечными командами, то есть обработка будет осуществляться в направлении возрастания адресов.

Если **df=1**, то значения индексных регистров **esi/si** и **edi/di** будут автоматически уменьшаться, то есть обработка будет идти в направлении убывания адресов.

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

1. **Cld** – очистить флаг направления. Команда сбрасывает флаг направления в 0.
2. **Std** – установить флаг направления. Команда устанавливает флаг направления в 1.

10. Сложные структуры данных (часть 1)

Команды данной группы называют командами обработки строк символов. Отличие от цепочечных команд в том, что под строкой символов понимается последовательность байт, а цепочка это более общее название для случаев, элементы последовательности имеют размер больше байта (слово или двойное слово). Таким образом, цепочечные команды позволяют проводить действия над блоками памяти, представляющими собой последовательность элементов следующего размера.

8 бит – байт;

16 бит – слово;

32 бита – двойное слово.

Содержимое этих блоков для микропроцессора никакого значения не имеет. Главное, чтобы размерность элементов совпадала, и эти элементы находились в соседних ячейках памяти.

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

При необходимости использовать массив в программе его нужно моделировать одним из следующих способов:

1. Перечислением элементов массива в поле операндов одной из директив описания данных. При перечислении элементы разделяются запятыми.

- **mas dd 1,2,3,4,5**
- **mas dw 5 dup (0)**

2. Используя оператор повторения **dup**. На примере размер каждого элемента 2 байта. Такой способ определения используется для резервирования памяти с целью размещения и инициализации элементов массива.

3. Используя директивы **label** и **rept**. Директива **rept** относится к макросредствам языка ассемблера и вызывает повторение указанное число раз строк, заключенных между директивой и строкой **endm**. Достоинство директивы **label** в том, что она не резервирует память, а лишь определяет характеристики объекта. Используя

несколько директив **label**, написанных одна за другой можно присвоить одной и той же области памяти разные имена и типы.

4. Использованием цикла для инициализации значениями области памяти, которую можно впоследствии трактовать как массив.

Доступ к элементам массива.

Все элементы массива располагаются в памяти последовательно. Одну и ту же область памяти можно трактовать как одномерный массив, и, одновременно, те же самые данные могут трактоваться как двумерный массив. Все зависит только от алгоритма обработки данных в конкретной программе. То же самое касается и индексов элементов массива. При программировании на ассемблере, говоря об индексе, подразумевается не номер элемента массива, а некоторый адрес. Нумерация элементов в массиве начинается с нуля. В общем случае для получения адреса элемента в массиве необходимо начальный базовый адрес массива сложить с произведением индекса (номер элемента минус единица) этого элемента на размер элемента массива.

база + (индекс*размер элемента)

Для работы с массивами предоставляются: базовые и индексные регистры, позволяющие реализовать несколько режимов адресации данных. Микропроцессор позволяет масштабировать индекс. Это означает, что если указать после имени индексного регистра знак умножения с последующей цифрой 2, 4 или 8, то содержимое индексного регистра будет умножаться на 2, 4 или 8. Применение масштабирования облегчает работу с массивами, которые имеют размер элементов равный 2, 4 или 8 байтам, так как микропроцессор сам производит коррекцию индекса для получения адреса очередного элемента массива. Нужно лишь загрузить в индексный регистр значение требуемого индекса, считая от 0. Возможность масштабирования появилась в микропроцессорах **Intel**, начиная с **i486**.

Соглашения:

1. Если для описания адреса используется только один регистр, то речь идет о базовой адресации и этот регистр рассматривается как базовый.

переслать байт из области данных, адрес которой находится в регистре ebx
mov al, [ebx]

2. Если для задания адреса в команде используется прямая адресация в виде идентификатора в сочетании с одним регистром, то речь идет об индексной адресации. Регистр считается индексным и поэтому можно использовать масштабирование для получения адреса нужного элемента массива.

Сложить содержимое `eax` с двойным словом в памяти по адресу `mas+(ebx)*4`
`add eax, mas[ebx*4]`

3. Если в описании адреса используются 2 регистра, то речь идет о базово-индексной адресации. Левый регистр рассматривается как базовый, а правый – как индексный. В общем случае это не принципиально, но если используется масштабирование с одним из регистров, то он всегда является индексным. При применении регистров **`ebp/bp`** и **`esp/sp`** по умолчанию подразумевается, что сегментная составляющая адреса находится в регистре **`ss`**. Базово-индексную адресацию не запрещено сочетать с прямой адресацией или указанием непосредственного значения. Адрес тогда будет формироваться как сумма всех компонентов.

Двумерные массивы

Двумерный массив нужно моделировать. На описании самих данных это почти никак не отражается – память под массив выделяется с помощью директив резервирования и инициализации памяти. Непосредственно моделирование обработки массива производится в сегменте кода, где определяется некоторая область памяти, которую необходимо трактовать как двумерный массив. При этом за разработчиком остается решение: как трактовать расположение элементов двумерного массива в памяти: по строкам или по столбцам.

Если последовательность однотипных элементов в памяти трактуется как двумерный массив, расположенный по строкам, то адрес элемента **`(i,j)`** вычисляется по формуле.

$(\text{база} + \text{количество элементов в строке} * \text{размер элемента} * i + j)$

Организовать адресацию двумерного массива логично, используя базово-индексную адресацию. При этом возможны 2 основных варианта выбора компонентов для формирования эффективного адреса:

1. Сочетание прямого адреса, как базового компонента адреса и двух индексных регистров для хранения индексов.

mov ax, mas[ebx][esi]

2. Сочетание двух индексных регистров, один из которых является и базовым и индексным одновременно, а другой только индексным.

mov ax, [ebx][esi]

Структуры

Структура – тип данных, состоящий из фиксированного числа элементов разного типа. Для использования структур в программе необходимо выполнить 3 действия:

1. Задать шаблон структуры. Это означает определение нового типа данных, который впоследствии можно использовать для определения переменных этого типа.
2. Определить экземпляр структуры. Это подразумевает инициализацию конкретной переменной с заранее определенной, с помощью шаблона, структурой.
3. Организовать обращение к элементам структуры.

Описать структуру в программе означает лишь указать ее схему или шаблон, память при этом не выделяется. Этот шаблон можно рассматривать только как информацию для транслятора о расположении полей и их значении по умолчанию. Определить структуру значит дать указание транслятору выделить память и присвоить этой области памяти символическое имя. Описать структуру в программе можно только один раз, а определить – любое количество раз.

Описание структуры шаблона имеет следующий синтаксис.

имя_структуры STRUCT

<описание полей>

имя_структуры ENDS

Здесь **<описание полей>** представляет собой последовательность директив описания данных **db**, **dw**, **dd**, **dq** и **dt**. Их операнды определяют размер полей и при необходимости начальные значения.

11. Сложные структуры данных (часть 2)

При описании шаблона память не выделяется, так как это всего лишь информация для *транслятора*. Местоположение шаблона в программе может быть произвольным, но, судя по логике работы однопроходного транслятора, он должен быть расположен до того места, где определяется переменная с типом данной структуры. То есть при описании в сегменте данных переменной с типом некоторой структуры ее шаблон необходимо поместить в начало сегмента данных, либо перед ним.

Для использования описанной с помощью шаблона структуры в программе необходимо определить переменную с типом данной структуры. Для этого используется следующая синтаксическая конструкция.

[имя переменной] имя_структуры <[список значений]>

Имя_переменной – идентификатор переменной данного структурного типа. Задание имени переменной необязательно. Если его не указать будет просто выделена область памяти размером в сумму длин всех элементов структуры.

Список значений – заключенный в угловые скобки список начальных значений элементов структуры, разделенных запятыми. Его задание также необязательно. Если список указан не полностью, то все поля структуры для данной переменной инициализируются значениями из шаблона, если таковые заданы. Допускается инициализация отдельных полей, но в этом случае пропущенные поля должны разделяться запятыми. Пропущенные поля будут инициализированы значениями из шаблона структуры. Если при определении новой переменной с типом данной структуры мы согласны со всеми значениями полей в ее шаблоне, то нужно написать пустые угловые скобки.

Методы работы со структурой

Для того, чтобы сослаться в команде на поле некоторой структуры, используется специальный оператор – символ «.» Он используется в следующей синтаксической конструкции

Адресное_выражение.имя_поля_структуры

Адресное выражение – идентификатор переменной некоторого структурного типа или выражение в скобках в соответствии с синтаксическими правилами

Имя_поля_структуры – имя поля из шаблона структуры. Это тоже адрес, а точнее, смещение поля от начала структуры.

Таким образом, оператор «точка» вычисляет выражение.

(адресное_выражение)+(имя_поля_структуры)

Аналогично другим идентификаторам, определенным в программе, транслятор назначает имени типа структуры и имени переменной с типом структуры атрибут типа. Значением этого атрибута является размер в байтах, занимаемый полями этой структуры. Извлечь это значение можно с помощью оператора `type`. После того как стал известен размер экземпляра структуры, организовать индексацию в массиве структур несложно.

Объединения

Язык ассемблера предоставляет возможность переопределения области памяти для работы с объектом другого имени и типа. Для этого существует специальный тип данных – объединения – это тип данных, позволяющий трактовать одну и ту же область памяти как имеющую разные типы и имена. Описание объединений в программе напоминает описание структур, то есть сначала описывается шаблон, в котором с помощью директив описания данных перечисляются имена и типы полей.

Имя_объединения UNION

<описание полей>

Имя_объединения ENDS

Отличие объединений от структур состоит в том, что при определении типа объединения память выделяется в соответствии с размером максимального элемента. Обращение к элементам объединения происходит по их именам, но при этом нужно помнить о том, что все поля в объединении накладываются друг на друга. Одновременная работа с элементами объединения исключена. В качестве элементов объединения можно использовать и структуры.

Записи.

Запись – это структурный тип данных, состоящий из фиксированного числа элементов, длиной от одного до нескольких бит. При описании записи для каждого элемента указывается его длина в битах и, необязательно, некоторое значение. Суммарный размер записи определяется суммой размеров ее полей и не может быть более 8, 16 или 32 бит. Если суммарный размер записи меньше указанных значений, то все поля записи прижимаются к младшим разрядам. Использование записей в программе, так же как и структур, организуется в три этапа:

1. Задание шаблона записи, то есть определение набора битовых полей их длин, и при необходимости, инициализация полей.
2. Определение экземпляра записи. Так же как и для структур, этот этап подразумевает инициализацию конкретной переменной типом заранее определенной с помощью шаблона записи.
3. Организация обращения к элементам записи.

Описание записи

Имя записи RECORD <описание элементов>

Описание элементов представляет собой последовательность описаний отдельных элементов.

Для использования шаблона записи в программе необходимо определить переменную с типом данной записи для чего применяется следующая синтаксическая конструкция.

12. Модульное программирование (часть 1)

Одним из самых существенных недостатков программ на языке ассемблера является их недостаточная наглядность! Причины этого лежат на поверхности – при программировании на ассемблере программисту необходимо производить самые элементарные действия. При этом он должен учитывать и контролировать большое количество информации.

Технологии программирования

К настоящему моменту времени наиболее популярными и жизнеспособными оказались две технологии: структурная и объектно-ориентированная.

Последние версии языка ассемблера поддерживают объектно-ориентированное программирование, но его реализация слишком сложна. Типичному процессу написания программы на ассемблере более всего удовлетворяют концепции структурного программирования. Для микропроцессора **Intel** эти концепции поддерживаются на аппаратном уровне с помощью сегментации памяти и реализации команд передачи управления.

Структурное программирование – методология программирования, базирующаяся на системном подходе к анализу, проектированию и реализации программного обеспечения. Основу этой технологии составляют следующие положения:

1. Сложная задача разбивается на более мелкие, функционально лучше управляемые задачи. Каждая задача имеет один вход и один выход. В этом случае управляющий поток программы состоит из совокупности элементарных подзадач с ясным функциональным назначением.

2. Простота управляющих структур, используемых в задаче. Логически задача должна состоять из минимальной функционально полной совокупности достаточно простых управляющих структур.

3. Разработка программы должна вестись поэтапно. На каждом этапе должно решаться ограниченное число четко поставленных задач с ясным пониманием их значения и роли в контексте всей задачи. Если такое понимание не достигается - это говорит о том, что этап слишком велик и его нужно разделить на более элементарные шаги.

Модульное программирование является частью общего структурного подхода.

Концепция модульного программирования:

1. Функциональная декомпозиция задачи – разбиение большой задачи на ряд более мелких, функционально связанных между собой модулей. Модули связаны между собой только входными и выходными данными.

2. Модуль – основа концепции модульного программирования. Каждый модуль представляет собой черный ящик с одним входом и одним выходом. Модульный подход позволяет безболезненно проводить модернизацию программы в процессе ее эксплуатации и облегчает ее сопровождение. Дополнительно модульный подход позволяет разрабатывать части программ одного проекта на разных языках программирования, после чего с помощью компоновочных средств объединять их в единый загрузочный модуль.

3. Реализуемые решения должны быть простыми и ясными. Если назначение модуля непонятно, то это говорит о том, что декомпозиция начальной или промежуточной задачи недостаточна.

4. Назначение всех переменных модуля должно быть описано с помощью комментариев по мере их описания.

5. Исходный текст модуля должен иметь заголовок, в котором отражены как назначение модуля, так и его внешние связи. Этот заголовок можно назвать интерфейсной частью модуля. В этой части с помощью комментариев помещается следующая информация: назначение модуля, особенности функционирования, описание входных аргументов, описание выходных аргументов, использование внешних модулей и переменных, сведения о разработчике для защиты авторских прав.

6. В ходе разработки программы следует предусматривать специальные блоки операций, учитывающие реакцию на возможные ошибки в данных или действиях пользователя. Этот момент означает отсутствие тупиковых ветвей в алгоритме программы, в результате которых программа виснет и перестает отвечать на запросы. Любые непредусмотренные действия пользователя должны приводить к генерации ошибочной ситуации или к предупреждению о возможности возникновения такой ситуации.

Применительно к ассемблеру, рассматривают несколько форм организации управляющих связей:

1. Использование механизма макроподстановок, позволяющего изменять исходный текст программы, в соответствии с некоторыми предварительно описанными параметризованными объектами. Эти объекты имеют формальные аргументы, что

позволяет производить замещение их фактическими аргументами в процессе макрогенерации.

2. Использование механизма подпрограмм (процедур), написанных на ассемблере и структурно входящих в одну программу. В отличие от макрокоманд, взаимодействие процедур осуществляется на этапе выполнения программы.

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

4. Использование механизма динамического вызова исполняемых модулей и подключения библиотек «*.dll» для ОС Windows.

Основные информационные связи:

1. Использование общих областей памяти и общих программно-аппаратных ресурсов микропроцессора для связи модулей.

2. Унифицированную передачу аргументов при вызове модуля. Унификацию можно представить как на уровне пользователя, так и на уровне конкретного компилятора.

3. Унифицированную передачу аргументов при возврате управления из модуля.

Процедуры в языке ассемблера

Для оформления процедур как отдельных объектов существуют специальные директивы.

PROC/ENDP

машинная команда

RET

Процедуры также как и макрокоманды ценны тем, что могут быть активизированы в любом месте программы. Процедурам, также как и макрокомандам, могут быть переданы некоторые аргументы, что позволяет, имея одну копию кода в памяти, изменять ее для каждого конкретного случая использования. Возможные варианты размещения процедур в программе:

в начале программы (до первой исполняемой команды);

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

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

в другом модуле.

Главное условие - чтобы на процедуру не попадало управление.

13. Модульное программирование (часть 2)

Так как отдельный модуль – это функционально автономный объект, то он ничего не должен знать о внутреннем устройстве других модулей и наоборот. Каждый модуль должен иметь такие средства, с помощью которых он извещал бы транслятор о том, что некоторый объект должен быть видимым вне этого модуля. И транслятору необходимо объяснить, что некоторый объект находится вне данного модуля. Это позволит ему правильно сформировать машинные команды. На своем этапе компоновщики произведут настройку модулей и разрешат все внешние ссылки в объединяемых модулях.

Для того чтобы объявить о подобного рода видимых извне объектах, программа должна использовать две директивы.

extrn имя: тип,..., имя:тип

public имя,..., имя

Директива **extrn** предназначена для объявления некоторого имени внешним, по отношению к данному модулю. Директива **public** предназначена для объявления некоторого имени, определенного в этом модуле и видимого в других модулях. В синтаксисе имя – это идентификатор, определенный в другом модуле. В качестве идентификатора могут выступать:

1. имена переменных, определенных директивами **db, dw**;
2. имена процедур;
3. имена констант, определенных операторами **=** и **equ**.

Тип определяет тип идентификатора. Указание типа необходимо для того, чтобы транслятор правильно сформировал соответствующую машинную команду. Действительные адреса будут вычислены на этапе редактирования, когда будут разрешаться внешние ссылки. Возможные значения типа определяются допустимыми типами объектов для этих директив.

<i>Имя – это...</i>	<i>Возможные значения</i>
имя переменной	Byte, word, dword, pword, fword, qword, tbyte
имя процедуры	Near, far
имя константы	abs

Организация интерфейса с процедурой

Аргумент – это ссылка на некоторые данные, которые требуются для выполнения возложенных на модуль функций и разрешенных вне этого модуля. По аналогии с макрокомандами рассматривают понятия формального и фактического аргументов.

Формальный аргумент – это местодержатель для действительных данных, которые будут подставлены с помощью фактического аргумента.

Фактический аргумент – это то, что фактически передается на место формального аргумента.

Переменная – это нечто, размещенное в ячейке памяти или регистре, и может подвергаться изменению.

Константа – это данные, значения которых никогда не меняются.

Существуют следующие варианты передачи аргументов в модуль:

1. через регистры;
2. через общую область памяти;
3. через стек;
4. с помощью директивы **extrn** и **public**.

Передача аргументов через регистры

Это наиболее простой в реализации способ передачи данных. Данные, переданные подобным способом, становятся доступными немедленно после передачи управления процедуре. Этот способ особенно эффективен при небольшом объеме передаваемых данных. Ограничения на способ передачи аргументов через регистры:

1. Небольшое число доступных для пользователя регистров.
2. Нужно постоянно помнить о том, какая информация, в каком регистре находится.
3. Ограничение размера передаваемых данных размерами регистра. Если размер данных превышает 8, 16 или 32 бита, то передачу данных посредством регистров произвести нельзя. В этом случае необходимо передавать не сами данные, а указатели на них.

Передача аргументов через общую область памяти

Этот вариант передачи аргументов предполагает, что вызывающая и вызываемая программы условились использовать некоторую область памяти как общую. Транслятор предоставляет специальное средство для организации такой области памяти – атрибут

комбинирования сегментов. Наличие данного атрибута указывает компоновщику как нужно комбинировать сегменты, имеющие одно имя. Значение

common

говорит о том, что все сегменты, имеющие одинаковое имя в объединяемых модулях, будут располагаться компоновщиком, начиная с одного адреса оперативной памяти. Это значит, что они будут просто перекрываться в памяти и, следовательно, совместно использовать выделенную память. Недостатком этого способа в реальном режиме работы микропроцессора является отсутствие средств защиты данных от разрушения, так как нельзя проконтролировать соблюдение правил доступа к этим данным.

Передача аргументов через стек

Этот способ наиболее часто используется для передачи аргументов при вызове процедур. Суть его заключается в том, что вызывающая процедура самостоятельно заносит в стек передаваемые данные, после чего производит вызов процедуры. Стек обслуживается тремя регистрами **ss**, **sp**, **bp**. Микропроцессор автоматически работает с регистрами **ss** и **sp** в предположении, что они всегда указывают вершину стека. По этой причине их содержимое изменять не рекомендуется. Для осуществления произвольного доступа к данным в стеке архитектура микропроцессора имеет специальный регистр **ebp\bp**, который автоматически предполагает работу с сегментом стека. Перед использованием этого регистра для доступа к данным стека его содержимое необходимо правильно проинициализировать, что предполагает формирование в нем адреса, который бы указывал непосредственно на переданные данные. Для этого в начало процедуры рекомендуется включать дополнительный фрагмент кода, который имеет название пролог процедуры. Обычно код пролога состоит всего из двух команд. Первая команда

push bp

сохраняет содержимое **bp** в стеке с тем, чтобы исключить порчу находящегося в нем значения. Вторая команда

mov bp, sp

настраивает **bp** на вершину стека, после чего можем не волноваться о том, что содержимое **sp** перестанет быть актуальным и осуществлять прямой доступ к содержимому стека.

Конец процедуры должен быть оформлен особым образом и содержать действия, обеспечивающие корректный возврат из процедуры. Фрагмент кода, выполняющего такие действия, имеет название эпилог процедуры. Код эпилога должен восстановить контекст

программы в точке вызова процедуры из вызывающей программы. При этом, в частности, нужно откорректировать содержимое стека, убрав из него ставшие ненужными аргументы, передававшиеся в процедуру. Это можно сделать несколькими способами:

1. Используя последовательность из **n** команд **pop xx**. Лучше всего это делать в вызывающей программе сразу после возврата управления из процедуры.
2. Откорректировать регистр указателя стека **sp** на величину **2*n**, например командой **add sp, NN** где **NN=2*n**, а **n** – количество аргументов. Это также лучше делать после возврата управления вызывающей процедуре.
3. Используя команду **ret n** в качестве последней исполняемой команды в процедуре, где **n** – количество байт, на которое нужно увеличить содержимое регистра **esp\sp** после того, как со стека будут сняты составляющие адреса возврата. Этот способ аналогичен предыдущему, но выполняется микропроцессором автоматически.

Аргументы могут передаваться либо *по значению*, либо *по адресу*.

14.Создание WINDOWS-приложений на ассемблере

Комбинирование программ на языке высокого уровня с кодом на ассемблере используется в том случае, если присутствуют участки, которые невозможно реализовать без использования ассемблера, либо его применение значительно повысит эффективность работы.

Существуют следующие формы комбинирования программ на языках высокого уровня с ассемблером:

Использование операторов типа **inline** и ассемблерных вставок. Эта форма сильно зависит от синтаксиса языка высокого уровня и конкретного компилятора. Она предполагает, что ассемблерные коды в виде команд ассемблера или прямо в машинных командах вставляются в текст программы на языке высокого уровня. Компилятор языка распознает их как ассемблера и без изменений включает в формируемый им объектный код. Эта форма удобна, если вставлять небольшой фрагмент.

Использование внешних процедур и функций. Это более универсальная форма комбинирования.

У нее есть ряд преимуществ:

1. Написание и отладку программ можно производить независимо.
2. Написанные программы можно использовать в других проектах.
3. Облегчаются модификация и сопровождение подпрограмм в течение жизненного цикла проекта.

Возможны два вида связи – программа на языке высокого уровня вызывает процедуру на ассемблере и наоборот.

Синтаксис директивы:

имя_процедуры PROC [[модификатор_языка]язык] [расстояние]

Значения операнда «язык».

<i>Операнд «язык»</i>	<i>Язык</i>	<i>Направление передачи аргументов</i>	<i>Какая процедура очищает стек</i>
NOLANGUAGE	Ассемблер	Слева направо	Вызываемая
BASIC	Basic	Слева направо	Вызываемая
PROLOG	Prolog	Справа налево	Вызывающая
FORTRAN	Fortran	Слева направо	Вызываемая
C	C	Справа налево	Вызывающая
C++	C++	Справа налево	Вызывающая
PASCAL	Pascal	Слева направо	Вызываемая
STDCALL	-	Справа налево	Вызываемая
SYSCALL	C++	Справа налево	Вызывающая

Обоснование необходимости разработки Windows-приложений на ассемблере:

1. Язык ассемблера позволяет программисту полностью контролировать создаваемый им программный код и оптимизировать его по своему усмотрению.
2. Компиляторы языков высокого уровня помещают в загрузочный модуль программы избыточную информацию. Эквивалентные исполняемые модули, исходный текст которых написан на языке ассемблера, имеют в несколько раз меньший размер.
3. При программировании на ассемблере сохраняется полный доступ к аппаратным ресурсам компьютера.
4. Приложение, написанное на языке ассемблера, как правило, быстрее загружается в оперативную память компьютера.
5. Приложение, написанное на языке ассемблера, как правило, обладает более высокой скоростью работы и реактивностью ответа на действия пользователя.

Минимальное приложение Windows состоит из трех частей:

1. *Главная функция.*
2. *Цикл обработки сообщений.*
3. *Оконная функция.*

Имея исходный файл Windows-приложения на языке C/C++, можно получить текст на языке ассемблера. На его основе впоследствии можно сформировать функционально эквивалентный исполняемый модуль. Необходимо дизассемблировать исполняемый модуль программы. Причем сделать это нужно тем дизассемблером, который понимает интерфейс **Win32 API**. Дизассемблированный файл можно сохранить как листинг (расширение «*.lst») и как исходный текст ассемблера (расширение «*.asm»). Файл листинга в своей левой части содержит колонку с адресами смещения команд. Все метки и символические имена в дизассемблированном тексте формируются с использованием этих смещений.

Одним из главных критериев выбора языка разработки Windows-приложения является наличие в нем средств, способных поддержать строго определенную последовательность шагов. Каркасное Windows-приложение на ассемблере содержит один сегмент данных (**.data**) и один сегмент кода (**.code**). Сегмент стека в исходных текстах Windows-приложений непосредственно описывать не нужно. Windows выделяет для стека объем памяти, размер которого задан программистом в файле с расширением «*.def».

Все символические имена в программе на ассемблере по умолчанию являются глобальными. Задание директивы **locals** включает в трансляторе механизм контроля областей видимости имен и позволяет использовать в программе локальные имена. Символическим именам, которые необходимо сделать локальными, должна предшествовать определенная последовательность не менее чем из двух символов. Эти символы задаются как параметры директивы **locals**. Если этого не сделать, то по умолчанию используется последовательность из двух символов **@@**. Блоком, в пределах которого можно объявить локальные имена, может быть не только функция, но и участок программы между двумя метками.

Flat

STDCALL

Директива **.model** задает модель сегментации **flat** и стиль генерации кода при входе в процедуры программы и выходе из них – **STDCALL**. Код загрузочного модуля, генерируемый с опцией **flat**, будет работать на микропроцессорах .386 и старше. По этой причине директиве **.model** должна предшествовать одна из директив .386, .486 или .586. Указание этой модели памяти заставляет компоновщик создавать исполняемые файлы с расширением «*.exe». В программе с плоской моделью памяти используется адресация программного кода типа **near**. Параметр **STDCALL** определяет порядок передачи параметров через стек, справа налево. Функции Win32API, используемые в программе, должны быть объявлены внешними с помощью директивы **extrn**. Это необходимо сделать для того, чтобы компилятор мог сгенерировать правильный код, так как тела функций Win32API содержатся в **dll**-библиотеках системы Windows. В соответствии с соглашениями операционной системы Windows оконная функция приложения должна быть видимой за пределами приложения, в котором она написана. Это связано с тем, что оконная функция вызывается самой операционной системой Windows при поступлении сообщения для данного приложения. Загрузчик Windows самостоятельно загружает сегментные регистры, при этом учитывается требуемая модель памяти.

Под классом окна понимается совокупность присущих ему характеристик, таких как стиль его границ, форм курсора, пиктограмм, цвет фона, наличие меню и т.д. Характеристики окна описываются с помощью специальной структуры **WND CLASS**.

WND CLASS

WNDCLASSEX – wcl

Использование в программе:

wcl WNDCLASSEX <?>

В сегменте данных определен экземпляр структуры **WNDCLASSEX** – **wcl**. Первое поле структуры **WNDCLASSEX** должно содержать длину структуры. В поле **style** можно определять стиль границ окна и его поведение при перерисовке. Значение стиля представляет собой целочисленное значение, формируемое из констант.

lpfnWndProc

hIcon и hCursor

hbrBackground

GetStockObject

В поле **lpfnWndProc** записывается адрес оконной функции. С помощью этой функции все окна, создаваемые на основе этого класса, будут обрабатывать посланные им сообщения. В поля **hIcon** и **hCursor** загружаются дескрипторы значка и курсора. После запуска приложения значок будет отображаться на панели задач и в левом верхнем углу приложения, а курсор появится в области окна. Поле **hbrBackground** должно содержать значение дескриптора кисти. Кисть представляет собой ресурс, которым закрашивается некоторый объект, т.е. фон окна приложения некоторого класса. Для получения такого дескриптора необходимо использовать функцию **GetStockObject**. В качестве параметра ей передается имя нужной кисти.

lpzMenuName

szClassName

RegisterClassExA

В поле **lpzMenuName** записывается указатель на ASCIIZ-строку с именем меню. Если меню не используется, то в поле записывается значение **NULL**. Последнее действие при описании класса окна – присвоение данному классу уникального имени. Это имя описано в виде ASCIIZ-строки в поле **szClassName** сегмента данных. После инициализации структуры необходимо зарегистрировать класс окна в системе. Это действие выполняется с помощью функции **RegisterClassExA**, которой в качестве параметра передается указатель на структуру **WNDCLASSEX**.

Оконная функция предназначена для организации адекватной реакции со стороны приложения на действия пользователя и поддержания в актуальном состоянии того окна приложения, сообщения которого она обрабатывает. Приложение может иметь несколько оконных функций. Их количество определяется количеством классов окна, зарегистрированных в системе. Когда для окна Windows-приложения появляется сообщение, операционная система Windows производит вызов соответствующей оконной функции. Сообщения в зависимости от источника их появления в оконной функции могут быть двух типов: *синхронные* и *асинхронные*.

К синхронным относятся те сообщения, которые помещаются в очередь сообщений приложения, и терпеливо ждут момента, когда они будут выбраны функцией. После этого поступившие сообщения попадают в оконную функцию, где и производится их обработка. Асинхронные сообщения попадают в оконную функцию в экстренном порядке, минуя при этом все очереди. Координацию синхронных и асинхронных сообщений осуществляет Windows.

GetMessage

DispatchMessage

Если рассматривать синхронное сообщение, то его извлечение производится функцией **GetMessage** с последующей передачей его обратно в Windows функцией **DispatchMessage**. Асинхронное сообщение, не зависимо от источника, который инициирует его появление, сначала попадает в Windows и затем в нужную оконную функцию.

Windows требует, чтобы оконная функция сохраняла значения регистров **ebx**, **edi**, **esi**. Причина в том, что функция должна быть рекурсивной. Возможна ситуация, когда несколько классов окон используют одну и ту же оконную функцию для обработки сообщений, поступающих в созданные на базе этих классов окна. Имеет смысл сохранять и другие регистры, за исключением **eax**, если они задействованы в оконной функции. Исходя из требования рекурсивности, все переменные, используемые в оконной функции, должны быть локальными. Чтобы удовлетворить требованию сохранения регистров, лучше использовать соответствующее средство транслятора ассемблера – директиву **uses**. При ее использовании транслятор вставляет в начало и конец оконной функции соответствующую последовательность команд ассемблера **push**.

Центральным местом оконной функции является синтаксическая конструкция, в задачу которой входит распознавание поступившего сообщения по его типу и передача

управления на ту ветвь кода оконной функции, которая продолжает работу с параметрами сообщения. По завершению работы оконная функция формирует значение в регистре **eax**. Если сообщение обрабатывалось в оконной функции, то в **eax** необходимо поместить нулевое значение.

Для включения ресурсов в Windows-приложения, написанные на ассемблере, используется та же самая технология, что и для программ на C/C++. Ресурс – это специальный объект, используемый программой, но не определяемый в ее теле. К ресурсам относятся следующие элементы пользовательского интерфейса: значки, меню, окна диалога и т.д. Определение ресурсов производится в текстовом файле с расширением «*.rc». Подготовку этого файла можно вести двумя способами: ручным и автоматизированным. Ручной способ предполагает следующее: разработчик ресурса хорошо знает операторы, необходимые для описания конкретного ресурса; ввод текста ресурсного файла выполняется с помощью редактора, который не вставляет в вводимый текст элементы форматирования. Автоматический способ создания ресурсного файла предполагает использование специальной программы – редактора ресурсов, который позволяет визуализировать процесс создания ресурса. Конечные результаты работы этой программы могут быть двух видов: в виде текстового файла с расширением «*.rc», который впоследствии можно подвергать ручному редактированию, либо в виде двоичного файла, уже пригодного к включению в исполняемый файл приложения.

15. Защищенный режим работы микропроцессора

Любой сегмент памяти в защищенном режиме имеет следующие атрибуты:

1. Расположение сегмента в памяти.
2. Размер сегмента.
3. Уровень привилегий – определяет права данного сегмента относительно других сегментов.
4. Тип доступа – определяет назначение сегмента.

Состав перечисленных атрибутов показывает, что в защищенном режиме микропроцессор поддерживает два типа защиты – по привилегиям и по доступу к памяти. В отличие от реального режима, в защищенном режиме программа уже не может обратиться по любому физическому адресу памяти. Для этого она должна иметь определенные полномочия и удовлетворять ряду требований.

Ключевым объектом защищенного режима является специальная структура – дескриптор сегмента, который представляет собой 8-байтовый дескриптор непрерывной области памяти. Любая область памяти, которая логически может являться сегментом данных, стека или кода, должна быть описана таким дескриптором. Все дескрипторы собираются в одну из трех дескрипторных таблиц. В какую именно таблицу должен быть помещен дескриптор, определяется его назначением. Адрес, по которому размещаются дескрипторные таблицы, может быть любым: он хранится в специально предназначенном для этого адреса сегментном регистре.

Сегментные регистры можно разделить на три группы:

1. 4 регистра управления;
2. 4 регистра системных адресов;
3. 8 регистров отладки.

В состав сегментных регистров микропроцессоров **Pentium** введены изменения:

1. задействован ранее зарезервированный регистр управления **cr4**;
2. введена группа **MSR**-регистров (модельно-зависимые регистры), назначение и возможности которых, привязаны к конкретной модели микропроцессора.

Регистры управления. В группу регистров управления входят пять регистров **cr0-cr4**. Эти регистры предназначены для общего управления системой. Регистры управления доступны только программам с уровнем привилегий 0. Хотя микропроцессор имеет 5 регистров управления, доступны из них только 4. **cr1** исключается, потому что его функции пока не определены, он зарезервирован для будущего использования.

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

pe – разрешение защищенного режима работы. Состояние этого флага показывает в каком из двух режимов реальном (**pe=0**) или защищенном (**pe=1**) работает микропроцессор в данный момент;

mp – наличие сопроцессора. Всегда равен 1;

ts – переключение задач. Процессор автоматически устанавливает этот бит при переключении на выполнение другой задачи;

am – маска выравнивания. Этот бит разрешает (**am=1**) или запрещает (**am=0**) контроль выравнивания;

cd – запрещение кэш-памяти. С помощью этого бита можно запретить или разрешить использование кэш-памяти первого уровня;

pg – разрешение (**pg=1**) или запрет (**pg=0**) страничного преобразования.

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

Регистр **cr3** также используется при страничной организации памяти. Это так называемый регистр каталога страниц первого уровня. Он содержит 20-битный физический базовый адрес каталога страниц текущей задачи.

Регистр **cr4** содержит признаки, в основном разрешительного характера, которые характеризуют те или иные архитектурные элементы. Устанавливая те или иные биты в регистре **cr4**, можно включать и отключать поддержку этих свойств.

Регистры системных адресов

Эти регистры называют регистрами управления памятью. Они предназначены для защиты программ и данных в мультизадачном режиме работы микропроцессора. При работе в защищенном режиме адресное пространство делится на:

1. **глобальное** – общее для всех задач;
2. **локальное** – отдельное для каждой задачи.

В защищенном режиме любой запрос к памяти, как со стороны операционной системы, так и со стороны прикладных программ должен быть санкционирован. Микропроцессор аппаратно контролирует доступ программ к любому адресу в оперативной памяти. Для получения доступа целевой адрес, к которому хочет получить доступ программа, должен быть описан для программы. Это означает что участок физической памяти, содержащий нужный адрес, должен быть описан с помощью некоторого дескриптора сегмента.

Программе, которая желает использовать данный участок памяти, должен быть сообщен указатель на соответствующий дескриптор в одной из дескрипторных таблиц. В защищенном режиме меняется роль сегментного регистра – теперь он содержит уже не адрес, а индекс в таблице дескрипторов сегментов. Но само назначение сегментных регистров не меняется – они по-прежнему указывают на сегменты команд, данных и стека, но делают это, используя принципиально другие механизмы.

Размер сегмента в защищенном режиме может достигать 4Гбайт, то есть занимать все возможное физическое пространство памяти. Выведение информации о базовом адресе сегмента и его размере на уровень микропроцессора позволяет аппаратно контролировать работу программ с памятью и предотвращать обращения по несуществующим адресам, либо адресам, находящимся вне предела, разрешенного полем размера сегмента **limit**.

Сегменты неравноправны в правах доступа к ним. Три аппаратно поддерживаемые дескрипторные таблицы:

1. **GDT – глобальная дескрипторная таблица**. Это основная общесистемная таблица, к которой допускается обращение со стороны программ, обладающих достаточными привилегиями. Расположение таблицы в памяти произвольно. В таблице могут содержаться следующие типы дескрипторов:

- дескрипторы сегментов кодов программ;
- дескрипторы сегментов данных программ;
- дескрипторы стековых сегментов программ;
- дескрипторы **TSS** – сегменты состояния задач;
- дескрипторы для таблиц **LDT**;
- шлюзы вызова;
- шлюзы задач.

2. Таблица **LDT – локальная дескрипторная таблица**. Для любой задачи в системе может быть создана своя дескрипторная таблица. Тем самым адресное пространство задачи локализуется в пределах установленных набором дескрипторов

таблицы. Расположение таблицы в памяти также произвольно. В таблице могут содержаться следующие типы дескрипторов:

- дескрипторы сегментов кодов программ;
- дескрипторы сегментов данных программ;
- дескрипторы стековых сегментов программ;
- шлюзы вызова;
- шлюзы задач.

3. Таблица **IDT** – *дескрипторная таблица прерываний*. Содержит дескрипторы специального типа, которые определяют местоположение программ обработчиков всех видов прерываний. Элементы данной таблицы называются шлюзами. Шлюзы бывают трех типов – шлюзы задач, шлюзы прерываний, шлюзы ловушек.

16. Обработка прерываний в защищенном режиме

Обработка прерываний в защищенном режиме отличается от обработки в реальном режиме так же сильно, как и сам защищенный режим отличается от реального. В защищенном режиме другое распределение номеров векторов прерываний и принципиально иным является сам механизм обработки прерываний.

Системные обработчики прерываний, для предотвращения конфликтов, выполняют действия по идентификации источника прерывания и лишь затем непосредственно обработку прерывания.

Ключевыми компонентами в схеме обработки прерываний являются дескрипторная таблица прерываний **IDT** и системный регистр **idtr**. При возникновении прерывания от источника с номером **N** микропроцессор, находясь в защищенном режиме, выполняет следующие действия:

1. Определяет местонахождение таблицы **IDT**, адрес и размер которой содержится в регистре **idtr**.
2. Складывает значение адреса, по которому размещена **IDT**, и значение **n*8**. По данному смещению в таблице **IDT** должен находиться 8-байтовый дескриптор, определяющий местоположение процедуры обработки прерывания.
3. Переключается на процедуру обработки прерывания.

Прерывания и исключения можно разделить на несколько групп:

сбой;
ловушка;
аварийное завершение.

Сбой (ошибка) – прерывание или исключение, при возникновении которого в стек записываются значения регистров **cs:ip**, указывающие на команду, вызвавшую данное прерывание. Это позволяет, получив доступ к сегменту кода, исправить ошибочную команду в обработчике прерывания, и, вернув управление программе, фактически осуществить ее рестарт.

Ловушка – прерывание или исключение, при возникновении которого в стек записываются значения регистров **cs:ip**, указывающие на команду, следующую за командой, вызвавшей данное прерывание. Так же как и в случае ошибок возможен рестарт

программы. Для этого необходимо лишь исправить в обработчике прерывания соответствующие код или данные, послужившие источником ошибки. После этого перед возвратом управления нужно скорректировать значение **ip** в стеке на длину команды, вызвавшей данное прерывание.

Аварийное завершение – прерывание, при котором информация о месте его возникновения недоступна или неполна, и поэтому рестарт практически невозможен, если только данная ситуация не была запланирована заранее.

Микропроцессор однозначно определяет, какие прерывания являются сбоями, какие ловушками, а какие аварийным завершением, потому что алгоритмы программ-обработчиков существенно различаются.

Шлюз – дескриптор в таблице прерываний **IDT**, предназначен для указания точки входа в программу обработки прерывания. В дескрипторной таблице прерываний **IDT** могут содержаться шлюзы 3 типов:

шлюзы ловушки;

шлюзы прерывания;

шлюзы задачи.

Физически микропроцессор отличает их по значению в поле типа и содержимому остальных полей.

В общем случае, для того, чтобы сделать возможным обработку прерываний в защищенном режиме, необходимо выполнить следующие действия:

1. инициализировать таблицу **IDT**;
2. составить процедуры обработчиков прерываний;
3. запретить аппаратные прерывания;
4. перепрограммировать контроллер прерываний **i8259A**;
5. загрузить регистр **IDTR** адресом и размером таблицы **IDT**;
6. перейти в защищенный режим;
7. разрешить обработку прерываний.

17. MMX – технология микропроцессоров

Модель целочисленного **MMX** - расширения.

Основу модели **MMX** - расширения составляют 2 компоненты: программная и аппаратная. Основа программной компоненты – система команд **MMX** - расширения (57 команд) и 4 новых типа данных. **MMX** - команды являются естественным дополнением основной системы команд микропроцессора. Основным принципом их работы является одновременная обработка нескольких единиц однотипных данных одной командой – **Single Instruction Multiple Data (SIMD)**.

Основа аппаратной компоненты – восемь регистров сопроцессора. Регистры сопроцессора стека имеют размерность 80 бит, когда регистры сопроцессора играют роль **MMX** - регистров, то доступными являются их младшие 64 бита. При работе стека сопроцессора в режиме **MMX** - расширения он рассматривается не как стек, а как обычный регистровый массив с произвольным доступом. Регистровый стек сопроцессора не может одновременно использоваться и по своему прямому назначению и как **MMX** - расширение.

Важное отличие **MMX** - команд от обычных команд процессора в том, как они реагируют на ситуации переноса и займа. В случае выхода значения результата за пределы операнда, в нем фиксируется максимальное или минимальное значение. Такой принцип формирования результата называется арифметикой с насыщением.

Не все трансляторы ассемблера поддерживают **MMX** - команды. В этом случае необходимо подключить файл **iammx.inc**. Включив этот файл директивой **include** в начало программы можно использовать все **MMX** - команды в определенном формате.

<i>Типы MMX - данных</i>	<i>Диапазон граничных значений</i>
Байт без знака	0...255
Слово без знака	0...65535
Двойное слово без знака	0...4294967295
Байт со знаком	-128...127
Слово со знаком	-32768...32767
Двойное слово со знаком	-2147483648...2147483647

MMX-команды выполняются в том же режиме процессора, что и команды с плавающей запятой. Поэтому при выполнении всех **MMX** - команд (кроме **EMMS**) «портится» слово состояния регистров с плавающей запятой. Доступны следующие **MMX** - команды (обозначения: **mm** - **MMX**-регистр; **m32**, **m64** - память объема 32 и 64 бит соответственно; **imm** - непосредственный операнд; **ir32** - целочисленный регистр):

EMMS

Команда обеспечивает переход процессора от выполнения **MMX** - команд к исполнению обычных команд с плавающей запятой: она устанавливает значение 1 во всех разрядах слова состояния.

PADDB mm, mm/m64

PADDW mm, mm/m64

PADD mm, mm/m64

Команды складывают элементы данных (байты, слова или двойные слова) входного и выходного операнда. Если сумма выходит за границу допустимого диапазона, то по правилам циклической арифметики избыток отсчитывается от другой границы диапазона. «Переноса» единицы из одного элемента данных в другой не происходит.

PADDSB mm, mm/m64

PADDSW mm, mm/m64

Команды складывают элементы данных (байты или слова) входного и выходного операнда. Если сумма выходит за граничное значение допустимого диапазона, то результатом считается это граничное значение.

PADDUSB mm, mm/m64

PADDUSW mm, mm/m64

Команды складывают элементы данных (байты или слова) входного и выходного операнда. Если сумма выходит за граничное значение допустимого диапазона, то результатом считается это граничное значение.

PSUBB mm, mm/m64

PSUBW mm, mm/m64

PSUBD mm, mm/m64

Команды вычитают элементы данных (байты, слова или двойные слова) входного операнда из элементов данных выходного операнда. Если результат выходит за границу допустимого диапазона, то по правилам циклической арифметики соответствующее число единиц отсчитывается от другой границы диапазона. «Переноса» единицы из одного элемента данных в другой не происходит.

PSUBSB mm, mm/m64

PSUBSW mm, mm/m64

Команды вычитают элементы данных (байты или слова) входного операнда из элементов данных выходного операнда. Если разность выходит за граничное значение допустимого диапазона, то результатом считается это граничное значение.

PSUBUSB mm, mm/m64

PSUBUSW mm, mm/m64

Команды вычитают элементы данных входного операнда из элементов данных выходного операнда. Если разность выходит за граничное значение допустимого диапазона, то результатом считается это граничное значение.

PSLLW mm, mm/m64/imm

PSLLD mm, mm/m64/imm

PSLLQ mm, mm/m64/imm

Команды выполняют сдвиг элементов данных (16-, 32- или 64-разрядных слов) в выходном операнде на число бит, задаваемое входным операндом. Освободившиеся младшие разряды заполняются нулями.

PSRAW mm, mm/m64/imm

PSRAD mm, mm/m64/imm

Команды выполняют сдвиг элементов данных (16- или 32-разрядных слов) в выходном операнде на число бит, задаваемое входным операндом. Если сдвигается положительное число, то освободившиеся старшие разряды заполняются нулями, а если отрицательное, то единицами.

PSRLW mm, mm/m64/imm

PSRLD mm, mm/m64/imm

PSRLQ mm, mm/m64/imm

Команды выполняют сдвиг элементов данных (16-, 32- или 64-разрядных слов) в выходном операнде на число бит, задаваемое входным операндом. Освободившиеся старшие разряды заполняются нулями.

PAND mm, mm/m64

Команда вычисляет поразрядное логическое «И» своих операндов.

PANDN mm, mm/m64

Команда вычисляет обращение, поразрядное «НЕ», выходного операнда, а затем поразрядное логическое «И» между входным операндом и обращенным значением выходного.

POR mm, mm/m64

Команда вычисляет поразрядное логическое «ИЛИ» своих операндов.

PXOR mm, mm/m64

Команда вычисляет поразрядное логическое «исключающее ИЛИ» своих операндов.

PMADDWD mm, mm/m64

Команда попарно перемножает 16-разрядные слова со знаком входного и выходного операндов. Это дает четыре 32-разрядных произведения. Затем первое произведение складывается со вторым, а третье с четвертым. Суммы записываются в 32-разрядные слова выходного операнда. Если все слова на входе равны 8000h, результатом будет 80000000h.

PMULHW mm, mm/m64

Команда попарно перемножает 16-разрядные слова со знаком входного и выходного операндов, что дает четыре 32-разрядных произведения. Старшие разряды произведений записываются в 16-разрядные слова выходного операнда. Младшие разряды произведений теряются.

PMULLW mm, mm/m64

Команда попарно перемножает 16-разрядные слова со знаком входного и выходного операндов, что дает четыре 32-разрядных произведения. Младшие разряды произведений записываются в 16-разрядные слова выходного операнда. Старшие разряды произведений теряются.

PCMPEQB mm, mm/m64

PCMPEQW mm, mm/m64

PCMPEQD mm, mm/m64

Команды попарно сравнивают элементы данных (байты, 16- или 32-разрядные слова) входного и выходного операндов. Если элемент данных выходного операнда равен соответствующему элементу входного, такой элемент выходного операнда заполняется единицами. Если равенства нет, он заполняется нулями.

PCMPGTB mm, mm/m64

PCMPGTW mm, mm/m64

PCMPGTD mm, mm/m64

Команды попарно сравнивают элементы данных (байты, 16- или 32-разрядные слова со знаком) входного и выходного операндов. Если элемент данных выходного

операнда больше соответствующего элемента входного, такой элемент выходного операнда заполняется единицами; если же он не больше входного, то он заполняется нулями.

PACKSSWB mm, mm/m64

PACKSSDW mm, mm/m64

Команды преобразуют длинные элементы данных (16- и 32-разрядные слова со знаком) в более короткие (байты или 16-разрядные слова со знаком). Если исходное значение было за пределами допустимого диапазона для выходного типа данных, то результатом упаковки считается ближайшее граничное значение диапазона.

MOVD mm, mm/m32/ir32

Команда копирует 32 бита из младших разрядов **MMX** - регистра, либо из памяти, либо из целочисленного регистра в младшие 32 разряда **MMX** - регистра (старшие разряды заполняются нулями).

MOVD m32/ir32, mm

Команда копирует 32 бита из младших разрядов **MMX** - регистра в память, либо в целочисленный регистр.

MOVQ mm, mm/m64

Команда пересылки данных в **MMX** - регистр.

MOVQ mm/m64, mm

Команда пересылки данных из **MMX** - регистра.

18. Макросредства языка ассемблера

Макрокоманда представляет собой развитие механизма замены текста. С помощью макрокоманд в текст программы можно вставлять последовательности строк, которые логически могут быть данными или командами, и привязывать их к контексту места вставки. Ситуации наличия повторяющихся участков кода, либо необходимости выполнения повторяющихся действий, можно оформить в виде макрокоманд и использовать эти фрагменты в различных программах.

Макрокоманда представляет собой строку, содержащую некоторое символическое имя, предназначенную для того, чтобы быть замещенной одной или несколькими другими строками. Имя макрокоманды может сопровождаться параметрами. При возникновении необходимости использования макрокоманды и отсутствии ее, необходимо задать шаблон-описание, который называется макроопределением.

Имя_макрокоманды масго список_форм_аргум

Тело макроопределения

endm

Существует три варианта расположения макроопределения:

1. В начале исходного текста программы до сегмента кода и данных с тем, чтобы не ухудшать читабельность программы. Этот вариант применяется, когда определенные макрокоманды актуальны только в пределах одной программы.

2. В отдельном файле. Этот вариант подходит при работе над несколькими программами одной проблемной области. Чтобы сделать доступными эти макроопределения в конкретной программе, необходимо в начале исходного текста записать директиву **include**.

Include имя_файла

3. В макробиблитеке. Если есть универсальные макрокоманды, то их целесообразно записать в макробиблитеку. Сделать актуальными макрокоманды из макробиблитеки можно с помощью все той же директивы **include**. Недостаток 2 и 3 способов в том, что в исходный текст программы включаются абсолютно все макроопределения. Для исправления ситуации можно использовать директиву **purge** в

качестве операндов которой через запятую перечисляются имена макрокоманд, которые не должны включаться в текст программы.

Функционально макроопределения похожи на процедуры. Сходство их в том, что и те, и другие достаточно один раз описать, а затем вызывать их специальным образом. Отличия можно рассматривать и как достоинства, и как недостатки:

1. В отличие от процедуры, текст которой неизменен, макроопределение в процессе макрогенерации может меняться в соответствии с фактическим набором параметров. При этом коррекции могут подвергаться как операнды команд, так и сами команды.

2. При каждом вызове макрокоманды ее текст в виде макрорасширения вставляется в программу. При вызове процедуры микропроцессор осуществляет передачу управления на начало процедуры, находящейся в некоторой области памяти в одном экземпляре. Код в этом случае получается более компактным, хотя быстроедействие несколько снижается за счет необходимости осуществления переходов.

Макроопределение обрабатывается компилятором особым образом. Для того чтобы использовать описанное макроопределение в нужном месте программы, оно должно быть активизировано с помощью макрокоманды.

Имя_макрокоманды список_факт_аргум

Результатом применения в исходном тексте программы будет замещение строками из конструкции тела макроопределения. *Но это не простая замена.* Обычно макрокоманда содержит некоторый список аргументов, которыми корректируется макроопределение. Места в теле макроопределения, которые будут замещаться фактическими аргументами из макрокоманды, обозначенными с помощью формальных аргументов. В результате применения макрокоманды в программе формальные аргументы в макроопределении замещаются соответствующими фактическими аргументами. В этом заключается учет контекста. Процесс такого замещения называется макрогенерацией, а результатом этого процесса является макрорасширение.

Каждый фактический аргумент представляет строку символов, для формирования которой применяются следующие правила:

1. Строка может состоять из:
 - a. последовательности символов без пробелов, точек, запятых, точек с запятой;

b. последовательности любых символов, заключенных в угловые скобки. В этой последовательности можно указывать как пробелы, так и точки, запятые и точки с запятыми.

2. Для того чтобы указать, что некоторый символ внутри строки, представляющей фактический параметр является собственно символом, а не чем-то иным применяется специальный оператор «!». Этот оператор ставится непосредственно перед описанным выше символом и его действие эквивалентно заключению данного символа в угловые скобки.

3. если требуется вычисление в строке некоторого константного выражения, то в начале этого выражения необходимо поставить знак «%».

Прежде всего, транслятор распознает формальные аргументы по их именам в заголовке макроопределения. В процессе генерации макрорасширения компилятор ассемблера ищет в тексте тела макроопределения последовательности символов, совпадающие с теми последовательностями символов, из которых состоят формальные параметры. При обнаружении такого совпадения формальный параметр из тела макроопределения замещается соответствующим фактическим параметром из макрокоманды. Этот процесс называется подстановкой аргументов.

Синтаксис формального аргумента.

Имя_форм_аргум [:тип]

Не всегда ассемблер может распознать в теле макроопределения формальный аргумент. Это может произойти в случае, когда он является частью некоторого идентификатора. Тогда последовательность символов формального аргумента отделяют от остального контекста с помощью специального символа «&». Если в программе некоторая макрокоманда вызывается несколько раз, то в процессе макрогенерации возникает ситуация, когда в программе один идентификатор будет определен несколько раз, что будет распознано транслятором как ошибка. Для выхода из подобной ситуации применяется директива **local**. Ее необходимо задавать непосредственно за заголовком макроопределения.

Результатом работы этой директивы будет генерация в каждом экземпляре макрорасширения уникальных имен для всех идентификаторов, перечисленных в списке идентификаторов.

В теле макроопределения можно размещать комментарии и делать их обычным образом. Если применить для обозначения комментария не одну, а две идущие подряд точки с запятой, то при генерации макрорасширения этот комментарий будет исключен.

Макродирективы

С помощью макросредств ассемблера можно не только частично изменять входящие в макроопределение строки, но и модифицировать сам набор этих строк и порядок их следования. Макродирективы можно разделить на 2 группы:

1. Директивы повторения. Предназначены для создания макросов, содержащих несколько идущих подряд одинаковых последовательностей строк. При этом возможно частичная модификация этих строк.

WHILE, REPT, IRP, IRPC

2. Директивы управления процессом генерации макрорасширения. Они предназначены для управления процессом формирования макрорасширения из набора строк соответствующего макроопределения. С помощью этих директив можно как исключать отдельные строки из макрорасширения, так и вовсе прекращать процесс генерации.

EXITM

GOTO

Директивы условной компиляции

Существует 2 типа этих директив:

1. Директивы компиляции по условию – позволяют проанализировать определенные условия в ходе генерации макрорасширения и при необходимости изменить этот процесс.

2. Директивы генерации ошибок по условию – также контролируют ход генерации макрорасширения с целью генерации или обнаружения определенных ситуаций, которые могут интерпретироваться как ошибочные.

19. Архитектура сопроцессора (часть 1)

Сопроцессор представляет собой совокупность регистров, каждый из которых имеет свое функциональное назначение. В программной модели микропроцессора можно выделить три группы регистров:

1. Восемь регистров (**r0-r7**), составляющих основу программной модели сопроцессора – *стек сопроцессора*. Размерность каждого регистра 80 бит. Такая организация характерна для устройств, специализирующихся на обработке вычислительных алгоритмов.

2. Три служебных регистра:

- a. *регистр состояния сопроцессора swr (Status Word Register – регистр слова состояния)* – отражает информацию о текущем состоянии сопроцессора. В регистре **swr** содержатся поля, позволяющие определить: какой регистр является текущей вершиной стека сопроцессора, какие исключения возникли после выполнения последней команды, особенности выполнения последней команды;
- b. *управляющий регистр сопроцессора cwr (Control Word Register – регистр слова управления)* – управляет режимами работы сопроцессора. С помощью полей в этом регистре можно регулировать точность выполнения численных вычислений, управлять округлением, маскировать исключения;
- c. *регистр слова тегов twr (Tags Word Register – слово тегов)* – используется для контроля за состоянием каждого из регистров **r0-r7**. Команды сопроцессора используют этот регистр для того, чтобы определить возможность записи значений в эти регистры.

3. *Два регистра указателей данных dpr (Data Point Register) и команд ipr (Instruction Point Register)*. Они предназначены для запоминания информации об адресе команды, вызвавшей исключительную ситуацию и адресе ее операнда. Эти указатели используются для обработки исключительных ситуаций.

Все эти регистры являются доступными программно.

Регистровый стек организован по принципу кольца. Это означает, что среди всех регистров, составляющих стек, нет такого, который является вершиной стека. Все регистры стека, с функциональной точки зрения, абсолютно одинаковы и равноправны. Но, как известно, в стеке всегда должна быть вершина. И она в действительности

существует, но является плавающей. Контроль текущей вершины осуществляется аппаратно с помощью трехбитового поля **top** регистра **swr**. В поле **top** фиксируется номер регистра стека (**r0-r7**), являющегося в данный момент текущей вершиной стека.

Команды сопроцессора не оперируют физическими номерами регистров стека. Вместо этого они используют логические номера этих регистров (**st0-st7**). С помощью логических номеров реализуется относительная адресация регистров стека сопроцессора. По мере записи в стек, указатель вершины движется по направлению к младшим номерам физических регистров (уменьшается на единицу). Если текущей вершиной является **r0**, то после записи очередного значения в стек сопроцессора, его текущей вершиной станет физический регистр **r7**.

Для повышения гибкости разработки и использования подпрограмм нежелательно привязывать их по передаваемым параметрам к аппаратным ресурсам (физическим номерам регистров сопроцессора). Гораздо удобнее задавать порядок следования передаваемых параметров в виде логических номеров регистров.

Логическая нумерация регистров сопроцессора, поддерживаемая на уровне системы команд, реализует эту идею. При этом не имеет значения, в какой физический регистр стека сопроцессора были помещены данные перед вызовом подпрограммы, определяющим является только порядок следования параметров в стеке. По этой причине подпрограмме важно знать уже не место, а только порядок размещения передаваемых параметров в стеке.

Процессор и сопроцессор, являясь двумя самостоятельными вычислительными устройствами, могут работать параллельно. Но этот параллелизм касается только их внутренней работы над исполнением очередной команды. Оба процессора подключены к общей системной шине и имеют доступ к одинаковой информации. Иницирует процесс выработки очередной команды всегда основной процессор. После выработки команда попадает одновременно в оба процессора. Любая команда сопроцессора имеет код операции, первые пять бит, которого имеют значение 11011. Когда код операции начинается этими битами, то основной процессор по дальнейшему содержимому кода операции выясняет, требует ли данная команда обращения к памяти. Если это так, то основной процессор формирует физический адрес операнда и обращается к памяти, после чего содержимое ячейки памяти выставляется на шину данных. Если обращение к памяти не требуется, то основной процессор заканчивает работу над данной командой и приступает к декодированию следующей команды из текущего входного командного потока.

Что же касается сопроцессора, то выбранная команда попадает в него одновременно с основным процессором. Сопроцессор, определив по первым пяти битам, что очередная команда принадлежит его системе команд, начинает ее исполнение. Если команда требовала операнд в памяти, то сопроцессор обращается к шине данных за чтением содержимого ячейки памяти, которое к этому моменту представлено основным процессором.

20. Архитектура сопроцессора (часть 2)

Если входной поток содержит последовательность из нескольких команд сопроцессора, то, очевидно, что процессор, в отличие от сопроцессора, проскочит их очень быстро, чего он не должен делать, так как он обеспечивает внешний интерфейс для сопроцессора. Эта и другие ситуации приводят к необходимости синхронизации между собой работы двух процессоров. В моделях микропроцессоров, начиная с **i486**, подобная синхронизация выполняется командами **wait/fwait**, которые введены в алгоритм работы большинства команд сопроцессора.

В общем случае следует воспринимать сопроцессор как набор дополнительных регистров, для работы с которыми предназначены специальные команды.

Регистр состояния swr.

Регистр **swr** отражает текущее состояние сопроцессора, после выполнения последней команды. Структурно **swr** состоит из:

1. 6 флагов исключительных ситуаций.
2. Бита **sf (Stack Fault)** – ошибка работы стека сопроцессора. Бит устанавливается в единицу, если возникает одна из трех исключительных ситуаций. **IE (Invalid operation Error)** – недействительная операция. **UE (Underflow Operation)** – ошибка антипереполнения. Возникает, когда результат слишком мал. **PE (Precision Error)** – ошибка точности. Устанавливается, когда сопроцессору приходится округлять результат из-за того, что его точное представление невозможно.

3. Бита **es (Error Summary)** – суммарная ошибка работы сопроцессора. Бит устанавливается в единицу, если возникает любая из шести исключительных ситуаций:

- a. **IE** (описано выше);
- b. **UE** (описано выше);
- c. **PE** (описано выше);
- d. **DE (Denormalized operand Error)** – денормализованный операнд;
- e. **ZE (devide by Zero Error)** – ошибка деления на ноль;
- f. **OE (Overflow Error)** – ошибка переполнения. Возникает в случае выхода порядка числа за максимально допустимый диапазон.

4. Четырех битов **c0-c3 (Condition Code)** – кода условия. Назначение этих битов аналогично флагам в регистре **eflags** основного процессора – отразить результат выполнения последней команды сопроцессора.

5. Трехбитного поля **top**. Поле содержит указатель регистра текущей вершины стека.

Почти половину регистра **swr** занимают флаги исключительных ситуаций. Внутренние прерывания возникают в ходе работы текущей программы и делятся на синхронные (по команде **int**) и асинхронные (исключения или особые случаи). Таким образом, исключения – это разновидность прерываний, с помощью которых процессор информирует программу о некоторых особенностях ее реального исполнения. Сопроцессор также обладает возможностью возбуждения подобных прерываний. Все возможные исключения сведены к шести типам, описанным выше. Сопроцессор умеет самостоятельно реагировать на многие из них. Это так называемая *обработка исключений по умолчанию*. Для того чтобы «заказать» сопроцессору обработку такого исключения, его необходимо замаскировать. Такое действие выполняется с помощью установки в единицу бита в управляющем регистре сопроцессора **cwr**.

Регистр управления cwr

Регистр управления работой сопроцессора определяет особенности обработки численных данных. Он состоит из:

- a. шести масок исключений;
- b. поля управления точностью **pc (Precision Control)**;
- c. поля управления округлением **rc (Rounding Control)**.

Шесть масок предназначены для маскирования исключительных ситуаций, возникновение которых фиксируется с помощью шести бит регистра **swr**. Если какие-либо биты исключений в регистре **cwr** установлены в единицу, то это означает, что соответствующие исключения будут обрабатываться самим сопроцессором. Если для какого-либо исключения в соответствующем бите масок исключений регистра **cwr** содержится нулевое значение, то при возникновении исключения этого типа будет возбуждено прерывание. Операционная система должна содержать обработчик этого прерывания.

Поле управления точностью **pc** предназначено для выбора длины мантиссы. Возможные значения в этом поле:

- a. **pc=00** – длина мантиссы 24 бита;
- b. **pc=10** – длина мантиссы 53 бита;
- c. **pc=11** – длина мантиссы 64 бита.

По умолчанию значение поля устанавливается **pc=11**.

Поле управления округлением **rc** позволяет управлять процессом округления чисел в работе сопроцессора. Необходимость операции округления может появиться в ситуации, когда после выполнения очередной команды сопроцессора получается непредставимый результат. Установив одно из значений поля **rc**, можно выполнить округление в необходимую сторону.

Возможны следующие значения в полях регистра **cwr**:

- a. **00** – значение округляется к ближайшему числу;
- b. **01** – значение округляется в меньшую сторону;
- c. **10** – значение округляется в большую сторону;
- d. **11** – производится отбрасывание дробной части. Используется для приведения значения к форме, которая может использоваться в операциях целочисленной арифметики.

Регистр тегов twr

Регистр тегов представляет собой совокупность двухбитовых полей. Каждое двухбитовое поле соответствует определенному физическому регистру стека и характеризует его текущее состояние. Изменение состояния любого регистра стека отражается на содержимом соответствующего этому регистру поля регистра тега. Возможны следующие значения в полях регистра тега:

- a. **00** – регистр стека сопроцессора занят допустимым нулевым значением;
- b. **01** – регистр стека сопроцессора содержит нулевое значение;
- c. **10** – регистр стека сопроцессора содержит одно из специальных численных значений, за исключением нуля;
- d. **11** – регистр пуст и в него можно производить запись.

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Лабораторная работа №1

Основные команды Ассемблера

Цель работы:

1. Изучить этапы разработки программы на языке АССЕМБЛЕРА.
2. Изучить основы работы с отладчиком OllyDbg.

Описание работы:

Разработка программы на языке ассемблера состоит из трех этапов:

- а) разработка алгоритма программы и запись его на языке ассемблера;
- б) трансляция исходного текста в машинный код и компоновка;
- в) отладка программы.

Текст программы на языке ассемблера может быть набран в любом текстовом редакторе.

Важно использовать формат файла «*.TXT» или подобный ему, без специального форматирования. Нельзя использовать форматы «*.DOC» или «*.RTF». Предпочтительно набирать текст программы во встроенных редакторах NC, FAR, Total Commander и пр. Файл должен быть сохранен с расширением «*.ASM».

В листинге 1 приведен исходный текст программы с комментариями. Наберите программу в редакторе и сохраните ее под именем «lab01_1.asm» в C:\WORK.

Листинг 1.

.386	; используются регистры и
	; команды i386
.model flat,stdcall	; плоская модель памяти
.code	; начало сегмента кода
start:	; точка входа в программу
mov eax, 2	; занести 2 в eax
add eax, 20	; добавить к eax 20
ret	; вернуться в ОС
end start	; завершение программы

Транслируйте программу командой **ml /c /coff lab01_1.asm**. Если текст программы набран правильно, вы увидите примерно следующее сообщение:

Microsoft (R) Macro Assembler Version 6.14.8444

Copyright (C) Microsoft Corp 1981-1997.

All rights reserved.

Assembling: lab01_1.asm

Если в программе есть ошибки, они будут указаны в формате «имя_файла.asm(номер_строки): номер_ошибки: Описание_ошибки».

В результате работы ассемблера будет получен объектный файл **lab01_1.obj**. Для того чтобы получить исполняемый файл нужно использовать компоновщик **link**. Результатом работы компоновщика является создание исполняемого файла с расширением «***.EXE**» или «***.COM**». Для создания исполняемого файла выполните команду «**link /SUBSYSTEM:CONSOLE lab01_1.obj**». Вы увидите примерно следующее сообщение:

Microsoft (R) Incremental Linker Version 5.12.8078 Copyright (C) Microsoft Corp 1992-1998.

All rights reserved.

В каталоге появится файл **lab01_1.exe**. Запустите полученный исполняемый файл командой **lab01_1**. Если все сделано правильно, на экране ничего не будет отображено и вы снова увидите приглашение командной строки.

Для отладки программы будет использоваться отладчик **OllyDbg** (см. *рисунок*). Загрузите программу в отладчик командой **ollydbg lab01_1**.

1 – окно с исходной программой в дизассемблированном виде. Пошаговую отладку можно производить прямо в этом окне; строка с текущей командой подсвечивается;

2 – окно регистров микропроцессора, отражающего текущее содержимое регистров. Обратите внимание на регистр флагов (сверху вниз под регистрами общего назначения);

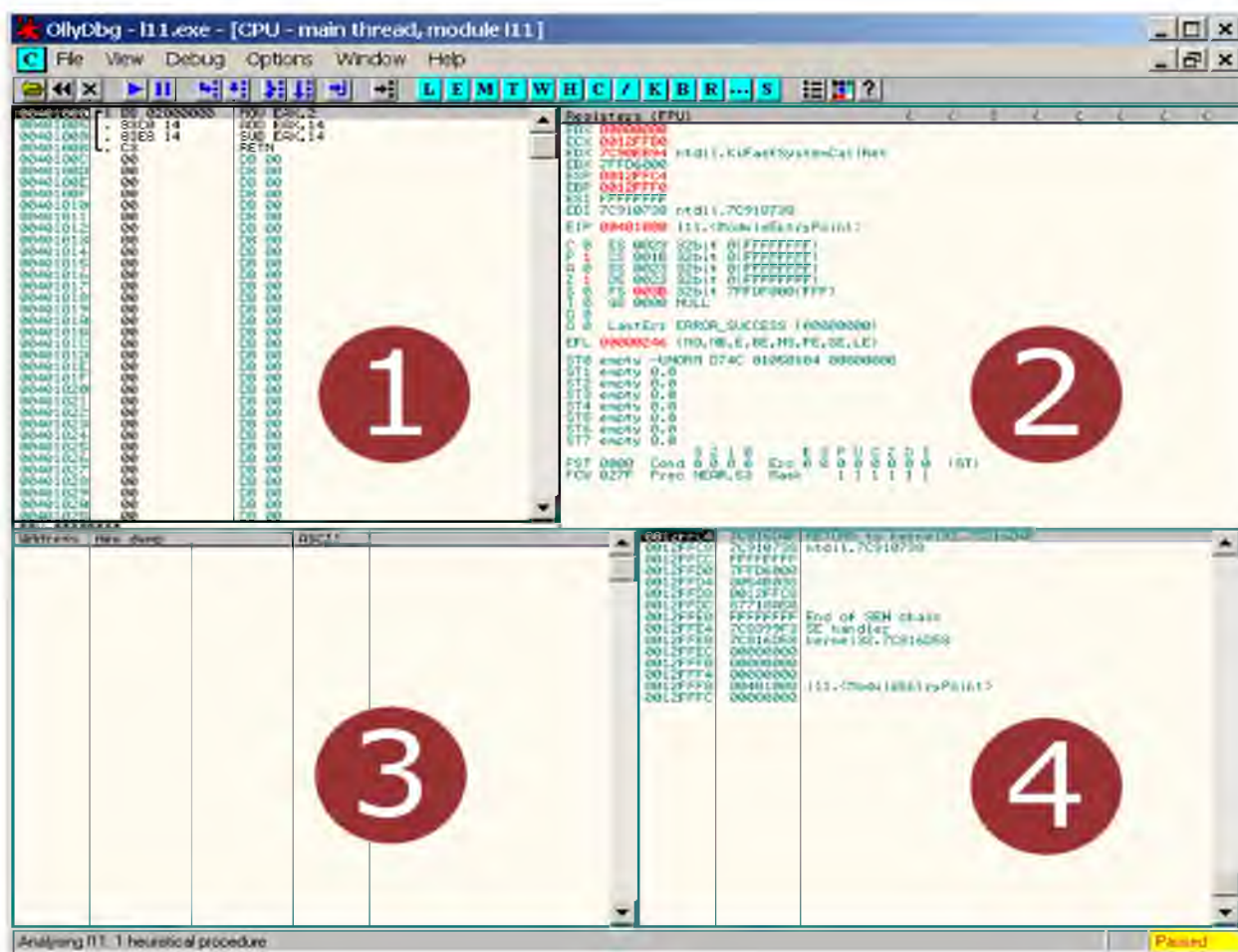
3 – окно дампа оперативной памяти, отражающее содержимое области памяти по адресу, который формируется из компонентов, указанных в левой части окна. В окне можно увидеть содержимое произвольной области памяти, а также выбрать тип отображения содержимого памяти;

4 – окно стека, отражающего содержимое памяти, выделенной для стека. Адрес области стека определяется содержимым регистров **SS** и **ESP**.

Для пошагового выполнения программы используйте клавишу «**F8**». Если в программе встречаются команды перехода в процедуры или прерывания и нужно проследить их выполнение по шагам, используется клавиша «**F7**». В нашем случае удобнее использовать «**F8**».

Для консольного приложения сразу создается окно, в котором вы сможете увидеть результаты работы (если используется вывод) или ввести данные. Выполните программу в пошаговом режиме и убедитесь в правильности ее работы (проконтролировать по состоянию регистров).

Наберите программу из листинга 2 и сохраните ее под именем lab01_2.asm в C:\WORK.



Листинг 2

.386

model flat, stdcall

option casemap:none

include c:\windows\include\kernel32.inc

includelib c:\windows\lib\kernel32.lib

.data

stdout dd ?

msg db "Hello, world! ", 0dh, 0ah

cWritten dd?

.code

start:

invoke GetStdHandle, -11

mov stdout, eax

invoke WriteConsoleA, stdout, ADDR msg, \; символ\ можно использовать для
переноса строки

sizeof msg, ADDR cWritten, 0

invoke ExitProcess. 0

end start

Создайте исполняемый файл и загрузите его в отладчик. Выполните программу в отладчике. Вернитесь к началу, выполнив сброс (**Ctrl+F2**).

Выполните программу в пошаговом режиме.

Индивидуальные задания

1. Проанализируйте принцип работы программы.
2. Прокомментируйте каждую строку листинга 2.

Указания к оформлению отчета

Отчет оформляется в виде текстового файла и должен содержать:

фамилию, имя, группу и вариант студента;

номер задания, ответ на задание (если требуется);

комментарий к выполнению задания (если требуется).

Лабораторная работа №2

Арифметические операции

Цель работы:

Изучить арифметические операции ассемблера.

Программирование арифметических выражений в языке Ассемблер происходит через некоторые команды, такие как: mul, div, sub, add. Эти команды называются командами арифметических операций.

Mul – команда умножения. Она умножает регистр ax на тот операнд, что стоит после него. Результат заносится в ax.

Div – команда деления. Делит регистр ax на операнд, находящийся после него. Результат заносится в ax.

Add – команда сложения. Складывает два числа. Результат заносится в первый регистр.

Sub – команда вычитания. Вычитает два числа. Результат заносится в первый регистр.

Пример:

Написать программу на ассемблере вычисления выражения:

$a \cdot e / b \cdot d \cdot e$

где:

$a = 5$

$b = 27$

$c = 86$

$e = 1986$

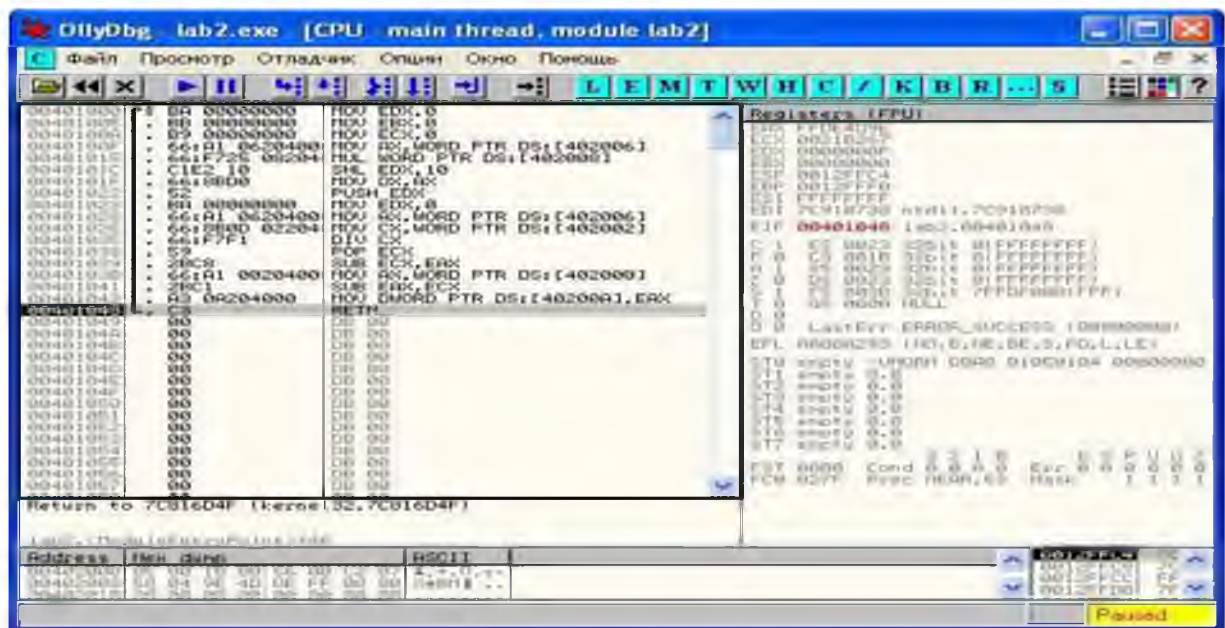
$d = 1112$

Листинг программы:

```
.686;                директива определения системы команд микропроцессора
.model flat,stdcall; задание линейной модели памяти
.data;              директива определения данных
_a dw 5;            запись в 16-разрядную константу памяти с именем _a числа 5
_b dw 27
_c dw 86
_e dw 1986
_d dw 1112
```

res dw 0;	резервирование памяти для сохранения переменной res
code;	директива начала сегмента команд
start:	
mov edx,0;	очистка регистров
mov ebx,0;	очистка регистров
mov ecx,0;	очистка регистров
mov ax,_e;	в регистр ax заносим число _e
mul _d;	множим _e и _d
SHL edx,16;	делаем сдвиг на 16
mov dx,ax	
push edx;	бросаем значение в стек
mov edx,0	
mov ax,_e	
mov cx,_b	
div cx;	делим ax на cx
pop ecx;	достаем из стека значения
sub ecx,eax;	отнимаем
mov ax,_a	
sub eax,ecx	
mov res, eax	
ret;	возвращение управление ОС
end start;	окончание программы с именем start

Результат работы программы показаны на рисунке.



Индивидуальные задания

1.	$y = \begin{cases} ax^2 - bx + c, & \text{если } x \leq 5; \\ ax - b, & \text{если } x > 5; \\ x^2(a - b) + c, & \text{если } x \leq 5. \end{cases}$
2.	$y = \begin{cases} ax - bz, & \text{если } \min(a, b) \leq 0; \\ \frac{a}{x} - \frac{b}{z}, & \text{если } \min(a, b) > 0. \end{cases}$
3.	$y = \begin{cases} \frac{ax}{ax - 5}, & \text{если } ax \leq 5; \\ ax^2 - 5x, & \text{если } ax > 5; \\ 5x^2 - ax, & \text{если } ax \leq 5; \end{cases}$
4.	$y = \begin{cases} \frac{abc}{x}, & \text{если } x \leq 7; \\ 7a^2 - bc, & \text{если } x > 7; \\ \frac{a - b + c}{x - 4}, & \text{если } x \leq 7. \end{cases}$
5.	$y = \begin{cases} \frac{3b - b^2 - 5}{b - 3}, & \text{если } b \in \mathbb{R}; \\ 3, & \text{если } b \in \mathbb{R}; \\ \frac{3b - b^2 - 5}{b - 3}, & \text{если } b \in \mathbb{R}; \end{cases}$
6.	$y = \begin{cases} x^2 - 4, & \text{если } x \leq 0; \\ x^2 - 4, & \text{если } 0 < x \leq 2; \\ 8, & \text{если } x > 2. \end{cases}$
7.	$y = \begin{cases} \frac{a^2 - 4x - 2}{b}, & \text{если } x \leq 5; \\ (a^2 - 17), & \text{если } x > 5; \\ x^2 - 4x - 2b, & \text{если } x \leq 5. \end{cases}$

8.	$y = \frac{a^2}{b-2} - 14 \sqrt{4x}, \text{ если } b \leq c ;$ $0, \text{ если } b > c .$
9.	$y = \frac{a-2}{c}, \text{ если } x \leq 10;$ $3 \sqrt{x^2 - c}, \text{ если } 10 < x \leq 13;$ $a^3, \text{ если } x > 13.$
10.	$y = \frac{100}{x^2 - ab}, \text{ если } x \leq 10;$ $\sqrt{ab^2 - \frac{a}{b}}, \text{ если } 10 < x \leq 0;$ $\frac{x^2 - ab - b^3}{a - bx}, \text{ если } x > 0.$
11.	$\frac{ay - bz}{x}, \text{ если } x \leq 0;$ $y = a^3 - 4b, \text{ если } x > 0;$ $\frac{a}{x} - \frac{y}{b}, \text{ если } x > 0.$
12.	Переменной k присвоить номер четверти плоскости, в которой расположена точка с координатами (x,y). (xy > 0)
13.	$y = \frac{a^2 - ax - bc}{abc - \frac{(b-c)x}{a}}, \text{ если } \max(a,b,c) \leq 5;$ $\frac{(b-c)x}{a}, \text{ если } \max(a,b,c) > 5.$
14.	$\frac{ax}{b}, \text{ если } x \leq 5;$ $y = ax^2 - b, \text{ если } x > 5;$ $\frac{ax^3}{b}, \text{ если } x > 5.$
15.	$x^2, \text{ если } \sqrt{x} \leq 2^2 - 4;$ $y = 16 - x, \text{ если } \sqrt{x} > 2^2 - 4;$ $ax^2 - 5x - 4, \text{ если } \sqrt{x} \leq 2^2 - 4.$

a	b	2
x	z	1
a^2		
16.	y	$\frac{(x-a)^2}{b^2}, \text{ если } x \leq 10;$ $\frac{(x-b)^2}{a^2}, \text{ если } x > 10.$
17.	y	$a^2 b^2 x, \text{ если } x^3 \leq 64;$ $3x^3, \text{ если } x^3 > 64;$ $\frac{ab^3}{2}, \text{ если } x^3 \leq 64.$
18.	y	$\max(a, b, c), \text{ если } a \leq b \leq c;$ $abc, \text{ если } a \leq b \leq c;$ $\min(a, b, c), \text{ если } a \leq b \leq c.$
19.	y	$mx - n, \text{ если } x \leq m;$ $\frac{n^2 - x^2}{x^2}, \text{ если } x > m;$ $(n - m)^2, \text{ если } x \leq m.$
20.	y	$mx - nz, \text{ если } x \leq z \leq 5;$ $\frac{25xz}{a^2 + b^2}, \text{ если } x \leq z \leq 5;$ $mx - nz$

Указания к оформлению отчета

Отчет оформляется в виде текстового файла и должен содержать:

- фамилию, имя, группу и вариант студента;
- номер задания, ответ на задание (если требуется);
- комментарий к выполнению задания (если требуется).

Лабораторная работа №3

Форматирование вывода и команды передачи управления

Цель работы:

1. Изучить функцию форматирования вывода.
2. Изучить команды передачи управления (безусловные и условные переходы).

1. Для форматирования вывода числовых данных используется функция `wsprintf`.

У функции переменное число параметров. Рассмотрим первые три.

`wsprintf ADDR buffer_for_string, ADDR szformat, number`

`ADDR buffer_for_string` - адрес начала строки для хранения результата форматирования;

`ADDR szformat` - адрес начала строки формата;

`Number` - параметр, который подставляется в строку формата.

Листинг 1

```
.386

.model flat, stdcall
option casemap:none

include c:\windows\include\windows.inc
include c:\windows\include\user32.inc
include c:\windows\include\kernel32.inc
includelib c:\windows\lib\user32.lib
includelib c:\windows\lib\kernel32.lib
bufsize equ 12

.data

fmt db "Number = %2d",0
buf db bufsize dup(?)
crlf db 0dh,0ah
stdout dd ?
cWritten dd ?

.code

start:
```



```

invoke GetStdHandle, STD_OUTPUT_HANDLE
mov stdout, eax
mov edx, 1
mov ecx, 10
nxt:
push ecx
push edx
invoke wsprintf, ADDR buf, ADDR fmt, edx
invoke WriteConsoleA, stdout, ADDR buf, \
bufsize, ADDR cWritten, NULL
invoke WriteConsoleA, stdout, ADDR crlf, \
2, ADDR cWritten, NULL
pop edx
inc edx
pop ecx
loop nxt
invoke ExitProcess, 0
end start

```

Для форматирования используются стандартные форматы (аналогично используемым в функции printf языка C). Числа с плавающей точкой форматировать таким образом нельзя.

В таблице приведены операторы условного перехода, зависящие от флагов. Их можно использовать для определения фактов переноса, переполнения и т.д.

Таблица – Команды условного перехода и флаги

Типы операндов	Мнемокод команды условного перехода	Критерий условного перехода	Значения флагов для осуществления перехода
Любые	je	операнд_1 = операнд_2	zf = 1
Любые	jne	операнд_1 <> операнд_2	zf = 0
Со знаком	jl/jnge	операнд_1 < операнд_2	sf <> of
Со знаком	jle/jng	операнд_1 <= операнд_2	sf <> of or zf = 1
Со знаком	jg/jnle	операнд_1 > операнд_2	sf = of and zf = 0
Со знаком	jge/jnl	операнд_1 >= операнд_2	sf = of
Без знака	jb/jnae	операнд_1 < операнд_2	cf = 1
Без знака	jbe/jna	операнд_1 <= операнд_2	cf = 1 or zf=1
Без знака	ja/jnbe	операнд_1 > операнд_2	cf = 0 and zf = 0
Без знака	jae/jnb	операнд_1 >= операнд_2	cf = 0

В листинге 2 приведена программа, определяющая и выводящая на экран большее из двух чисел. Числа заданы переменными в сегменте данных.

Листинг 2

.386

```
.model flat, stdcall
option casemap:none
include c:\windows\include\windows.inc
include c:\windows\include\user32.inc
include c:\windows\include\kernel32.inc
includelib c:\windows\lib\user32.lib
includelib c:\windows\lib\kernel32.lib

bufsize equ 8

.data
fmt db "Max=%d",0
buf db bufsize dup(?)
crlf db 0dh,0ah
stdout dd ?
cWritten dd ?
a dd 100
b dd 210

.code
start:
invoke GetStdHandle, STD_OUTPUT_HANDLE
mov stdout, eax
mov eax, a
cmp eax, b
jb _b
mov edx, eax
jmp print
```

```
_b:
    mov edx, b
print:
    invoke wsprintf, ADDR buf, ADDR fmt,edx
    invoke WriteConsoleA, stdout, ADDR buf,\
bufsize, ADDR cWritten, NULL
    invoke WriteConsoleA, stdout, ADDR crlf,\
2, ADDR cWritten, NULL

    invoke ExitProcess, 0
end start
```

Индивидуальные задания

1. Довести программу до логического завершения из листинга 2 так, чтобы она определяла наибольшее число и выводила в зависимости от результата проверки одно из трех сообщений: «а больше b», «а меньше b», «а равно b».
2. Написать программу, выводящую максимальное значение из 3 чисел (числа задаются переменными в сегменте данных).

Указания к оформлению отчета

Отчет оформляется в виде текстового файла и должен содержать:

- фамилию, имя, группу и вариант студента;
- номер задания, ответ на задание (если требуется);
- комментарий к выполнению задания (если требуется).

Лабораторная работа №4

Основы работы с числами с плавающей точкой

Цель работы:

Научиться работать с вещественными числами (числа с плавающей запятой).

Краткая теория

Чтобы повысить точность и максимально компактно расположить вещественное число в памяти компьютера, были придуманы числа с плавающей точкой. В старшие биты стали записывать порядок числа (Нормализованная запись числа). Размер порядка числа известен заранее (зависит от типа данных) и занимает намного меньше места, чем могла бы занять целая часть числа.

В младшие биты стали записывать мантиссу – нормализованную экспоненциальную форму числа без запятой. Таким образом, в пределах мантиссы точка может как бы «плавать», то есть её расположение зависит от порядка числа.

Но как заставить эту точку плавать? Этим занимается процессор, то есть аппаратная часть компьютера. Во многих современных процессорах даже есть специальные команды для операций над числами с плавающей точкой (но об этом позже). В большинстве компьютеров используются именно числа с плавающей точкой (а не с фиксированной), потому что это позволяет экономить память и получать большую точность.

Вспомним алгоритм представления вещественного числа в памяти компьютера:

1. Перевести число из Р-ичной системы в двоичную.
2. Представить двоичное число в нормализованной экспоненциальной форме.
3. Рассчитать смещённый порядок числа.
4. Разместить знак, порядок и мантиссу в соответствующие разряды.

Первый шаг мы уже сделали и получили двоичное представление целой и дробной части числа 3,14:

$$3 = 11b$$

$$0,14 = 0,00100011b$$

То есть число 3,14 в двоичном виде равно:

$$3,14 = 11,00100011b$$

Теперь преобразуем это число в нормализованную экспоненциальную форму:

$$11,00100011b = 1,100100011b \times 2^1$$

Теперь рассчитаем смещённый порядок (предположим, что для хранения порядка у нас используется 5 бит). Тогда исходные данные:

$$\text{ИП} = 1 \text{ (у нас } 2 \text{ в степени } 1)$$

$$k = 5$$

$$\text{СП} = \text{ИП} + 2^{k-1} - 1 = 1 + 2^{5-1} - 1 = 1 + 16 - 1 = 16$$

Записываем знак числа, порядок и мантиссу в соответствующие разряды:

Знак	Порядок	Мантисса
0	10000	0010001100

Как видите, в мантиссе у нас младшие два разряда – это нули. Эти разряды нами не используются, но при желании мы бы могли их использовать и тем самым повысить точность.

А теперь давайте спустимся с небес на землю. Все приведённые нами примеры являются упрощёнными. В реальных машинах обычно для чисел с плавающей точкой используются числа с большим количеством разрядов. Но реальные числа мы здесь рассматривать не будем, тем более что представление данных может отличаться в зависимости от процессора. Для первого знакомства информации достаточно. Возможно, что эту тему я расширю в будущих изданиях книги. А пока, если хотите знать больше – изучайте стандарт IEEE 754, который реализован во всех x86-совместимых процессорах. Переходим непосредственно к Ассемблеру.

Микропроцессор для работы с ЧПТ использует математический сопроцессор (MC). В MC есть 8 регистров для хранения чисел, обозначаемых ST0-ST7. Эти регистры образуют стек. Таким образом, при загрузке числа в регистр командой **fld** оно помещается на вершину стека в регистр ST0 (или просто ST).

Команды пересылки данных:

fld – загрузка в регистр ЧПТ;

fild – загрузка в регистр целого числа;

fstp – извлечение из стека и запись в переменную ЧПТ;

fistp – извлечение из стека и запись в переменную целого числа.

Арифметические команды

Сопроцессор использует шесть основных типов арифметических команд:

- Fxxx – первый операнд берется из верхушки стека (источник), второй - следующий элемент стека. Результат выполнения команды записывается в стек;
- Fxxx память – источник берется из памяти, приемником является верхушка стека ST(0). Указатель стека ST не изменяется, команда действительна только для операндов с одинарной и двойной точностью;
- Fixxx память – аналогично предыдущему типу команды, но операндами могут быть 16- или 32-разрядные целые числа;
- Fxxx ST, ST(i) – для этого типа регистр ST(i) является источником, а ST(0) - верхушка стека - приемником. Указатель стека не изменяется;
- Fxxx ST(i), ST – для этого типа регистр ST(0) является источником, а ST(i) - приемником. Указатель стека не изменяется;
- FxxxP ST(i), ST – регистр ST(i) - приемник, регистр ST(0) - источник. После выполнения команды источник ST(0) извлекается из стека.

Строка «xxx» может принимать следующие значения:

- ADD – Сложение;
- SUB – Вычитание;
- SUBR - Обратное вычитание, уменьшаемое и вычитаемое меняются местами;
- MUL – Умножение;
- DIV – Деление;
- DIVR - Обратное деление, делимое и делитель меняются местами.

Кроме основных арифметических команд имеются дополнительные арифметические команды:

- FSQRT - Извлечение квадратного корня;
- FRNDINT - Округление до целого;
- FABS - Вычисление абсолютной величины числа;
- FCHS - Изменение знака числа.

По команде FSQRT вычисленное значение квадратного корня записывается в верхушку стека ST(0).

Команда RNDINT округляет ST(0) в соответствии с содержимым поля RC управляющего регистра.

Команда FABS вычисляет абсолютное значение ST(0). Аналогично, команда FCHS изменяет знак ST(0) на противоположный.

Для подготовки к выводу на экран ЧПТ можно использовать функцию FpuFLtoA, преобразующую ЧПТ в строку.

Формат функции следующий:

FpuFLtoA Num, DIGS, BUF – константы;

Num – адрес числа;

DIGS – число знаков после запятой;

BUF – адрес буфера для хранения строки.

Константы – одна или несколько констант, указывающих вид числа (SRC1_REAL – вещественное).

Трансцендентные команды математического сопроцессора

•FCOS Вычисление $\cos(x)$;

•FSIN Вычисление $\sin(x)$.

Аргумент функций должен находиться в st(0), результат сохраняется там же.

Индивидуальные задания

Задание 1. Вычислите значение выражения

$$1) \quad (x - y) \cdot \frac{z}{x}$$

$$2) \quad (y - x) \cdot z \cdot z \cdot x$$

$$3) \quad \frac{(z - x)}{y} \cdot \frac{z}{x}$$

$$4) \quad (x - z) \cdot y \cdot \frac{z}{x}$$

$$5) \quad (x - y) \cdot y$$

$$6) \quad (y - z) \cdot x \cdot y \cdot x$$

$$7) \quad \frac{(z - x)}{z} \cdot \frac{y}{x}$$

$$8) \quad \frac{(z - x)}{y} \cdot x \cdot y$$

9) $(y - z) - z - x - x$

10) $\frac{(y - y)}{z} - z - x$

при $x = 749.05$

$y = 900.40$

$z = 355.31$

Задание 2. Вычислите значение выражения

1) $\sqrt{\left| \sin\left(\frac{\frac{x}{y}}{y - x}\right) \right|}$

2) $\sqrt{\left| \sin\left(\frac{x - y}{\left| \cos\left(\frac{y}{x}\right) \right|}\right) \right|}$

3) $\frac{\sqrt{\cos(x - y)}}{y}$

4) $\sqrt{\left| \frac{\cos\left(\frac{x}{y}\right)}{\sin(y)} \right|}$

5) $\sqrt{\cos(x)}$

6) $\sqrt{\left| \sin\left(\frac{x}{y}\right) \right| \left| \cos(y - x) \right|}$

7) $\sqrt{\left| \cos(x - y) \right| - \frac{y}{x}}$

8) $\sqrt{\left| \sin(x) \right| \left| \cos(y - x) \right|}$

9) $\sqrt{\left| \sin(x) \right| \cos(y)}$

10) $\sqrt{\left| \sin(x) \right| \cos(y)}$

где $x = 19.76$

$y = 10.23$

Указания к оформлению отчета

Отчет оформляется в виде текстового файла и должен содержать:

- фамилию, имя, группу и вариант студента;
- номер задания, ответ на задание (если требуется);
- комментарий к выполнению задания (если требуется).

Лабораторная работа №5

Директивы управления потоком. Основы работы с массивами

Цель работы:

1. Изучить работу директив, управляющих потоками.
2. Изучить и применить на практике работу с массивами на языке низкого уровня программирования.

Для организации ветвления и циклов можно использовать специальные директивы ассемблера для управления потоком. При этом директивы при ассемблировании будут преобразованы в обычные команды проверки условий и переходов.

Пример 1 – Условный оператор

```
.IF условие  
оператор1  
_ELSE  
оператор2  
_ENDIF
```

Блок «оператор1» выполняется, если условие истинно, блок «оператор2» - если ложно.

Листинг 1. Определить $a < b$ или $a > b$

```
.386  
_model flat, stdcall  
option casemap:none  
  
include c:\windows\include\windows.inc  
include c:\windows\include\user32.inc  
include c:\windows\include\kernel32.inc  
include c:\windows\include\fpw.inc  
includelib c:\windows\lib\user32.lib
```

```
includelib c:\windows\lib\kernel32.lib
```

```
includelib c:\windows\lib\fpu.lib
```

```
BSIZE equ 30
```

```
.data
```

```
a dd 10
```

```
b dd 20
```

```
stdout dd ?
```

```
cWritten dd ?
```

```
buf db BSIZE dup (?)
```

```
fmt db "a %c b",13,10,0
```

```
.code
```

```
start:
```

```
invoke GetStdHandle, STD_OUTPUT_HANDLE
```

```
mov stdout, eax
```

```
mov eax,a
```

```
.IF eax < b
```

```
mov edi,"<"
```

```
.ELSE
```

```
mov edi,">"
```

```
.ENDIF
```

```
invoke sprintf, ADDR buf, ADDR fmt,edi
```

```
invoke WriteConsoleA, stdout, ADDR buf,\
```

```
BSIZE, ADDR cWritten, NULL
```

```
invoke ExitProcess, 0
```

```
end start
```

```
invoke ExitProcess, 0
```

```
end start
```

Пример 2 - Циклы типа WHILE .. DO и REPEAT ... UNTIL

.WHILE условие

операторы

.ENDW

.REPEAT

операторы

.UNTIL условие

2. Массив в ассемблере можно определить, например, следующим образом

```
m1 dd 10 dup(0);      массив из 10 элементов (32 бита каждый),  
                      ;инициализированных нулями  
m2 dd 10,20,30,50,100; массив из 5 элементов, инициализированных явно
```

Листинг 2. Работа с массивами

.386

.model flat, stdcall

option casemap:none

include c:\windows\include\windows.inc

include c:\windows\include\user32.inc

include c:\windows\include\kernel32.inc

include c:\windows\include\fpu.inc

includelib c:\windows\lib\user32.lib

includelib c:\windows\lib\kernel32.lib

includelib c:\windows\lib\fpu.lib

BSIZE equ 7

.data

m1 dd 10 dup(0)

```
m2 dd 10,20,30,50,100
```

```
stdout dd ?
```

```
cWritten dd ?
```

```
buf db BSIZE dup (?)
```

```
fmt db "%4d",13,10,0
```

```
.code
```

```
start:
```

```
invoke GetStdHandle, STD_OUTPUT_HANDLE
```

```
mov stdout, eax
```

```
mov edi,0; в - индекс массива
```

```
mov eax,1; всего 10 элементов, счетчик цикла
```

```
;заполним массив m1 квадратами чисел от 1 до 10.
```

```
.WHILE eax <= 10
```

```
push eax
```

```
mul eax
```

```
mov m1[edi],eax
```

```
add edi,4
```

```
pop eax
```

```
inc eax
```

```
.ENDW
```

```
; выведем на экран массив m2
```

```
mov edi,0
```

```
mov ecx,0
```

```
.REPEAT
```

```
push ecx
```

```
mov eax,m2[edi]
```

```
invoke wsprintf, ADDR buf, ADDR fmt,eax
```

```
invoke WriteConsoleA, stdout, ADDR buf,\
```

```
BSIZE, ADDR cWritten, NULL
```

```
add edi,4
```

```
pop ecx  
inc ecx  
_UNTIL ecx == 5  
  
invoke ExitProcess, 0  
end start
```

Индивидуальные задания

Задание 1. Определите максимальное и минимальное значения в массиве из 25 элементов. Массив, найденные элементы и их номера выведите на экран. Массив задать явно.

Задание 2. Отсортируйте массив из 17 элементов по возрастанию. Исходный и отсортированный массив выведите на экран. Массив задать явно, использовать алгоритм пузырьковой сортировки.

Указания к оформлению отчета

Отчет оформляется в виде текстового файла и должен содержать:

- фамилию, имя, группу и вариант студента;
- номер задания, ответ на задание (если требуется);
- комментарий к выполнению задания (если требуется).

Лабораторная работа №6

Условные переходы

Цель работы:

Изучить работу команд условного перехода (команды CMP и TEST).

Для организации ветвлений и более сложных чем loop циклов в ассемблере используется следующая конструкция.

Сначала производится сравнение двух чисел с помощью команды cmp:

cmp eax, ebx

Операнды у команды cmp могут быть любые, кроме двух ячеек памяти. Результат сравнения заносится в регистр флагов.

Затем на основании этого результата осуществляется условный переход на указанную метку.

jb label; jb - jump if below

Если eax < ebx, то будет произведен переход на метку label, иначе будет выполнен следующий после jb оператор.

Список операторов условного перехода

ja - jump if above (>)

jb - jump if below (<)

jg - jump if greater (>)

jl - jump if less (<)

jae - jump if above or equal (>=)

jbe - jump if below or equal (<=)

jge - jump if greater or equal (>=)

jle - jump if less or equal (<=)

je - jump if equal (==)

jne - jump if not equal (!=)

jmp - unconditional jump

above/below означают без знаковое сравнение, greater/less - знаковое:

```
mov al, 0xff
cmp al, 1
ja x;           без знаковое сравнение: 255 > 1, переход будет выполнен
mov al, 0xff
cmp al, 1
jg x;           знаковое сравнение: -1 < 1, переход не будет выполнен
```

Пример

Программа, проверяющая, является ли знаковое число «a» положительным.

```
char* istrue = "a > 0"
char* isfalse = "a <= 0"
char* answer
int a; scanf("%i", &a)
__asm {
    cmp a, 0
    jg _greater
    mov eax, isfalse
    jmp _end
    _greater:
    mov eax, istrue
    _end:
    mov answer, eax
}
printf("%s\n", answer)
```

Индивидуальные задания

Задание 1

Дана точка (x, y) и прямоугольник (x1, y1), (x1, y2), (x2, y1), (x2, y2), известно, что $x1 < x2$ и $y1 < y2$.

Проверить, находится ли точка внутри прямоугольника.

Задание 2

Определить количество корней у уравнения $ax^2 + bx + c = 0$
(варианты: нет, 1, 2, бесконечно много).

Задание 3

Перевернуть число x задом наперед и увеличить результат на 1. Числа $x_1 = 1234$, $x_2 = 987$.

Примеры:

$$1234 \rightarrow 4321 + 1 = 4322$$

$$987 \rightarrow 789 + 1 = 790$$

Указания к оформлению отчета

Отчет оформляется в виде текстового файла и должен содержать:

- фамилию, имя, группу и вариант студента;
- номер задания, ответ на задание (если требуется);
- комментарий к выполнению задания (если требуется).

Лабораторная работа №7

Процедуры

Цель работы:

Работа с процедурами (синтаксис определения, Описание, Описание языка).

Ассемблер позволяет нам описывать процедуры несколькими способами. В данной работе описываются процедуры, объявление языка процедур, использование в процедурах аргументов и переменных, сохранение регистров и вложенные процедуры.

В упрощенном виде шаблон процедуры выглядит следующим образом.

Имя_процедуры proc [uses используемые_регистры]

...

[код процедуры]

ret

имя_процедуры endp

Пример 1. Процедура, вычисляющая сумму трех целых чисел

; используются регистры: eax, ebx, ecx – значения суммируемых чисел

; eax – результат

sum2 proc

add eax,ebx

add eax, ecx

ret

sum2 endp

Листинг 1. Подсчет суммы элементов целочисленного массива

.386

.model flat, stdcall

option casemap:none

include c:\windows\include\windows.inc

include c:\windows\include\kernel32.inc

include c:\windows\include\fpw.inc

```

includelib c:\windows\lib\user32.lib
includelib c:\windows\lib\kernel32.lib
includelib c:\windows\lib\fpu.lib
BSIZE equ 10
.data
array dd 10, 15, 49, 30, 85, 100, 15, 94, 12, 34
sum dd 0
stdout dd ?
cWritten dd ?
buf db BSIZE dup (?)
fmt db "%4d",13,10,0
.code
;вычисляет сумму элементов массива 32-х разрядных целых чисел
;адрес массива – esi, количество элементов – ecx
;сумма – eax
arraySum proc
push esi;    сохраним используемые регистры
push ecx
mov eax, 0
L1:
add eax, [esi];    добавим к сумме очередное значение из массива
add esi, 4;    установим указатель на следующий элемент
loop L1
pop ecx
pop esi;    восстановим используемые регистры
ret
arraySum endp
start:
invoke GetStdHandle, STD_OUTPUT_HANDLE
mov stdout, eax
mov esi, offset array;    указатель на начало массива
mov ecx, lengthof array;    размер массива
call arraySum
mov sum, eax
invoke SetConsoleTextAttribute, \

```

```
stdout, 00000101b;           установим цвет текста и фона
                               ;старшая тетрада - фон, младшая тетрада - текст
invoke wsprintf, ADDR buf, ADDR fmt,sum
invoke WriteConsoleA, stdout, ADDR buf, \
BSIZE, ADDR cWritten, NULL
invoke SetConsoleTextAttribute, stdout, 7;       вернем цвет по умолчанию – серый
invoke ExitProcess, 0
end start
```

Если в заголовок процедуры добавить ключевое слово `uses`, то сохранение используемых регистров и их восстановление ассемблер выполнит автоматически.

Заголовок должен выглядеть так:

```
arraySum proc uses esi ecx
```

Регистры перечисляются через пробел или символ табуляции.

Индивидуальные задания:

Задание 1

Доработать программу из листинга 1 таким образом, чтобы она выводила массив на экран. Каждый элемент отобразить разным цветом (используйте номера от 1 до 10).

Задание 2

Напишите процедуру вывода на экран целого числа заданным цветом. Параметры – число и номер цвета (номер цвета фона всегда 0 – черный). Доработайте программу из задания 1 так, чтобы в ней использовалась эта процедура.

Задание 3

Напишите программу для вычисления среднего значения элементов целочисленного массива. Для определения среднего используйте процедуру с параметрами: адрес массива, число элементов массива. Программа должна выводить исходный массив и найденное среднее значение.

Задание 4

Доработайте программу из задания 3 таким образом, чтобы при выводе элементов массива использовалось цветовое разделение значений: красным цветом элементы, большие, либо равные среднему (в этом случае используйте целое среднее); желтым – остальные.

Задание 5

Напишите программу преобразования температуры по шкале Цельсия в температуру по шкале Фаренгейта для значений от 15 до 36 градусов Цельсия (указание: $1F = 32 + 1.8C$).

Задание 6

Определите сумму ряда, где x – номер вариант*3, y – номер варианта. Промежуточные значения суммы вывести на экран. Для вывода и вычисления очередного элемента ряда используйте процедуры.

Указания к оформлению отчета

Отчет оформляется в виде текстового файла и должен содержать:

- фамилию, имя, группу и вариант студента;
- номер задания, ответ на задание (если требуется);
- комментарий к выполнению задания (если требуется).

Лабораторная работа №8

Связь подпрограмм на ассемблере с программами на языке высокого уровня

Цель работы:

Изучение методов использования ассемблерных подпрограмм в программах на языках высокого уровня.

Комбинирование программ на языке высокого уровня с кодом на ассемблере используется в том случае, если присутствуют участки, которые невозможно реализовать без использования ассемблера, либо его применение значительно повысит эффективность работы.

Существуют следующие формы комбинирования программ на языках высокого уровня с ассемблером:

Использование операторов типа **inline** и ассемблерных вставок. Эта форма сильно зависит от синтаксиса языка высокого уровня и конкретного компилятора. Она предполагает, что ассемблерные коды в виде команд ассемблера или прямо в машинных командах вставляются в текст программы на языке высокого уровня. Компилятор языка распознает их как ассемблера и без изменений включает в формируемый им объектный код. Эта форма удобна, если вставлять небольшой фрагмент.

Использование внешних процедур и функций. Это более универсальная форма комбинирования. У нее есть ряд преимуществ:

1. Написание и отладку программ можно производить независимо.
2. Написанные программы можно использовать в других проектах.
3. Облегчаются модификация и сопровождение подпрограмм в течение жизненного цикла проекта.

Возможны два вида связи – программа на языке высокого уровня вызывает процедуру на ассемблере и наоборот. Синтаксис директивы

имя_процедуры PROC [[модификатор_языка]язык] [расстояние]

Таблица – Значения операнда язык

Операнд «язык»	Язык	Направление передачи аргументов	Какая процедура очищает стек
NOLANGUAGE	Ассемблер	Слева направо	Вызываемая
BASIC	Basic	Слева направо	Вызываемая
PROLOG	Prolog	Справа налево	Вызывающая
FORTRAN	Fortran	Слева направо	Вызываемая
C	C	Справа налево	Вызывающая
C++	C++	Справа налево	Вызывающая
PASCAL	Pascal	Слева направо	Вызываемая
STDCALL	–	Справа налево	Вызываемая
SYSCALL	C++	Справа налево	Вызывающая

Индивидуальные задания:

0. Подсчитать количество элементов $<x$ в массиве $[4 \times 4]$. Число x задано в сегменте данных.

1. Дано натуральное число. Найти сумму первой и последней цифры данного числа. Результат поместить в АХ. Число двухзначное.

2. Дано натуральное число. Найти сумму первой и последней цифры данного числа. Результат поместить в АХ. Число трехзначное.

3. Составить программу, которая находит количество отрицательных элементов массива $[5 \times 5]$ и вывести их.

4. Разработать программу, которая при вводе с клавиатуры двух строк определяет, сколько раз вторая строка встречается в первой.

5. Напишите программу, которая запрашивает строку символов с клавиатуры, а затем на экран выводит все символы этой строки кроме знаков препинания.

6. Среднее арифметическое отрицательное чисел массива $N \times M$ чисел (Сделать счетчик отрицательных чисел, разделить их, получить среднее арифметическое).

7. В матрице 4×4 разделить все положительные элементы главной диагонали на 2.

8. Сформировать одномерный массив $A=[i]$ ($i=1 \dots 20$), где a_i вычисляется по формулам:

$$A[i] = 2 \cdot i - 10, \text{ где } i = 0 < i < 11$$

$A[i] = i / 3$), где $i = 10 < i < 21$

9. С клавиатуры вводится матрица $N \times M$. Найти максимальный элемент в каждой строке.

Указания к оформлению отчета

Отчет оформляется в виде текстового файла и должен содержать:

- фамилию, имя, группу и вариант студента;
- номер задания, ответ на задание (если требуется);
- комментарий к выполнению задания (если требуется).

Лабораторная работа №9

Изучение команд обработки строк

Цель работы:

Научиться использовать команды обработки строк при написании ассемблерных программ.

Замечания: реализовать работу со строками при помощи команд обработки строк.

Индивидуальные задания

1. Написать процедуру копирования строки.
2. Написать процедуру объединения двух строк.
3. Дана строка. Преобразовать строчные буквы в прописные. Рассмотреть только латинский алфавит.
4. Дана строка. Преобразовать строчные буквы в прописные. Рассмотреть только русский алфавит.
5. Дана строка. Преобразовать прописные буквы в строчные. Рассмотреть только латинский алфавит.
6. Дана строка. Преобразовать прописные буквы в строчные. Рассмотреть только русский алфавит.
7. Написать процедуру, осуществляющую сравнение строк. Рассмотреть только латинский алфавит.
8. Написать процедуру, осуществляющую сравнение строк. Рассмотреть только русский алфавит.
9. Написать процедуру, выводящую строку на экран путем прямого доступа к видеопамяти.
10. Зашифровать и расшифровать исходную строку.
11. Написать процедуру копирования массива типа `char`. Во входных параметрах необходимо задать количество элементов массива.
12. Написать процедуру копирования массива типа `long`. Во входных параметрах необходимо задать количество элементов массива.
13. Обменять между собой содержимое 0-й и 1-й страниц видеопамяти.
14. Сохранить содержимое 0-й страницы видеопамяти в файле на диске.

15. Очистить экран заданным цветом путем прямого доступа к видеопамяти.

Указания к оформлению отчета

Отчет оформляется в виде текстового файла и должен содержать:

- фамилию, имя, группу и вариант студента;
- номер задания, ответ на задание (если требуется);
- комментарий к выполнению задания (если требуется).

Лабораторная работа №10

Написание собственного обработчика прерывания

Цель работы:

Научиться разрабатывать собственные обработчики аппаратных прерываний.

Индивидуальные задания:

Написать резидентную программу, которая содержит собственный ISR прерывания 9 (аппаратное прерывание клавиатуры). Обработчик должен выдавать на экран в заданную позицию экрана заданное сообщение, при нажатии на определенную клавишу. Позиция экрана, сообщение и нажатая клавиша задается согласно варианту.

Таблица - Позиция экрана, куда выдается сообщение.

1-я цифра варианта	0	1	2	3	4	5	6	7	8	9
X	4	8	17	9	20	3	4	0	7	12
Y	65	78	2	4	5	26	47	54	18	0

Таблица - Сообщение, которое выдается на экран.

2-я цифра варианта	0	1	2	3	4	5	6	7	8	9
Сообщение	Lat	Num	0	Hi	Good	P41	Pk	Mm	sk	ok

Таблица - Клавиша, по которой выдается сообщение на экран.

3-я цифра варианта	0	1	2	3	4	5	6	7	8	9
Клавиша	Tab	F1	End	Ctrl	Alt	K	BKSP	F5	F8	Del

Указания к оформлению отчета

Отчет оформляется в виде текстового файла и должен содержать:

- фамилию, имя, группу и вариант студента;
- номер задания, ответ на задание (если требуется);
- комментарий к выполнению задания (если требуется).

КУРСОВОЕ ПРОЕКТИРОВАНИЕ

Тематика курсовых работ (примерная)

1. Исследование и сравнение реализации 2-мерных массивов и стеков на языке ассемблера.
2. Исследование и сравнение реализации очередей и записей на языке ассемблера.
3. Исследование реализации MMX-технологии микропроцессоров.
4. Исследование методов и способов обработки всех видов прерываний на языке ассемблера.
5. Исследование методов реализации работы с процедурами (организация интерфейса с процедурами).
6. Исследование работы с логическими командами (логические данные; команды сдвига; работа с битовыми строками; пересылка битов).
7. Исследование цепочечных команд (пересылка цепочек; команды пересылки байтов, слов, двойных слов; сравнение цепочек; сканирование цепочек; загрузка элемента цепочки в аккумулятор; перенос элемента из аккумулятора в цепочку; ввод элемента цепочки из порта ввода/вывода).
8. Создание Windows-приложения «Калькулятор» на ассемблере.
9. Исследование обработки прерываний в защищенном режиме (обработка прерываний в защищенном режиме; шлюз ловушки; шлюз прерывания; шлюз задачи; инициализация таблицы IDT; обработчики прерываний; программирование контроллера прерываний).
10. Исследование команд работы с портом ввода/вывода (на примере).
11. Исследование и сравнение реализации объединений и записей на языке ассемблера.
12. Исследование связи ассемблера с языками высокого уровня (связь Pascal-Assembler).
13. Исследование связи ассемблера с языками высокого уровня (связь C-Assembler).
14. Исследование и сравнение реализации объединений и стеков на языке ассемблера.
15. Создание Windows-приложения «Мировое время» на ассемблере.

Курсовая работа – это самостоятельная учебно-исследовательская работа студента, выполненная под руководством преподавателя, одна из основных форм учебных занятий и форм контроля учебной работы студентов.

Важным условием подготовки высококвалифицированных специалистов является их самостоятельная творческая работа, к которой следует отнести написание курсовых работ. Курсовая работа — один из важнейших показателей результатов обучения, в определенной мере свидетельствующий о степени овладения учебным материалом, об уровне подготовленности обучающегося по той или иной дисциплине, его кругозора, грамотности и общей культуры.

Курсовая работа должна выполняться в тесной связи с реальной практикой, базироваться на конкретном фактическом материале, носить исследовательский характер и способствовать развитию творческого потенциала обучаемых (студентов).

Выполнение курсовой работы является очень важным компонентом учебного процесса. Ее написание способствует формированию у студента навыков самостоятельного научного исследования, интереса к углубленному изучению предмета – в данном случае, дисциплины «Микропроцессоры». Благодаря подобной научной деятельности углубляются и закрепляются уже полученные знания, приобретаются новые, развивается творческое мышление, вырабатываются навыки письменного изложения студентом своих мыслей с использованием специальной технической терминологии.

Тема курсовой работы определяется по выбору самого студента, желательно, после консультации с преподавателем, ведущим практические занятия или читающим лекции: это может быть как одна из предлагаемых в примерном перечне тем, так и иная (обязательно – в рамках учебной программы), в последнем случае необходимо согласие преподавателя.

При выполнении и защите курсовой работы студент должен выполнить следующие пункты:

- *выбрать тему и определить руководителя курсовой работы;*
- *получить задание на выполнение курсовой работы;*
- *провести необходимые исследования, и реализацию результатов курсовой работы;*
- *выполнить и оформить пояснительную записку к курсовой работе;*
- *представить работу на проверку руководителю курсовой работы;*
- *зарегистрировать курсовую работу на кафедре;*
- *защитить курсовую работу у руководителя.*

Особо рекомендуется обратить внимание на новейшие взгляды на анализируемую проблему, для чего нужно ознакомиться с последними вышедшими в свет монографиями, журнальными статьями и т.д.

Курсовая работа должна быть написана студентом лично, самостоятельно. Категорически запрещено переписывание первоисточников без ссылки на них: при выявлении преподавателем плагиата в работе она возвращается студенту для повторного написания.

Тема курсовой работы должна быть раскрыта, причем сам по себе размер работы не является показателем этого. Студенту необходимо грамотно, логически верно, с использованием необходимой литературы и материалов практики рассмотреть все поставленные в плане вопросы.

При выборе темы учитывается ее актуальность, интерес студента к данной теме, наличие необходимой литературы. Выбор одной и той же темы студентами в пределах одной подгруппы не допускается. Студент вправе предложить свою собственную тему исследования, отсутствующую в предложенном перечне, согласовав её с руководителем.

Структура и объем пояснительной записки курсовой работы

В пояснительную записку курсовой работы должны быть включены, в заданной последовательности, следующие структурные элементы:

- Титульный лист.
- Задание на выполнение курсовой работы.
- Реферат должен содержать:
 - сведения об объеме курсовой работы, количестве иллюстраций, таблиц, приложений, количестве использованных источников;
 - перечень ключевых слов должен включать от 5 до 15 слов или словосочетаний из текста курсовой работы, которые в наибольшей мере характеризуют ее содержание. Ключевые слова приводятся в именительном падеже и печатаются строчными буквами в строку через запятые;
 - текст реферата должен отражать цель исследования, методы исследования, полученные результаты курсовой работы, область применения результатов курсовой работы.
- Содержание должно включать наименования структурных элементов (введение; заключение; список использованных источников, приложения и др.), номера и заголовки разделов, подразделов и пунктов основной части с указанием номеров страниц, с которых начинаются эти структурные элементы, разделы, подразделы, пункты пояснительной записки курсовой работы.
- Введение должно содержать:
 - оценку современного состояния исследуемой темы;
 - основные исходные данные для выполнения курсовой работы.
- Основная часть:
 - постановка исследуемой темы;
 - выбор, анализ и оценка возможных способов исследования темы курсовой работы;
 - содержание исследования, проблемные характеристики функционирования разработанной программы.
- Заключение:
 - основные результаты, полученные при выполнении курсовой работы;
 - разработка рекомендации и исходных данных по использованию результатов исследования.

- Список используемых источников должен содержать сведения об источниках, использованных при выполнении курсовой работы.
- Приложения, в них рекомендуется включать материалы, связанные с выполненной дипломной работой, которые не могут быть включены в основную часть (части прокомментированных листингов программ, блок-схемы алгоритма работы программы и др.).

Примерный постраничный объем составных частей курсовой работы:

№ п/п	Структурные элементы работы	Объем структурных элементов курсовой работы (листов)
1	Титульный лист	1
2	Задание на выполнение курсовой работы	1 - 2
3	Реферат	1 – 2
4	Содержание	1 – 2
5	Введение	2 – 4
6	Раздел 1	3 – 7
7	Раздел 2	5 – 8
8	Раздел 3	7 – 10
9	Заключение	1 – 2
10	Список используемых источников	1 – 2
11	Приложения	не более 5

Оформление курсовой работы

Текстовая часть пояснительной записки курсовой работы должна быть напечатана на одной стороне листа белой бумаги формата А4 с размерами полей: правое – 10 мм; верхнее, нижнее – 20мм; левое 30мм.

Текст набирается на компьютере с применением программных средств Microsoft Office, нежирным шрифтом Times New Roman размером 14 пунктов через полуторный межстрочный интервал, с размером абзацного отступа: 15мм. Цвет шрифта: черный.

Математические формулы должны быть записаны с использованием редактора формул Microsoft Equation.

Иллюстрации (схемы, рисунки, диаграммы и др.) и таблицы, помещенные в текстовой части, должны быть подготовлены программными средствами MS Word или Microsoft Visio.

Основная часть пояснительной записки подразделяется на разделы и подразделы.

Разделы должны иметь порядковую нумерацию в пределах основной части пояснительной записки и должны иметь свои заголовки. Номер раздела обозначается прописной арабской цифрой, ставится перед заголовком раздела без точки между номером и заголовком раздела. Заголовок пишется с прописной буквы, после заголовка раздела точка не ставится. Номер и заголовок раздела следует набирать нежирным шрифтом и начинать с абзацного отступа по ширине поля листа без подчеркивания.

Подразделы должны иметь порядковую нумерацию в пределах одного раздела и должны иметь свои заголовки. Номер подраздела включает номер раздела и порядковый номер подраздела в разделе, обозначается прописными арабскими цифрами, разделенными точкой. Номер и заголовок подраздела следует печатать также как номер и заголовок раздела.

Структурные элементы (Титульный лист, Реферат, Содержание, Введение, Заключение, Список используемых источников, Приложения) пояснительной записки не нумеруются, после заголовков точки не ставятся.

Структурные элементы (Титульный лист, Реферат, Содержание, Введение, Заключение, Список используемых источников, Приложения) и главы пояснительной записки должны начинаться с нового листа (при этом настоятельно рекомендуется использовать команду «Вставка» - «Разрыв страницы», чтобы текст не смещался во время последующей правки).

Заголовки структурных элементов, номера и заголовки разделов должны быть напечатаны с абзацным отступом по верхнему полю листа.

Между заголовком структурного элемента и следующим за ним текстом пропускается одна строка.

Формулы и уравнения следует выделять из текста в отдельную строку и выравнивать по ширине. Выше и ниже каждой формулы или уравнения должно быть оставлено не менее одной свободной строки. Если уравнение не умещается в одну строку, то оно должно быть перенесено после математических знаков, причем знак в начале следующей строки повторяют.

Пояснение значений символов и числовых коэффициентов следует приводить непосредственно под формулой в той же последовательности, в которой они приведены в формуле.

Формулы и уравнения, помещаемые в пояснительной записке, должны нумероваться в пределах каждого раздела. Номер формулы и уравнения должен включать номер раздела и порядковый номер формулы или уравнения в разделе, должен быть записан прописными арабскими цифрами с точкой между номером раздела и порядковым номером формулы или уравнения, помещен в скобках с выравниванием по правому полю листа на уровне середины формулы или уравнения.

Иллюстрации (рисунки, графики, схемы, диаграммы и др.), помещенные в пояснительной записке, должны иметь обозначения, нумерацию и наименования.

Обозначение всех видов иллюстраций осуществляется словом «Рисунок». Рисунки подписываются снизу.

Нумерация иллюстраций должна осуществляться в пределах каждого раздела. Номер иллюстрации должен включать номер раздела и порядковый номер иллюстрации в разделе, должен быть напечатан прописными арабскими цифрами с точкой между номером раздела и порядковым номером иллюстрации в разделе. Наименование иллюстрации должно отражать его вид и содержание. Иллюстрации следует располагать непосредственно после текста, в котором они впервые упоминаются. Таблицы, помещаемые в иллюстрационной записке, должны иметь обозначения, номера и названия.

Например:

Рисунок 1.1 – Блок-схема алгоритма SPT.

Данный рисунок будет являться первым рисунком в первой главе и должен отображать блок-схему работы алгоритма SPT.

Обозначение всех видов таблиц осуществляется словом «Таблица».

Нумерация таблиц должна осуществляться в пределах каждого раздела. Номер таблицы должен включать номер раздела и порядковый номер таблицы в разделе, должен быть напечатан арабскими цифрами с точкой между номером раздела и порядковым номером таблицы в разделе. Название таблицы должно отображать её содержание.

Таблицы следует располагать непосредственно после текста, в котором она упоминается в первые, или на следующей странице.

Таблицу с большим количеством строк допускается переносить на следующий лист. При переносе части таблицы на другой лист, справа над продолжением таблицы пишут слово «Продолжение» и номер таблицы.

Например:

Таблица 2.3 – Описание основных элементов объектной модели.

Таблицы подписываются сверху.

Список используемых источников должен содержать сведения о применяемых, цитируемых и обсуждаемых в пояснительной записке законодательных, нормативных, научных, учебных и периодических изданиях. Сведения о каждом издании вносятся на основе его издательских данных.

Перечень сведений обо всех использованных изданиях оформляется в виде списка использованных источников.

Для оформления списка использованных источников необходимо выполнить следующее:

- определить перечень всех использованных источников;
- распределить и упорядочить перечень использованных источников на следующие группы:

государственные законодательные акты, упорядоченные по их важности и по датам их принятия;

государственные стандарты, упорядоченные по возрастанию номеров их обозначений;

ведомственные нормативные документы, упорядоченные по датам их принятия;

научные, учебные и периодические издания, упорядоченные в алфавитном порядке;

- провести сквозную последовательную порядковую нумерацию использованных источников всех выделенных групп, начиная с государственных законодательных актов.

Ссылки на использованные источники выполняется по их присвоенным порядковым номерам.

Страницы пояснительной записки следует нумеровать в следующем порядке:

- Титульный лист, Содержание, Задание на выполнение курсовой работы, Реферат, Содержание, Разделы основной части, Заключение, Список используемых источников и Приложения входят в общую нумерацию страниц пояснительной записки;

- номера страниц пояснительной записки проставляются, начиная со второй страницы Введения, и далее нумеруются сквозной нумерацией все листы пояснительной записки и приложения к пояснительной записке;

- страницы пояснительной записки следует нумеровать арабскими цифрами, номер страницы проставляют в центре нижней части листа без точки.

Критерии оценки курсовой работы

Оценка **«отлично»** выставляется, если тема курсовой работы раскрыта в полной мере, работа выполнена самостоятельно, содержит анализ практических проблем. Представленный материал работы свидетельствует о глубоком понимании автором рассматриваемых вопросов. Изложение материала работы отличается логической последовательностью, наличием иллюстративно-аналитического материала (таблицы, диаграммы, блок-схемы и т.д.), ссылок на литературные источники, завершается конкретными выводами.

Курсовая работа оформлена аккуратно, в соответствии с предъявленными требованиями.

Оценка **«хорошо»** выставляется, если раскрыто основное содержание темы, работа выполнена преимущественно самостоятельно, содержит анализ практических проблем. Представленный в ней материал свидетельствует о достаточно глубоком понимании автором рассматриваемых вопросов. Изложение материала работы отличается логической последовательностью, наличием иллюстративно-аналитического материала (таблицы, диаграммы, схемы и т.д.), ссылок на литературные и нормативные источники, завершается конкретными выводами. Имеются недостатки, не носящие принципиального характера. Курсовая работа оформлена аккуратно, в соответствии с предъявленными требованиями. Имеются незначительные ошибки в оформлении работы.

Оценка **«удовлетворительно»** выставляется, если тема курсовой работы раскрыта частично, работа выполнена в основном самостоятельно, содержит элементы анализа реальных проблем. Не все рассматриваемые вопросы изложены достаточно глубоко, есть нарушения логической последовательности, ограниченно применяется иллюстративно-аналитический материал (таблицы, схемы и т.д.), отсутствуют ссылки на литературные источники. В работе допущено большое количество ошибок и опечаток.

Оценка **«неудовлетворительно»** выставляется, если не раскрыта тема курсовой работы. Материал изложен неграмотно, без логической последовательности.

ФОНД ОЦЕНОЧНЫХ
СРЕДСТВ ПО
ДИСЦИПЛИНЕ

Паспорт фонда оценочных средств

Контроль освоения дисциплины «Микропроцессоры» проводится в соответствии с ПлКубГАУ 2.5.1 — 2011 «Текущий контроль успеваемости и промежуточная аттестация студентов», версия 1.0.

Текущий контроль по дисциплине «Микропроцессоры» позволяет оценить степень восприятия учебного материала и проводится для оценки результатов изучения разделов/тем дисциплины.

Текущий контроль проводится как контроль тематический (по итогам изучения определенных тем дисциплины) и рубежный (контроль определенного раздела или нескольких разделов, перед тем, как приступить к изучению очередной части учебного материала).

Реферат это краткое изложение в письменном виде содержания и результатов индивидуальной учебно-исследовательской деятельности, имеет регламентированную структуру, содержание и оформление. Его задачами являются:

1. Формирование умений самостоятельной работы студентов с источниками литературы, их систематизация;
2. Развитие навыков логического мышления;
3. Углубление теоретических знаний по проблеме исследования.

Текст реферата должен содержать аргументированное изложение определенной темы. Реферат должен быть структурирован (по главам, разделам, параграфам) и включать разделы: введение, основная часть, заключение, список используемых источников. В зависимости от тематики реферата к нему могут быть оформлены приложения, содержащие документы, иллюстрации, таблицы, схемы и т.д.

Критериями оценки реферата являются: новизна текста, обоснованность выбора источников литературы, степень раскрытия сущности вопроса, соблюдения требований к оформлению.

Оценка **«отлично»** выполнены все требования к написанию реферата: обозначена проблема и обоснована её актуальность; сделан анализ различных точек зрения на рассматриваемую проблему и логично изложена собственная позиция; сформулированы выводы, тема раскрыта полностью, выдержан объём; соблюдены требования к внешнему оформлению.

Оценка **«хорошо»** основные требования к реферату выполнены, но при этом допущены недочёты. В частности, имеются неточности в изложении материала; отсутствует логическая последовательность в суждениях; не выдержан объём реферата;

имеются упущения в оформлении.

Оценка **«удовлетворительно»** имеются существенные отступления от требований к реферированию. В частности: тема освещена лишь частично; допущены фактические ошибки в содержании реферата; отсутствуют выводы.

Оценка **«неудовлетворительно»** тема реферата не раскрыта, обнаруживается существенное непонимание проблемы или реферат не представлен вовсе.

Рекомендуемая тематика рефератов по курсу приведена в таблице 2.

Таблица — Темы рефератов, рекомендуемые к написанию при изучении дисциплины «Микропроцессоры».

№ п/п	Наименование темы реферата
1	Основы вызова функций WinAPI.
2	Директивы определения данных и представления данных в памяти.
3	B CD-числа.
4	Директивы управления потоком.
5	Использование процедур.
6	Программирование драйверов устройств.
7	Основы работы с числами с плавающей точкой (ЧПТ).
8	Работа с математическим сопроцессором.
9	MMX - технология микропроцессоров.
10	Денормализация операнда.
11	Немаскируемая обработка исключений.
12	Обработчики прерываний.
13	Регистры управления, тегов и состояния.
14	Программирование контроллера прерываний.
15	Каркасное Windows-приложение на ассемблере.

Перечень вопросов к экзамену (примерный)

1. Характеристики микропроцессора. Структурная схема микропроцессора.

Основные блоки.

2. Особенности реализации микропроцессоров Intel и AMD.
3. Режимы работы микропроцессора.
4. Чипсеты (наборы системной логики) для микропроцессоров Intel и AMD.

Структура чипсета.

5. Программная модель микропроцессора.
6. Регистры общего назначения. Сегментные регистры.
7. Регистры состояния и управления. Флаги.
8. Организация памяти и режимы работы микропроцессора. Понятие сегментации. Сегментированная модель памяти.

9. Формирование физического адреса в реальном режиме.
10. Жизненный цикл программы на языке ассемблера.
11. Трансляция программы. Компоновка. Отладка.
12. Система команд микропроцессора.
13. Структура машинной команды. Способы задания операндов.
14. Функциональная классификация машинных команд.
15. Директивы сегментации: стандартные и упрощенные.
16. Модели памяти при использовании упрощенных директив сегментации.
17. Простые типы данных ассемблера. Директивы определения данных.
18. Команды обмена данными. Пересылка данных.
19. Ввод-вывод в порт.
20. Организация циклов.
21. Безусловные переходы.
22. Условные переходы.
23. Работа с адресами и указателями.
24. Преобразование данных.
25. Организация стека. Команды работы со стеком.
26. Арифметические команды. Форматы целых чисел в ассемблере. BCD-числа.
27. Сложение двоичных чисел без знака. Вычитание двоичных чисел без знака.
28. Сложение двоичных чисел со знаком. Вычитание двоичных чисел со знаком.
29. Умножение двоичных чисел без знака. Деление двоичных чисел без знака.
30. Умножение двоичных чисел со знаком. Деление двоичных чисел со знаком.

31. Сложение неупакованных BCD-чисел. Вычитание неупакованных BCD-чисел.
32. Умножение неупакованных BCD-чисел. Деление неупакованных BCD-чисел.
33. Логические команды. Условные и безусловные переходы.
34. Флаги и команды условных переходов. Организация циклов.
35. Процедуры. Основные понятия.
36. Организация процедур. Передача параметров в процедуру.
37. Концепция прерываний. Классификация прерываний.
38. Внешние прерывания.
39. Внутренние прерывания.
40. Программные прерывания.
41. Обработка прерываний в реальном режиме.
42. Обработка прерываний в защищенном режиме.
43. Сложные структуры данных. Способы организации массивов.
44. Цепочечные команды.
45. Макросредства языка ассемблера.
46. Директивы компиляции по условию. Директивы генерации ошибок.
47. Ассемблер и языки высокого уровня.
48. Интерфейс с языками высокого уровня.
49. Ассемблерные вставки на C и Pascal.
50. Использование процедур на ассемблере.
51. Защищенный режим работы микропроцессора.
52. Характеристики защищенного режима работы микропроцессоров Intel.
53. Сегментные регистры и структуры данных защищенного режима.
54. Перевод микропроцессора в защищенный режим.
55. Особенности работы в защищенном режиме.
56. Ассемблер для Windows. Структура Windows – программы на языке ассемблера.
57. Программирование на ассемблере с использованием Win32 API.
58. Система команд сопроцессора.
59. Исключения сопроцессора и их обработка.
60. MMX – расширение архитектуры микропроцессора.

Вопросы к тестовым заданиям (примерные)

Критерии оценки знаний студентов при проведении тестирования

Оценка «**отлично**» выставляется при условии правильного ответа студента не менее чем 85% тестовых заданий.

Оценка «**хорошо**» выставляется при условии правильного ответа студента не менее чем 70% тестовых заданий.

Оценка «**удовлетворительно**» выставляется при условии правильного ответа студента не менее 51%.

Оценка «**неудовлетворительно**» выставляется при условии правильного ответа студента менее чем на 50% тестовых заданий.

Результаты текущего контроля используются при проведении промежуточной аттестации.

1. Язык ассемблера - это...

+: язык программирования низкого уровня. В отличие от языка машинных кодов, позволяет использовать более удобные для человека мнемонические (символьные) обозначения команд. При этом для перевода программы с языка ассемблера в понимаемый процессором машинный код требуется специальная программа, называемая ассемблером.

-: язык программирования высокого уровня. В отличие от языка машинных кодов, позволяет использовать более удобные для человека мнемонические (символьные) обозначения команд. При этом для перевода программы с языка ассемблера в понимаемый процессором машинный код требуется специальная программа, называемая ассемблером.

-: язык программирования высокого уровня, основанный на pascal. В отличие от языка машинных кодов, позволяет использовать более удобные для человека мнемонические (символьные) обозначения команд. При этом для перевода программы с языка ассемблера в понимаемый процессором машинный код требуется специальная программа, называемая ассемблером.

-: язык программирования высокого уровня, основанный на C++. В отличие от языка машинных кодов, позволяет использовать более удобные для человека мнемонические (символьные) обозначения команд. При этом для перевода программы с языка ассемблера в понимаемый процессором машинный код требуется специальная программа, называемая ассемблером.

2. Какими достоинствами обладают программы на ассемблере?

- : легким восприятием.
- +: компактностью кода.
- +: высокой скоростью.
- : малым количеством операторов.

3. Область применения ассемблера?

- : веб - приложения.
- +: драйверы устройств.
- +: приложения для встраиваемых систем.
- +: приложения, с высоким требованием к размеру кода.

4. CPU - это...

-: (*central processing unit, CPU*, дословно— центральное обрабатывающее устройство)— исполнитель машинных инструкций, часть программного обеспечения компьютера или программируемого логического контроллера; отвечает за выполнение операций, заданных программами.

+: (*central processing unit, CPU*, дословно— центральное обрабатывающее устройство)— исполнитель машинных инструкций, часть аппаратного обеспечения компьютера или программируемого логического контроллера; отвечает за выполнение операций, заданных программами.

-: Central position output.

-: (*central processing unit, CPU*, дословно— центральное обрабатывающее устройство)— исполнитель программных инструкций, часть аппаратного обеспечения компьютера или программируемого логического контроллера; отвечает за выполнение операций, заданных программами.

5. Основные функции МП - ...

- +: исполнение инструкций программ.
- : исполнение инструкций пользователя.
- +: управление и координация работы остальных компонентов ЭВМ.
- : исполнение инструкций операционной системы.

6. Архитектуры МП?

- +: CISC (Complex Instruction Set Computing).
- : MRSC (Minimal Routing Set Computing)
- +: RISC (Reduced Instruction Set Computing).
- +: MISC (Minimum Instruction Set Computing)

7. Характеристики МП?

- : Скорость.
- +: Разрядность.
- +: Частота системной шины и памяти.
- +: Система команд.

8. Из чего состоит МП?

- : Оперативная память.
- +: Устройство управления.
- +: Арифметико-логическое устройство.
- +: Микропроцессорная память.

9. Наименьшая единица памяти?

- : 1 байт.
- : 8 бит.
- : 1 кбит.
- +: 1 бит.

10. Как происходит нумерация битов в байте?

- +: 7,6,5,4,3,2,1,0.
- : 0,1,2,3,4,5,6,7. -
- : 0,7,1,6,2,5,3,4. -:
- 15, ...,5,4,3,2,1.

11. Как происходит нумерация битов в слове?

- : 1,2,3,4,5,...,15.
- : 0,1,2,3,4,5...,15.
- +: 15, ...,5,4,3,2,1.
- : 0,7,1,6,2,5,3,4.

12. Какие модели памяти поддерживают МП аппаратно?

- : Сегментскую.
- +: Сегментарную.
- : Статичную.
- +: Страничную.

13. Как реализован комментарий в ассемблере?

- : //комментарий//
- : / комментарий/
- +: -комментарий||
- : */ комментарий/*

14. Отметьте правильные сочетания операндов.

+; Регистр – регистр.

-; Память-память.

+; Память – регистр.

+; Непосредственный операнд – регистр.

15. Что обозначает сегментный регистр CS?

+; сегмент кода.

-; сегмент данных.

-; сегмент стека.

-; дополнительный сегмент.

16. Что обозначает сегментный регистр DS?

-; сегмент кода.

+; сегмент данных.

-; сегмент стека.

-; дополнительный сегмент.

17. Что обозначает сегментный регистр SS?

-; сегмент кода.

-; сегмент данных.

+; сегмент стека.

-; дополнительный сегмент.

18. Что обозначает сегментный регистр ES?

-; сегмент кода.

-; сегмент данных.

-; сегмент стека.

+; дополнительный сегмент.

19. Что обозначает флаг CF?

-; Флаг четности. В рез. пред. операции четное число единиц.

-; Вспомогательный флаг переноса, для BCD-чисел.

-; Флаг переполнения. Результат превосходит доп. величину.

+; Флаг переноса. Пред. операции произвела перенос.

20. Что обозначает флаг PF?

+; Флаг четности. В рез. пред. операции четное число единиц.

-; Вспомогательный флаг переноса, для BCD-чисел.

-; Флаг переполнения. Результат превосходит доп. величину.

-; Флаг переноса. Пред. операции произвела перенос.

21. Что обозначает флаг AF?

- : Флаг четности. В рез. пред. операции четное число единиц.
- +: Вспомогательный флаг переноса, для BCD-чисел.
- : Флаг переполнения. Результат превосходит доп. величину.
- : Флаг трассировки.

22. Что обозначает флаг ZF?

- : Флаг знака. В пред. операции отрицательный результат.
- : Вспомогательный флаг переноса, для BCD-чисел.
- +: Флаг нуля. Результат пред. операции = 0.
- : Флаг трассировки.

23. Что обозначает флаг SF?

- +: Флаг знака. В пред. операции отрицательный результат.
- : Вспомогательный флаг переноса, для BCD-чисел.
- : Флаг нуля. Результат пред. операции = 0.
- : Флаг трассировки.

24. Что обозначает флаг OF?

- : Флаг знака. В пред. операции отрицательный результат.
- +: Флаг переполнения. Результат превосходит доп. величину
- : Флаг нуля. Результат пред. операции = 0.
- : Флаг прерываний.

25. Что обозначает флаг TF?

- +: Флаг трассировки.
- : Флаг переполнения. Результат превосходит доп. величину.
- : Флаг нуля. Результат пред. операции = 0.
- : Флаг прерываний.

26. Что обозначает флаг IF?

- +: Флаг трассировки.
- : Флаг прерываний.
- : Флаг нуля. Результат пред. операции = 0.
- : Флаг четности. В рез. пред. операции четное число единиц.

27. Что обозначает флаг DF?

- : Флаг знака. В пред. операции отрицательный результат.
- +: Флаг переполнения. Результат превосходит доп. величину
- : Флаг нуля. Результат пред. операции = 0.
- : Флаг направления.

28. Микроконтроллер (MCU) - это ...

-: это аппаратно-программное устройство вычислительной техники, предназначенное для объединения аудио- и видеоконференции в многоточечный режим.

-: проект по разработке и практическому использованию универсальной компьютерной программы для численного моделирования процессов переноса различного вида излучений (нейтронов, гамма-квантов, электронов) в трёхмерных системах методом Монте-Карло.

+: микросхема, предназначенная для управления электронными устройствами. Типичный микроконтроллер сочетает в себе функции процессора и периферийных устройств, может содержать ОЗУ и ПЗУ. По сути, это однокристальный компьютер, способный выполнять простые задачи.

-: микросхема, предназначенная для управления электронными устройствами. типичный микроконтроллер сочетает в себе функции процессора, не содержит ОЗУ и ПЗУ.

29. Как реализуется параллелизм ассемблерного уровня?

+: в виде конвейера (конвейера команд, арифметического конвейера или конвейера смешанного типа).

-: в виде многопоточности.

-: в виде параллельности.

+: в виде многопроцессорной системы.

30. Какие группы конвейеров существуют?

-: Интегральные векторы.

+: Векторные конвейеры.

+: Скалярные конвейеры.

-: Производные конвейеры.

31. Какие ступени содержит целочисленный конвейер?

+: Ступень предвыборки.

-: Ступени перевыборки

+: Ступень декодирования полей команды D1.

+: Ступень исполнения EX.

32. Какой размер данных соответствует байту в ассемблере?

-: $w(16)$.

-: $d(64)$.

-: $q(8 \text{ bait})$.

+: $b(8 \text{ bit})$.

33. Какой размер данных соответствует слову в ассемблере?

- +; w(16).
- ; d(64).
- ; q(8 bait).
- ; t(10 bait).

34. Какой размер данных соответствует двойному слову в ассемблере?

- ; w(16).
- +; d(64).
- ; q(8 bait).
- ; t(10 bait).

35. Какие команды отвечают за пересылку данных?

- +; MOV.
- ; XCOG.
- +; XCHG.
- +; PTR.

36. По каким направлениям команда mov может пересылать данные?

- +; Регистр → регистр.
- +; Регистр → память.
- ; Регистр или основная память → константа.
- +; Память → регистр.

37. Какие модификаторы размера используются в ассемблере?

- +; byte ptr.
- +; word ptr.
- +; dword ptr.
- ; bit ptr.

38. Стек - это...

-; это область памяти, специально выделяемая для постоянного хранения данных программы.

+; это область памяти, специально выделяемая для временного хранения данных программы.

-; это область памяти, выделяемая для постоянного хранения данных программы.

-; это область памяти, специально образующиеся для постоянного хранения данных программы.

39. SS - это...

- ; регистр указателя стека.
- ; регистр указателя базы стека.

+: сегментный регистр стека.

-: сегмент стека.

40. Sp/esp - это...

-: регистр указателя стека.

+: регистр указателя базы стека.

-: сегментный регистр стека.

-: сегмент стека.

41. Bp/ebp - это...

+: регистр указателя стека.

-: регистр указателя базы стека.

-: сегментный регистр стека.

-: сегмент стека.

42. Назначение команды PUSH:

-: извлекает операнд из стека и записывает его в указанный регистр.

+: записывает в стек содержимое указанного регистра.

-: команда групповой записи/чтения в стек.

-: команда группового извлечения из стека.

43. Назначение команды POP:

+: извлекает операнд из стека и записывает его в указанный регистр.

-: записывает в стек содержимое указанного регистра.

-: команда групповой записи/чтения в стек.

-: команда группового извлечения из стека.

44. Назначение команды PUSHA:

-: извлекает операнд из стека и записывает его в указанный регистр.

-: записывает в стек содержимое указанного регистра.

+: команда групповой записи/чтения в стек.

-: команда группового извлечения из стека.

45. Процедура - это...

-: это дополнительная функциональная единица декомпозиции (разделения на несколько частей) некоторой задачи.

-: это дополнительная функциональная единица декомпозиции (разделения на несколько частей) некоторой задачи, предназначена для выполнения основных операций.

+: это основная функциональная единица декомпозиции (разделения на несколько частей) некоторой задачи.

-: это часть основной функции.

46. Какие директивы используют для написания процедуры?

- +; PROC.
- ; PROD.
- ; NEDP.
- +; ENDP.

47. Где можно размещать процедуру?

- +; До точки входа;
- +; После команды завершения основной программы;
- ; После точки входа;
- +; В отдельном сегменте кода.

48. Какими способами можно передавать параметры процедуре?

- +; Через регистры (непосредственно).
- +; Через регистры (как адреса).
- +; Через стек.
- ; Через память.

49. Какими способами можно вызвать процедуру?

- +; Поместить параметры в стек.
- +; Использовать стандартный вызов call.
- ; Поместить параметры в регистр.
- +; Использовать директиву invoke.

50. Какие разновидности имеет команда jmp?

- +; переход прямой короткий (в пределах от -128 до + 127 байтов);
- +; переход прямой ближний (в пределах текущего программного сегмента);
- ; переход прямой дальний (в другой программный сегмент);
- +; переход косвенный ближний;

51. Что обозначает атрибутный оператор short?

- ; прямой ближний переход.
- +; прямой короткий переход.
- ; прямой дальний переход.
- ; косвенный ближний переход.

52. Что обозначает атрибутный оператор near ptr?

- ; прямой ближний переход.
- ; прямой короткий переход.
- ; прямой дальний переход.
- +; косвенный ближний переход.

53. Что обозначает атрибутный оператор far ptr?

- : прямой ближний переход.
- : прямой короткий переход.
- +: прямой дальний переход.
- : косвенный ближний переход.

54. Что обозначает атрибутный оператор word ptr?

- : прямой ближний переход.
- : прямой короткий переход.
- : прямой дальний переход.
- +: косвенный ближний переход.

55. Что обозначает атрибутный оператор dword ptr?

- +: косвенный дальний переход.
- : прямой короткий переход.
- : прямой дальний переход.
- : прямой ближний переход.

56. Что обозначает команда and?

- : дизъюнкция.
- : строгая дизъюнкция.
- +: конъюнкция.
- : команда сдвига.

57. Что обозначает команда or?

- +: дизъюнкция.
- : строгая дизъюнкция.
- : конъюнкция.
- : команда сдвига.

58. Что обозначает команда xor?

- : дизъюнкция.
- +: строгая дизъюнкция.
- : конъюнкция.
- : команда сдвига.

59. Что обозначает команда shl?

- : сдвиг содержимого ячейки влево.
- : сдвиг содержимого ячейки вправо.
- : сдвиг содержимого ячейки в конец строки.
- +: перемещают содержимое ячейки влево или вправо.

60. Что обозначает команда Shr bx?

- : арифметический сдвиг содержимого регистра вправо.
- : арифметический сдвиг содержимого регистра влево.
- : логический сдвиг содержимого регистра влево.
- +: логический сдвиг содержимого регистра вправо.

61. Что обозначает команда Sar ax?

- +: арифметический сдвиг содержимого регистра вправо.
- : арифметический сдвиг содержимого регистра влево.
- : логический сдвиг содержимого регистра влево.
- : логический сдвиг содержимого регистра вправо.

62. Что обозначает команда Sal bx?

- : арифметический сдвиг содержимого регистра вправо.
- +: арифметический сдвиг содержимого регистра влево.
- : логический сдвиг содержимого регистра влево.
- : логический сдвиг содержимого регистра вправо.

63. Что обозначает команда Shl ax?

- : арифметический сдвиг содержимого регистра вправо.
- : арифметический сдвиг содержимого регистра влево.
- +: логический сдвиг содержимого регистра влево.
- : логический сдвиг содержимого регистра вправо.

64. Что обозначает команда INC, AX?

- +: увеличивает содержимое регистра на единицу.
- : уменьшает содержимое регистра на единицу.
- : увеличивает содержимое регистра на две единицы.
- : уменьшает содержимое регистра на две единицы.

65. Что обозначает команда DEC, AX?

- : увеличивает содержимое регистра на единицу.
- +: уменьшает содержимое регистра на единицу.
- : увеличивает содержимое регистра на две единицы.
- : уменьшает содержимое регистра на две единицы.

66. Что обозначает команда inc?

- : Уменьшение операнда.
- +: Увеличение операнда.
- : вычитание второго операнда из другого.
- : изменение знака операнда.

67. Что обозначает команда add?

- : добавления операнда.
- +: сложение двух операндов с записью результата.
- : вычитание второго операнда из другого.
- : изменение знака операнда.

68. Что обозначает команда adc?

- : добавления операнда.
- : сложение двух операндов с записью результата.
- : вычитание второго операнда из другого.
- +: сложения с учетом переноса.

69. Что обозначает команда sub?

- +: вычитание второго операнда из первого.
- : сложение двух операндов с записью результата.
- : вычитание второго операнда из другого.
- : сложения с учетом переноса.

70. Что обозначает команда sbb?

- : вычитание второго операнда из первого.
- : сложение двух операндов с записью результата.
- +: вычитания с учетом займа.
- : сложения с учетом переноса?

71. Что обозначает команда neg?

- +: изменение знака операнда.
- : сложение двух операндов с записью результата.
- : вычитания с учетом займа.
- : сложения с учетом переноса?

72. Что обозначает команда Mul?

- : команда деления.
- : сложение двух операндов с записью результата.
- +: команда умножения.
- : сложения с учетом переноса?

73. Что обозначает команда div?

- +: команда деления.
- : сложение двух операндов с записью результата.
- : команда умножения.
- : сложения с учетом переноса?

74. Что обозначает команда imul?

- : команда деления.
- : знаковое целочисленное деление.
- : команда умножения.
- +: знаковое целочисленное умножение.

75. Что обозначает команда idiv?

- : команда деления.
- +: знаковое целочисленное деление.
- : команда умножения.
- : знаковое целочисленное умножение.

76. Что обозначает команда CBW?

+: преобразует байт, содержащийся в регистре AL в слово, помещаемое в регистр AX.

-: преобразует слово, содержащееся в регистре AX в двойное слово, помещаемое в регистры DX:AX. Старшая часть значения разместится в DX, а младшая – в AX.

-: преобразует слово, содержащееся в регистре AX в двойное слово, помещаемое в регистр EAX.

-: преобразует двойное слово, содержащееся в EAX, в учетверенное слово, помещаемое в регистры EDX:EAX.

77. Что обозначает команда CWD?

-: преобразует байт, содержащийся в регистре AL в слово, помещаемое в регистр AX.

+: преобразует слово, содержащееся в регистре AX в двойное слово, помещаемое в регистры DX:AX. Старшая часть значения разместится в DX, а младшая – в AX.

-: преобразует слово, содержащееся в регистре AX в двойное слово, помещаемое в регистр EAX.

-: преобразует двойное слово, содержащееся в EAX, в учетверенное слово, помещаемое в регистры EDX:EAX.

78. Что обозначает команда CWDE?

-: преобразует байт, содержащийся в регистре AL в слово, помещаемое в регистр AX.

-: преобразует слово, содержащееся в регистре AX в двойное слово, помещаемое в регистры DX:AX. Старшая часть значения разместится в DX, а младшая – в AX.

+: преобразует слово, содержащееся в регистре AX в двойное слово, помещаемое в регистр EAX.

-: преобразует двойное слово, содержащееся в EAX, в учетверенное слово, помещаемое в регистры EDX:EAX.

79. Что обозначает команда CDQ?

-: преобразует байт, содержащийся в регистре AL в слово, помещаемое в регистр AX.

-: преобразует слово, содержащееся в регистре AX в двойное слово, помещаемое в регистры DX:AX. Старшая часть значения разместится в DX, а младшая – в AX.

-: преобразует слово, содержащееся в регистре AX в двойное слово, помещаемое в регистр EAX.

+: преобразует двойное слово, содержащееся в EAX, в учетверенное слово, помещаемое в регистры EDX:EAX.

80. На какие группы можно разделить команды MC?

+: команды пересылки данных.

+: арифметические команды.

+: команды сравнений чисел.

-: команды кодировки данных.

81. Команды пересылки данных предназначены для...?

+: загрузки чисел из оперативной памяти в числовые регистры.

-: загрузка чисел из регистров в оперативную память.

+: записи данных из числовых регистров в оперативную память.

+: копирования данных из одного числового регистра в другой.

82. Что обозначает команда fstsw?

-: загружает ЧПТ в стек FPU.

-: копирует значение из вершины стека сопроцессора в операнд-адресат.

-: меняет местами содержимое регистра st0 и другого регистра FPU.

+: чтение слова состояния в регистр или переменную.

83. Что обозначает команда fld?

+: загружает ЧПТ в стек FPU.

-: копирует значение из вершины стека сопроцессора в операнд-адресат.

-: меняет местами содержимое регистра st0 и другого регистра FPU.

-: чтение слова состояния в регистр или переменную.

Глоссарий

Адресация памяти (addressing mode) – осуществление ссылки (обращение) к устройству или элементу данных по его адресу; установление соответствия между множеством однотипных объектов и множеством их адресов; метод идентификации местоположения объекта.

Активационная запись (activation record) – область стека, заполняемая при вызове процедуры.

Ассемблер (assembly language) – язык программирования низкого уровня.

Байт (byte) – тип данных, имеющий размер 8 бит, минимальная адресуемая единица памяти.

Бит (bit) – минимальная единица измерения информации.

«Всплывающая» программа (popup program) – резидентная программа, активирующаяся по нажатию определенной «горячей» клавиши.

«Горячая» клавиша (hotkey) – клавиша или комбинация клавиш, используемая не для ввода символов, а для вызова программ и подобных необычных действий.

Дамп памяти (англ. memory dump; в Unix — core dump) — содержимое рабочей памяти одного процесса, ядра или всей операционной системы. Также может включать дополнительную информацию о состоянии программы или системы, например значения регистров процессора и содержимое стека. Многие операционные системы позволяют сохранять дамп памяти для отладки программы. Как правило, дамп памяти процесса сохраняется автоматически, когда процесс завершается из-за критической ошибки (например, из-за ошибки сегментации). Дамп также можно сохранить вручную через отладчик или любую другую специальную программу.

Двойное слово (double word) – тип данных, имеющий размер 32 бита.

Дескриптор (descriptor) – восьмибайтная структура, хранящаяся в одной из таблиц GDT, LDT или IDT и описывающая сегмент или шлюз.

Директива (directive) – команда ассемблеру, которая не соответствует командам процессора.

Драйвер (driver) – служебная программа, выполняющая функции посредника между операционной системой и внешним устройством.

Защищенный режим (protected mode) – режим процессора, в котором действуют механизмы защиты, сегментная адресация с дескрипторами и селекторами и страничная адресация.

Задача (task) – программа, модуль или другой участок кода программы, который можно запустить, выполнять, отложить и завершить.

Идентификатор (handle или identifier) – число (если handle) или переменная другого типа, используемая для идентификации того или иного ресурса.

Исключение (exception) – событие, при котором выполнение программы прекращается и управление передается обработчику исключения.

Итерация (iteration) — организация обработки данных, при которой действия повторяются многократно, не приводя при этом к вызовам самих себя (не путать с рекурсией).

Код (code) – исполнимая часть программы (обычная программа состоит из кода, данных и стека).

Команда перехода (branch)– команда процессора, которая нарушает естественный порядок исполнения команд, вынуждая выбирать и исполнять последующие команды с произвольно заданного адреса.

Компилятор (compiler) – программа, преобразующая текст, написанный на понятном человеку языке программирования, в исполнимый файл.

Конвейер (pipe) – последовательность блоков процессора, которая задействуется при выполнении команды.

Конвенция (convention) – договоренность о передаче параметров между процедурами.

Конечный автомат (finite state machine) – программа, которая может переключаться между различными состояниями и выполнять в разных состояниях разные действия.

Кэш (cache) – быстрая память, используемая для буферизации обращений к основной памяти.

Лимит (limit) – поле дескриптора (равно размеру сегмента минус 1).

Линейный адрес (linear address) — адрес, получаемый сложением смещения и базы сегмента.

Ловушка (trap) – исключение, происходящее после вызвавшей его команды.

Локальная метка (local label) – это метка, которая известна только внутри того оператора Asm, где она была определена.

(также мэйнфрейм, от англ. mainframe) — большой универсальный высокопроизводительный отказоустойчивый сервер со значительными ресурсами ввода-вывода, большим объемом оперативной и внешней памяти, предназначенный для использования в критически важных системах (англ. mission-critical) с интенсивной

пакетной и оперативной транзакционной обработкой. Основной разработчик мейнфреймов — корпорация IBM, самые известные мейнфреймы ей выпущены в рамках продуктовых линеек System/360, 370, 390, zSeries. В разное время мейнфреймы производили Hitachi, Bull, Unisys, DEC, Honeywell, Burroughs, Siemens, Amdahl, Fujitsu, в странах СЭВ выпускались мейнфреймы ЕС ЭВМ.

Модуль ядра (англ. loadable kernel module, LKM) — объект, содержащий код, который расширяет функциональность запущенного или т.н. базового ядра ОС. Большинство текущих систем, основанных на Unix и Windows, поддерживают загружаемые модули ядра, хотя они могут называться по-разному (например, kernel loadable module в FreeBSD и kernel extension в Mac OS X).

Метка (label) — идентификатор, связанный с адресом в программе.

Нить (thread) — процесс, данные и код которого совпадают с данными и кодом других процессов.

Нереальный режим (unreal mode) — реальный режим с границами сегментов по 4Гб.

Операнд (operand) — параметр, передаваемый команде процессора.

Описатель носителя (media descriptor) — байт, используемый DOS для идентификации типа носителя (обычно не используется).

Останов (abort) — исключение, происходящее асинхронно.

Отложенное вычисление (lazy evaluation) — вычисление, которое выполняется, только если реально требуется его результат.

Очередь предвыборки (prefetch queue) — буфер, из которого команды передаются на расшифровку и выполнение.

Ошибка (fault) — исключение, происходящее перед вызвавшей его командой.

Пиксель (pixel) — минимальный элемент растрового изображения.

Повторная входимость (reentrancy) — возможность запуска процедуры из обработчика прерывания, прервавшего выполнение этой же процедуры.

Подчиненный сегмент (conforming segment) — сегмент, на который можно передавать управление программам с более низким уровнем привилегий.

Препроцессор (preprocessor) — это компьютерная программа, принимающая данные на входе и выдающая данные, предназначенные для входа другой программы (например, компилятора). О данных на выходе препроцессора говорят, что они находятся в препроцессированной форме, пригодной для обработки последующими программами (компилятор).

Прерывание (interrupt) – сигнал от внешнего устройства, приводящий к прерыванию выполнения текущей программы и передаче управления специальной программе-обработчику.

Разворачивание циклов (loop unrolling) – превращение циклов, выполняющихся известное число раз, в линейный участок кода.

Реальный режим (real mode) – режим, в котором процессор ведет себя идентично 8086 – адресация не выше одного мегабайта памяти, размер всех сегментов ограничен и равен 64Кб, только 16-битный режим.

Резидентная программа (resident program) – программа, остающаяся в памяти после возврата управления в DOS.

Сегмент (segment) – элемент сегментной адресации в памяти или участок программы для DOS/Windows.

Селектор (selector) – число, хранящееся в сегментном регистре.

Секция (section) – участок программы для UNIX.

(англ. *Server* *on to serve* — *служить*, *мн. ч.*) — специализированный компьютер и/или специализированное оборудование для выполнения на нём сервисного программного обеспечения (в том числе серверов тех или иных задач).

Скан-код (scan-code) – любой код, посылаемый клавиатурой.

Слово (word) – тип данных, имеющий размер 16 бит.

Смещение (offset) – относительный адрес, отсчитываемый от начала сегмента.

Стек (англ. Stack — стопка) — абстрактный тип данных, представляющий собой список элементов, организованных по принципу LIFO (англ. last in — first out, «последним пришёл — первым вышел»).

Стековый кадр (stack frame) – область стека, занимаемая параметрами процедуры, активационной записью и локальными переменными или только локальными переменными.

Страничная адресация (pagination) – механизм адресации, в котором линейное адресное пространство разделяется на страницы, которые могут располагаться в разных областях памяти или вообще отсутствовать.

Строб – дополнительный сигнал, является подтверждением действительности других сигналов. Стробирование может осуществляться по фронту или по уровню.

Таблица переходов (jump table) – массив адресов процедур для косвенного перехода на процедуру с известным номером.

Шлюз (gate) – структура данных, позволяющая осуществлять передачу управления между разными уровнями привилегий в защищенном режиме.

DDK (от англ. Driver Development Kit) — набор из средств разработки, заголовочных файлов, библиотек, утилит, программного кода примеров и документации, который позволяет программистам создавать драйверы для устройств по определённой технологии или для определённой платформы (программной или программно-аппаратной). Название произошло от более общего термина SDK (англ. Software Development Kit), которым обозначают комплекты для разработки программ вообще, не только драйверов.

DriverPack Solution (сокр. DRPSu) — менеджер установки драйверов, предназначенный для автоматизации работы с драйверами на платформе Windows ОС. Распространяется бесплатно, по свободной лицензии GNU GPL, с открытым исходным кодом.

Список использованных источников

Основная литература:

1. Микропроцессоры: лабораторный практикум (по специальности 230700.62 – «Прикладная информатика» и 230400.62 – «Информационные системы и технологии»). / А.В. Параскевов, Д. Ю. Жмурко, С.А. Курносов, В.И. Лойко. – Краснодар: КубГАУ, 2013. – 71с.
2. Микропроцессоры: метод. рекомендации к выполнению курсовой работы / сост. А.В. Параскевов, А.Н. Бардак, А.Г. Дмитриева [и др.]. – Краснодар: КубГАУ, 2015. – 57 с.
3. <http://www.programmersclub.ru/assembler/>
4. Гуревич В.И. Микропроцессорные реле защиты. Устройство, проблемы, перспективы / В.И. Гуревич. - М.: Инфра-Инженерия, 2011. - 336 с.
5. Гусев В.Г. Электроника и микропроцессорная техника: Учебник / В.Г. Гусев, Ю.М. Гусев. - М.: КноРус, 2013. - 800 с.
6. Калашников В.И. Электроника и микропроцессорная техника: Учебник для студ. учреждений высш. проф. обр. / В.И. Калашников, С.В. Нефедов. - М.: ИЦ Академия, 2012. - 368 с.

Дополнительная литература:

1. Александров Е., Грушвицкий Р., Куприянов М. Микропроцессорные системы. – М.: – Политехника, 2008. – 543с.
2. Пильщиков В. Язык макроассемблера IBM PC. – М.: –Академия, 2008.–412с.
3. Пирогов В. Ассемблер для Windows. – М.: – Вильямс, 2007. – 522с.
4. Юров В. Ю. Ассемблер. – СПб.: Питер, 2007. – 624с.
5. К. Ирвин. Язык Ассемблера для процессоров Intel. М.:–Вильямс, 2006. 616с.
6. Корнеев В., Киселев А. Современные микропроцессоры. – СПб.: – BHV-СПб, 2007. – 448с.
7. Новиков Ю. Основы микропроцессорной техники: Курс лекций. – М.: – Интернет-университет информационных технологий, 2007. – 436с.
8. Финогенов К.Г., Рудаков П.И. Язык Ассемблера: уроки программирования. М.: – Диалог-МИФИ, 2005. – 235с.
9. Белов А.В. Самоучитель по микропроцессорной технике. – М.: – Наука и Техника, 2007. – 224с.
10. Кузин А.В., Жаворонков М.А. Микропроцессорная техника. Учебник. – М.: – Академия, 2008. – 314с.

Учебное издание

Параскевов Александр Владимирович
Бардак Алексей Николаевич

МИКРОПРОЦЕССОРЫ
Учебное пособие

В авторской редакции

Усл. печ.л. – 10