Электронный учебно-методический комплекс

ЯЗЫКИ ПРОГРАММИРОВАНИЯ

Учебная программа дисциплины
Конспект лекций. Язык С
Конспект лекций. Ассемблер
Методические указания по лабораторным работам
Методические указания по самостоятельной работе
Банк тестовых заданий в системе UniTest

Красноярск
ИПК СФУ
2008

УДК 004.438
ББК 32.973
Т39

Электронный учебно-методический комплекс по дисциплине «Языки программирования» подготовлен в рамках инновационной образовательной программы «Информатизация и автоматизированные системы управления», реализованной в ФГОУ ВПО СФУ в 2007 г.

Рецензенты:
Красноярский краевой фонд науки;
Экспертная комиссия СФУ по подготовке учебно-методических комплексов дисциплин

Титовский, С. В.

Т39 Языки программирования. Ассемблер. Версия 1.0 [Электронный ресурс] : конспект лекций / С. В. Титовский, Н. В. Титовская. – Электрон. дан. (2 Мб). – Красноярск : ИПК СФУ, 2008. – (Языки программирования : УМКД № 147-2007 / рук. творч. коллектива Ю. А. Шитов). – 1 электрон. опт. диск (*DVD*). – Систем. требования : *Intel Pentium* (или аналогичный процессор других производителей) 1 ГГц ; 512 Мб оперативной памяти ; 2 Мб свободного дискового пространства ; привод *DVD* ; операционная система *Microsoft Windows* 2000 *SP* 4 / *XP SP* 2 / *Vista* (32 бит) ; *Adobe Reader* 7.0 (или аналогичный продукт для чтения файлов формата *pdf*).

ISBN 978-5-7638-1250-3 (комплекса)
ISBN 978-5-7638-1460-6 (конспекта лекций)
Номер гос. регистрации в ФГУП НТЦ «Информрегистр» 0320802545 от 02.12.2008 г. (комплекса)

Настоящее издание является частью электронного учебно-методического комплекса по дисциплине «Языки программирования», включающего учебную программу, конспект лекций «Языки программирования. Язык С», методические указания по лабораторным работам, методические указания по самостоятельной работе, контрольно-измерительные материалы «Языки программирования. Банк тестовых заданий», наглядное пособие «Языки программирования. Презентационные материалы».

Даны общая характеристика языков ассемблера и обзор услуг операционной системы, рассмотрены основы архитектуры ЭКМ линии x86, система прерывайний x86 и макросредства ассемблера.

Предназначен для студентов направления подготовки специалистов 090102.65 «Компьютерная безопасность» укрупненной группы 090000 «Информационная безопасность».

Рекомендовано к изданию
Инновационно-методическим управлением СФУ

Редактор Т. И. Тайгина

Разработка и оформление электронного образовательного ресурса: Центр технологий электронного обучения информационно-аналитического департамента СФУ; лаборатория по разработке мультимедийных электронных образовательных ресурсов при КрЦНИТ

Подп. к использованию 01.10.2008
Объем 2 Мб
Красноярск: СФУ, 660041, Красноярск, пр. Свободный, 79

ОГЛАВЛЕНИЕ

ЛЕКЦИЯ 1
ВВЕДЕНИЕ В НИЗКОУРОВНЕВОЕ ПРОГРАММИРОВАНИЕ

План лекции

1. Обзор и общая характеристика языков программирования
2. Стандарты языков программирования.
3. Понятие низкоуровневого программирования.

1. Обзор и общая характеристика языков программирования.

В настоящее время разработаны сотни языков программирования. Однако далеко не все из них нашли широкое применение. Выбор подходящего языка программирования для решения той или иной задачи часто зависит от того, к какой предметной области она принадлежит. В таблице приведены итоги использования различных языков программирования в различных предметных областях и в разные периоды.

Таблица

Период	Область применения	Основные языки	Другие языки
1960-е годы	Обработка деловой информации	COBOL	Assembler
	Научные вычисления	FORTRAN	ALGOL, BASIC, APL
	Системная область	Assembler	JOVIAL, Forth
	Искусственный интеллект	LISP	SNOBOL
Настоящее время	Обработка деловой информации	COBOL, C++, Java, spreadsheet	C, PL/1, 4GLs
	Научные вычисления	FORTRAN, C, C++, Java	BASIC
	Системная область	C, C++, Java	Ada, BASIC, Modula
	Искусственный интеллект	LISP, Prolog	
	Издательская деятельность	TeX, Postscript, текстовые процессоры	
	Создание процессов	UNIX, shell, TCL, Perl, JavaScript	AWK, Marvel, SED
	Новые парадигмы	ML, Smalltalk	Eifell

Программные приложения, которые в 1960-е гг. интенсивно разрабатывались, можно разделить на четыре основных типа: обработка деловой информации, научные вычисления, системное программирование, системы искусственного интеллекта.

Большинство приложений для обработки деловой информации было предназначено для огромного количества данных и выполнялось на больших универсальных машинах (big iron mainframes). Класс приложений этого типа включал в

себя программы для учета поступления заказов, управления ресурсами и персоналом, а также для начисления зарплаты. Они были предназначены для считывания больших объемов данных, собранных за длительный период времени и хранящихся на магнитных лентах, и создания новых данных, обновленных в результате незначительных преобразований. Для таких приложений был разработан язык COBOL. Программисту, использовавшему COBOL, обычно требовалось несколько месяцев на создание типичного приложения.

Приложения, связанные с научными вычислениями, сводятся к нахождению решений различных математических уравнений. Они включают в себя задачи численного анализа, решения дифференциальных и интегральных уравнений, также задачи статистики. Изначально компьютеры создавались для составления баллистических таблиц во время Второй мировой войны. Здесь всегда доминировал FORTRAN. Его синтаксис был близок к математическому языку, и ученым было легко использовать его.

Системная область. Для создания операционных систем и реализации компиляторов в то время не существовало эффективного языка. Такие приложения должны были иметь доступ ко всем функциональным возможностям и ресурсам аппаратной части компьютера. Для достижения максимальной эффективности часто выбирался язык ассемблер. В некоторых проектах Министерства обороны США использовался JOVIAL — разновидность языка ALGOL, кроме того, вплоть до конца 1960-х гг. для таких приложений использовались также языки типа PL/1.

Искусственный интеллект. Искусственный интеллект в те годы был относительно новой областью исследований, и в разработке приложений в ней доминировал язык LISP. Отличительной особенностью программ, написанных на этом языке, является реализация алгоритмов, осуществляющих поиск в больших объемах данных. Например, при игре в шахматы компьютер генерирует множество потенциальных ходов, а затем в течение отведенного на один ход времени выбирает наилучший вариант.

Приложения XXI века. В свое время язык Ada был разработан с целью устранения дублирования возможностей в конкурирующих между собой языках программирования. В настоящее время ситуация гораздо более сложная, чем в 60-е гг. XX в. Сейчас мы имеем больше областей применения, для которых разработаны и реализованы специализированные языки программирования, хорошо адаптированные к решению возникающих задач. Более того, в каждой области применения существует несколько подобных языков.

Приложения для обработки деловой информации. По-прежнему основным языком в этой области остается COBOL, хотя иногда используются языки C и C++. Однако сценарий оценки возможных вариантов кардинально изменился. Электронные таблицы, используемые на персональных компьютерах, полностью реформировали эту область применения. В то время как раньше программист тратил несколько месяцев на создание обычной программы делового планирования, теперь аналитик может за несколько часов составить много таких таблиц.

Языки четвертого поколения 4GL (Fourth Generation Languages) также заняли определенную нишу в этой области. Языки 4GL – это языки, специально адаптированные под конкретные области применения обработки деловой информации. Как правило, они имеют средства для создания оконного интерфейса и простой доступ к записям базы данных. Также предусмотрены специальные возможности для создания форм заполнения стандартного бланка и генерирования красиво оформленных отчетов. Иногда компиляторы языков 4GL в качестве результата выдают программы на языке COBOL.

Средства, позволяющие вести диалог пользователя (то есть покупателя) и компании (продавца) посредством «всемирной паутины», дали толчок к развитию новой роли языков программирования. Язык Java был разработан для обеспечения конфиденциальности частной жизни пользователя, а такие языки, как Perl и JavaScript, позволяют продавцу получить от пользователя сведения, необходимые для проведения сделки.

Научные вычисления. Здесь по-прежнему FORTRAN не сдает своих позиций, хотя языки Java и *C++* вполне успешно конкурируют с FORTRAN 90.

Системная область. В этой области доминирует язык *C*, созданный в конце 1960-х гг., и его более новый вариант *C++*. Язык *C* обеспечивает очень эффективное выполнение программ и позволяет программисту получить полный доступ к операционной системе и аппаратной части. Кроме того, используются такие языки, как Modula и современный вариант BASIC. Хотя язык Ada и создавался для применения в этой области, он не получил здесь статуса основного языка.

С появлением недорогих микропроцессоров, используемых в автомобилях, микроволновых печах, видеоиграх и электронных часах, возросла необходимость в языках, позволяющих писать программы для работы в реальном времени. К таким языкам относятся *C*, Ada и *C++*.

Искусственный интеллект. Здесь по-прежнему используется LISP, хотя на смену MIT LISP 1.5 середины прошлого века пришли современные версии Scheme и Common LISP. Также развился Prolog. Оба языка признаны наиболее подходящими для задач поиска оптимального решения.

Издательская деятельность. Издательская деятельность является относительно новой областью применения языков программирования. Системы обработки текстов имеют свой собственный синтаксис входных команд и выходных файлов.

Процессы. В настоящее время для управления одной программой часто используется другая: например, для регулярного резервного копирования файлов в полночь, ежечасной синхронизации времени, автоматического ответа на электронные письма во время отпуска, автоматического тестирования программы после ее успешной компиляции и т. д. Такие операции называются процессами. В настоящее время имеется значительный интерес к разработке таких языков, в которых можно определять подобные процессы и после успешной трансляции автоматически запускать на выполнение.

В системе UNIX командный язык пользователя называется командным интерпретатором или оболочкой shell, а программы называются сценариями shell. Эти сценарии активизируются при условии выполнения некоторых допустимых условий. Кроме того, появилось множество других языков сценариев (например, для тех же целей можно использовать Perl или TCL).

Новые парадигмы программирования. Постоянно появляются и изучаются новые модели приложений. В области исследования теории типов в языках программирования используется язык ML. Хотя промышленное применение этого языка не слишком значительно, его популярность постоянно растет. Другой важный язык – Smalltalk. Хотя он также не получил широкого использования в коммерческой области, но оказал глубокое воздействие на идеологию языков. Многие из объектно-ориентированных свойств языков *C++* и Ada заимствованы из Smalltalk.

Специализированные языки, предназначенные для решения задач в различных прикладных областях, являются неиссякаемым источником новых исследований и разработок. По мере углубления наших знаний о методах компиляции и способах построения сложных систем постоянно открываются новые области применения языков программирования и возникает необходимость создания новых языков именно для этих областей.

2. Стандарты языков программирования.

Как и в любой другой области, вопросы стандартизации применительно к языкам программирования имеют огромное значение. Стандарты обычно бывают двух видов:

1. Частный стандарт. Сюда входят определения, сделанные той компанией, которая разработала язык и имеет на него авторские права. В большинстве случаев для популярных и широко используемых языков такие стандарты не работают, поскольку в этих случаях часто появляются новые реализации, усовершенствованные и несовместимые.

2. Согласительный стандарт. К нему относятся созданные специальными организациями документы, основанные на соглашении всех заинтересованных участников. Согласительный стандарт является основным способом обеспечения единообразия различных реализаций языка.

В каждой стране, как правило, есть одна или несколько организаций, наделенных правом разработки стандартов. В Соединенных Штатах это Американский национальный институт стандартов (ANSI — American National Standards Institute). Стандарты языков программирования могут разрабатываться комитетом X3 Ассоциации производителей делового компьютерного оборудования (CBEMA — Computer Business Equipment Manufactures Association), а также Институтом инженеров по электротехнике и электронике (IEEE — Institute of Electrical and Electronic Engineers).

В Великобритании такими полномочиями наделен Британский институт стандартов (BSI — British Standards Institute). Международные стандарты

создаются Организацией международных стандартов (ISO — International Standards Organization), штаб-квартира которой находится в Женеве (Швейцария).

Хотя теоретически все выглядит хорошо, на самом деле создание стандартов является как технической, так и политической проблемой. Например, поставщики компиляторов имеют определенный финансовый интерес в создании определенных стандартов. Естественно, они хотят, чтобы стандарт был как можно ближе к их компилятору, чтобы не пришлось вносить изменения в свою реализацию. Такие изменения не только дорого стоят сами по себе, они приводят еще и к тому, что программы пользователей, работающих с измененным компилятором, перестают отвечать стандарту, что создает неудобства потребителям.

Создание стандартов — процесс, основанный на компромиссах, а получившийся в результате язык будет приемлемым для всех. Чтобы использовать стандарт эффективно, следует рассмотреть три аспекта стандартизации.

1. Своевременность, т. е. следует создавать стандарт языка не слишком рано, чтобы накопилось достаточно опыта в его применении, но и не слишком поздно, чтобы не поощрять создание несовместимых реализаций.

2. Соответствие. Программа считается соответствующей стандарту, если она использует только те возможности, которые определены данным стандартом. Компилятор является соответствующим стандарту, когда после компиляции соответствующей стандарту программы при ее выполнении получается правильный результат.

3. Устаревание. Процесс стандартизации предусматривает возможность обновления. Каждый стандарт должен пересматриваться раз в пять лет и либо обновляться, либо совсем отменяться.

Поскольку расширения языка допускаются и в согласующихся со стандартом компиляторах (до тех пор, пока они правильно компилируют согласующиеся со стандартом программы), многие из них имеют дополнения, которые с точки зрения поставщика программного обеспечения являются полезными и способствуют распространению программного продукта. Благодаря такому подходу появляются нововведения и развивается язык. Конечно, большинство программистов не связывают себя никакими стандартами, а разрабатывают свои собственные продукты, в которых языки модифицированы и расширены подходящим для них образом. Это создает плодородную почву для испытания новых языковых концепций, а некоторые наиболее удачные идеи попадают в коммерческие языки и компиляторы.

3. Понятие низкоуровневого программирования.

Низкоуровневое программирование – это программирование, основанное на прямом использовании возможностей и особенностей конкретной вычислительной системы. Для того чтобы писать программы на этом уровне, необходимо знать архитектуру аппаратной части системы:

структуру и функционирование системы в целом;

организацию оперативной памяти;

состав внешних устройств, их адреса и форматы регистров;

организацию и функционирование процессора, состав и форматы его регистров, способы адресации, систему команд;

систему прерываний и т. д.

При этом не следует забывать, что вычислительная система – это совокупность не только аппаратных, но и программных средств, и особенности имеющегося программного обеспечения (операционной системы) оказывают существенное влияние на разработку программ.

В истории развития программирования существовали три разновидности низкоуровневых языков, последовательно сменивших друг друга:

машинный код;

мнемокод;

ассемблер.

В машинных кодах программа представляется в виде последовательности чисел, являющихся кодами команд процессора, адресами оперативной памяти, номерами регистров процессора и внешних устройств и т. д. Фрагмент последовательности кодов команд микропроцессора Intel 8086 для добавления двухбайтового слова с адресом 36 к слову с адресом 32:

Адрес	Код
0000	A1
0001	20
0002	00
0003	03
0004	06
0005	24
0006	00
0007	A3
0008	20
0009	00

Очевидно, что в таком виде создание сколько-нибудь сложной программы является весьма трудной задачей.

Для решения этой задачи были разработаны так называемые мнемокоды, которые и явились основой для современных ассемблеров. Они вместо чисел позволяли использовать мнемонические (символьные) имена, отражающие смысл выполняемой команды. Приведенная выше последовательность команд в мнемонической записи:

```
Mov     ax,ds:[32]
Add     ax,ds:[36]
Mov     ds:[32],ax
```

В этом фрагменте уже виден смысл команд, из которых складывается сложение двух слов памяти.

Ассемблер отличается от своего предшественника – мнемокода обширным набором директив транслятора, существенно упрощающих процесс

кодирования программы, в первую очередь директивами оформления программы в виде логически законченных элементов и макросредствами.

ЛЕКЦИЯ 2
ОБЩАЯ ХАРАКТЕРИСТИКА ЯЗЫКОВ АССЕМБЛЕРА

План лекции

1. Общие сведения о языках ассемблера.
2. Синтаксис ассемблера.
3. Директивы ассемблера.
4. Обзор системы команд процессора.

1. Назначение языков ассемблера.

Ассемблер (от англ. assemble — собирать) — компилятор с языка ассемблера в команды машинного языка.

На сегодняшний день ассемблер как язык программирования, предназначенный для создания программ, используется крайне редко. Это связано с тем, что этот процесс является чрезвычайно трудоемким и получающиеся программы являются системнозависимыми, их нельзя свободно переносить между компьютерами, имеющими различные аппаратные архитектуры и операционные системы. Поэтому на сегодняшний день ассемблер изучают в учебных целях, так как только он дает полное представление о реальном устройстве и функционировании аппаратуры и операционной системы.

Сейчас разработка программ на ассемблере применяется в основном в программировании небольших микропроцессорных систем (микроконтроллеров), как правило, встраиваемых в какое-либо оборудование. Очень редко возникает потребность использования ассемблера в разрабатываемых программах в тех случаях, когда необходимо, например, получить наивысшую скорость выполнения определенного участка программы, выполнить операцию, которую невозможно реализовать средствами языков высокого уровня, либо уместить программу в память ограниченного объема (типичное требование для загрузчиков операционной системы).

Под каждую архитектуру процессора и под каждую ОС или семейство ОС существует свой ассемблер. Есть также так называемые кросс-ассемблеры, позволяющие на машинах с одной архитектурой (или в среде одной ОС) ассемблировать программы для другой целевой архитектуры или другой ОС и получать исполняемый код в формате, пригодном к исполнению на целевой архитектуре или в среде целевой ОС.

Язык ассемблера — тип языка программирования низкого уровня. Команды языка ассемблера один в один соответствуют командам процессора и представляют собой удобную символьную форму записи (мнемокод) команд и аргументов. Язык ассемблера обеспечивает связывание частей программы и данных через метки, выполняемое при ассемблировании (для каждой метки

высчитывается адрес, после чего каждое вхождение метки заменяется на этот адрес).

Каждая модель процессора имеет свой набор команд и соответствующий ему язык (или диалект) ассемблера.

Обычно программы или участки кода пишутся на языке ассемблера в случаях, когда разработчику критически важно оптимизировать такие параметры, как быстродействие (например, при создании драйверов) и размер кода (загрузочные секторы, программное обеспечение для микроконтроллеров и процессоров с ограниченными ресурсами, вирусы, навесные защиты).

Связывание ассемблерного кода с другими языками. Большинство современных компиляторов позволяют комбинировать в одной программе код, написанный на разных языках программирования. Это дает возможность быстро писать сложные программы, используя высокоуровневый язык, не теряя быстродействия в критических ко времени задачах, применяя для них части, написанные на языке ассемблера. Комбинирование достигается несколькими приемами:

1. Вставка фрагментов на языке ассемблера в текст программы (специальными директивами языка) или написание процедур на языке ассемблера. Способ хороший для несложных преобразований данных, но полноценного ассемблерного кода с данными и подпрограммами, включая подпрограммы с множеством входов и выходов, не поддерживаемых высокоуровневыми языками, с помощью него сделать нельзя.

2. Модульная компиляция. Большинство современных компиляторов работают в два этапа. На первом этапе каждый файл программы компилируется в объектный модуль. А на втором объектные модули линкуются (связываются) в готовую программу. Прелесть модульной компиляции состоит в том, что каждый объектный модуль будущей программы может быть полноценно написан на своем языке программирования и скомпилирован своим компилятором (ассемблером).

Достоинства языков ассемблера.

1. Максимально оптимальное использование средств процессора, использование меньшего количества команд и обращений в память и, как следствие, большая скорость и меньший размер программы.

2. Использование расширенных наборов инструкций процессора (MMX, SSE, SSE2, SSE3).

3. Доступ к портам ввода-вывода и особым регистрам процессора (в большинстве ОС эта возможность доступна только на уровне модулей ядра и драйверов)

4. Возможность использования самомодифицирующегося (в том числе перемещаемого) кода (под многими платформами она недоступна, так как запись в страницы кода запрещена, в том числе и аппаратно, однако в большинстве общедоступных систем из-за их врожденных недостатков имеется возможность исполнения кода, содержащегося в сегменте (секции) данных, куда запись разрешена).

5. Максимальная адаптация для нужной платформы.

Однако следует отметить, что последние технологии безопасности, внедряемые в операционные системы и компиляторы, не позволяют делать самомодифицирующего кода, так как исключают одновременную возможность исполнения программы и запись в одном и том же участке памяти (технология W^X).

Технология W^X используется в OpenBSD , в других BSD-системах, в Linux. В Microsoft Windows (начиная с Windows XP SP2) применяется схожая технология DEP.

Недостатки языков ассемблера.

1. Большие объемы кода, большое число дополнительных мелких задач, меньшее количество доступных для использования библиотек по сравнению с языками высокого уровня.
2. Трудоемкость чтения и поиска ошибок (хотя здесь многое зависит от комментариев и стиля программирования).
3. Часто компилятор языка высокого уровня, благодаря современным алгоритмам оптимизации, даёт более эффективную программу (по соотношению качество/время разработки).
4. Непереносимость на другие платформы (кроме совместимых).
5. Ассемблер более сложен для совместных проектов.

2. Синтаксис ассемблера.

Синтаксис общих элементов языка. В отличие от языков программирования высокого уровня, где основным элементом языка является оператор, синтаксически программа на ассемблере состоит из последовательности строк. Строка – основная единица ассемблерной программы (рис. 2.1).

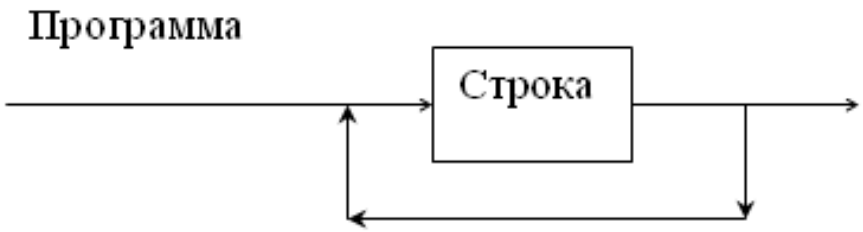


Рис. 2.1. Синтаксис программы

Если строка в программе одна, то она должна содержать директиву ассемблера end, завершающую процесс трансляции.

Синтаксис строки схематично может быть изображен в виде следующей диаграммы (рис.2.2).

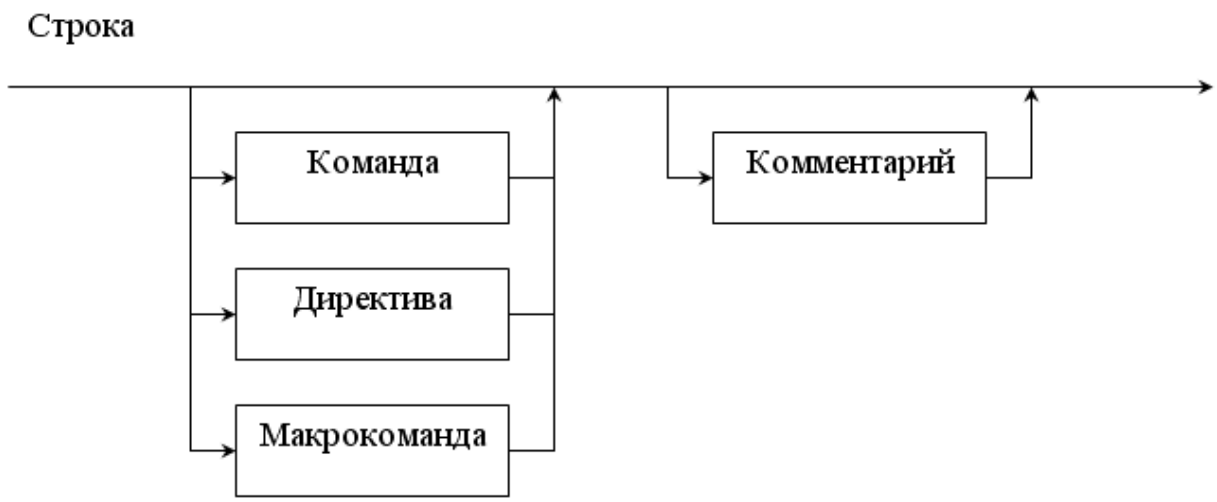
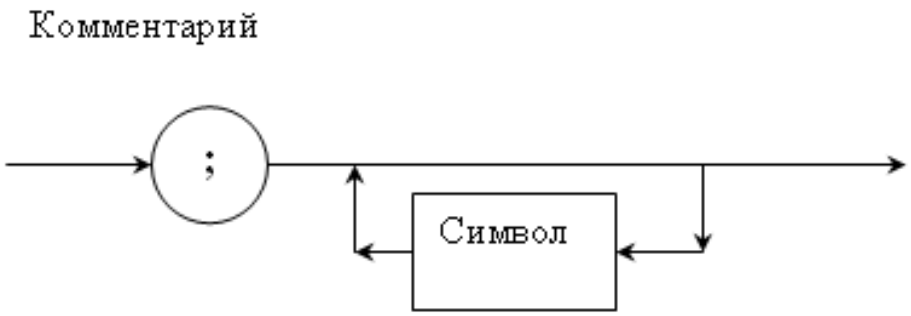


Рис. 2.2. Синтаксис строки

Комментарий описывается следующей диаграммой (рис.2.3).



Символ – любой отображаемый (печатный) символ.

Рис. 2.3. Синтаксис комментария

Команда – указание команды (инструкции) процессора (рис. 2.4).

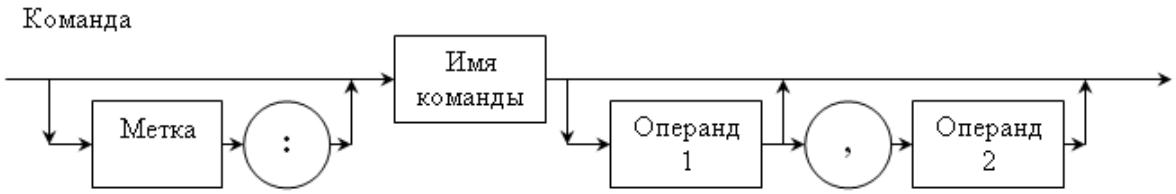


Рис. 2.4. Синтаксис команды

Директива – команда, выполняемая транслятором во время обработки программы, имеет следующий синтаксис (рис. 2.5).

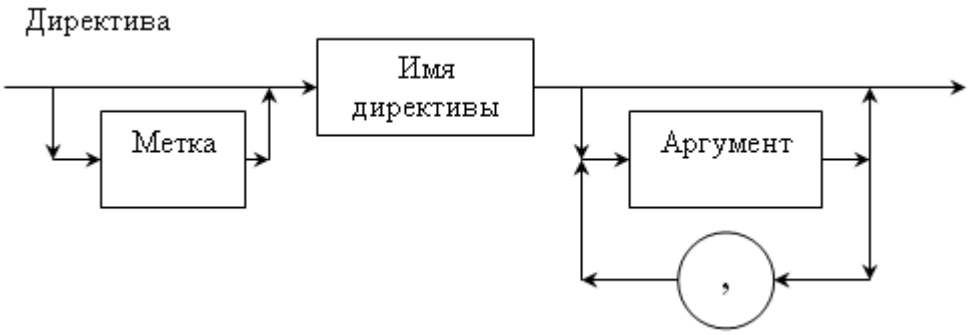


Рис. 2.5.Синтаксис директивы

Синтаксис макрокоманды выглядит следующим образом (рис. 2.6).

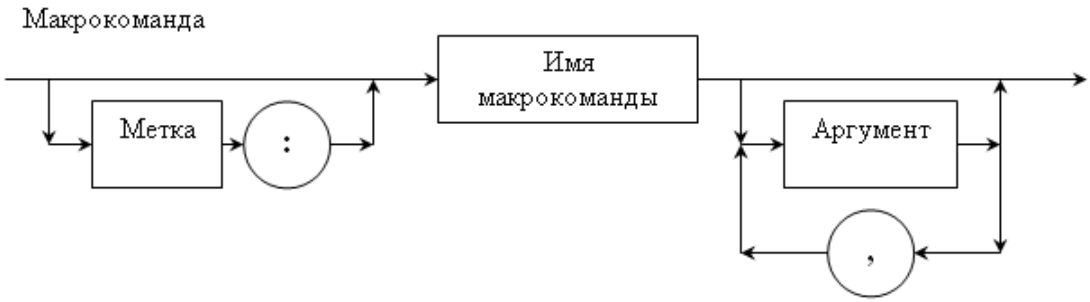


Рис. 2.6. Синтаксис макрокоманды

В приведенных синтаксических диаграммах в качестве необязательного элемента присутствует метка (рис. 2.7).

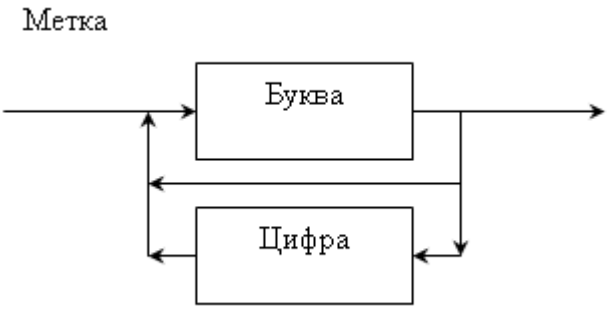


Рис. 2.7. Синтаксис метки

Понятие буквы в ассемблере включает в себя все латинские буквы, причем по умолчанию заглавные и прописные буквы не различаются, и символы @, $, &, _, ?. Цифры – арабские от 0 до 9.

Сегментация в ассемблере. Дальнейшее рассмотрение синтаксиса ассемблера лучше всего проводить, имея перед глазами пример программы. Традиционно первая программа в изучении языков программирования.

```
                Title       hello
                Assume      cs:c, ds:d, ss:s

s               segment     stack
                dw          128 dup (?)
s               ends

d               segment
msg             db          ‘hello’,  0dh, 0ah, ‘world!$’
d               ends

c               segment
start:          mov         ax,d
                mov         ds,ax

                mov         ah,9
                lea         dx,msg
                int         21H

                mov         ah,4ch
                int         21h
c               ends
                end         start
```

Из приведенного примера видно, что по существу программа состоит из описаний сегментов, начинающихся с директивы segment и оканчивающихся директивой ends.

Сегментом называется часть программы, содержащая совокупность логически однородной информации.

Типичный набор сегментов программы содержит сегмент стека, сегмент данных и сегмент кодов команд (в приведенном примере – это сегменты с именами s, d и c соответственно). Вне сегментов указываются директивы транслятора, определяющие общие особенности трансляции программы.

В примере это директивы:

title – дает модулю имя hello;

assume – информируют транслятор о том, что во время исполнения программы регистр cs будет содержать начальный адрес сегмента с именем c, ds – сегмента d, ss – сегмента s;

end – указывает, что запускать программу следует с метки start (определяет точку входа в программу), и завершает трансляцию.

Синтаксическая диаграмма сегмента приведена на рис. 2.8.

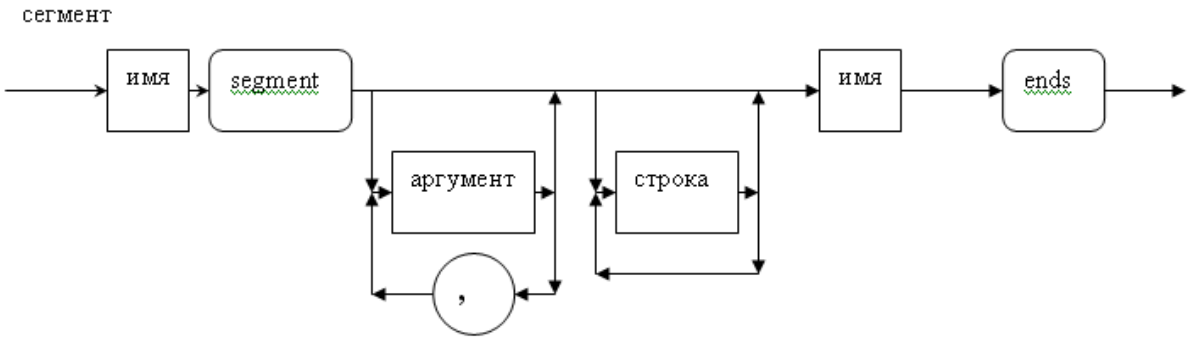


Рис. 2.8. Синтаксис сегмента

Описание сегмента начинается с директивы segment и оканчивается директивой ends, перед которыми указывается имя сегмента. После ключевого слова segment могут следовать аргументы, описывающие атрибуты сегмента (в приведенном примере сегмент s имеет атрибут stack, указывающий, что данный сегмент является сегментом стека).

Сегменты могут описываться в любой последовательности, и количество сегментов данных и кодов команд не ограничивается. Сегмент стека должен присутствовать обязательно, и он всегда единственный.

Максимальный размен сегмента ограничен и составляет 2^{16} байт.

В одном из сегментов кодов программы обычно присутствует метка, являющаяся точкой входа в программу. Эта метка указывается в последней директиве end, заставляющей транслятор завершить свою работу.

3. Директивы ассемблера.

Директивы ассемблера можно классифицировать следующим образом (рис. 2.9).

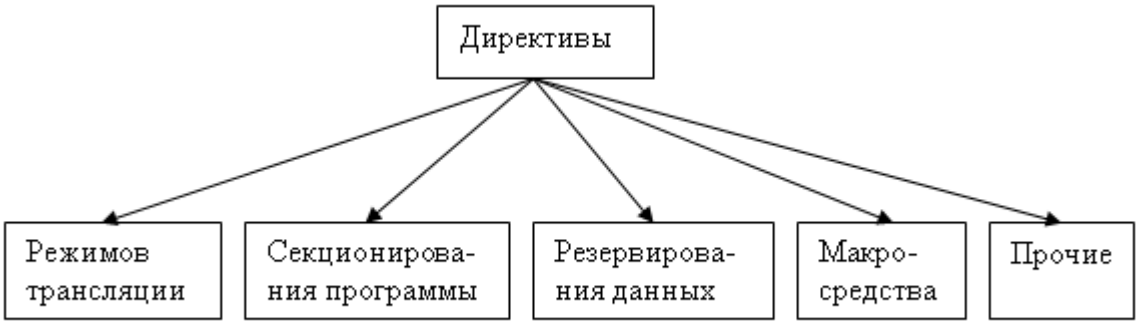


Рис. 2.9. Классификация директив ассемблера

Директивы режимов трансляции. Директивы определения общего режима трансляции определяют глобальные особенности трансляции, которые должны учитываться при обработке текста программы (рис. 2.10).

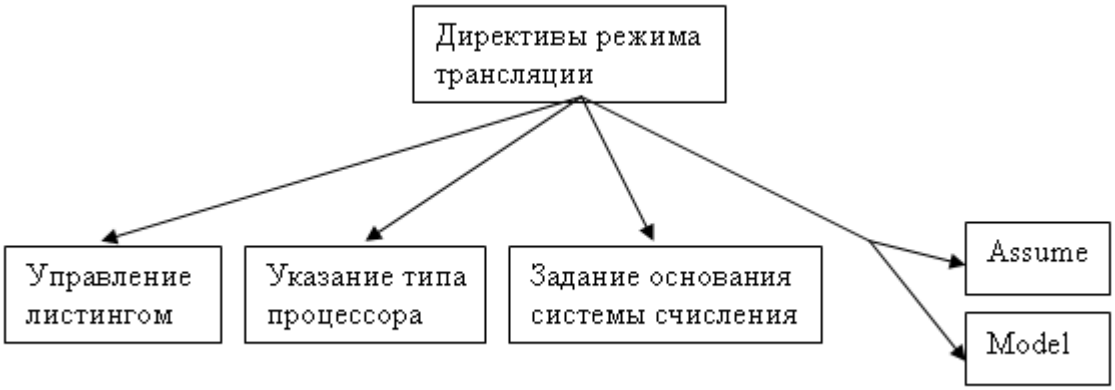


Рис. 2.10. Директивы определения общего режима трансляции

Директивы управления листингом определяют, что следует включать в содержимое файла листинга. Примерами директив этой группы являются:

.LIST – включить в листинг все строки исходного текста программы;

.NOLIST – исключить из листинга исходный текст;

.SUMS – включить в листинг таблицу символов;

.NOSUMS – исключить из листинга таблицу символов и т. д.

Директивы указания типа процессора определяют, какие регистры процессора, режимы адресации, команды разрешено использовать в программе. Примеры этих директив:

.8086 – разрешено использовать аппаратуру микропроцессора 8086;

.80286 – то же самое, но микропроцессор 80286;

.80286P – разрешает использование привилегированных команд микропроцессора 80286;

.80386;

80386P и т. д.

Наиболее широкие возможности дает директива .80586P.

Основание системы счисления определяет, как будут интерпретироваться числа по умолчанию. Для этих целей используются директивы .RADIX:

.RADIX 2 – как двоичные;

.RADIX 8 – как восьмеричные;

.RADIX 10 – как десятичные;

.RADIX 16 – как шестнадцатеричные.

Директивы assume фактически определяют точки программы, от которых транслятор отсчитывает смещения до меток, используемых в программе в качестве символических адресов.

Директива model чаще всего используется совместно с директивами упрощенной сегментации, рассматриваемыми далее. Она определяет, как будут расположены сегменты по отношению друг к другу, каким образом будут адресоваться данные, как будут вызываться подпрограммы

Директивы секционирования программы. Директивы секционирования программы предназначены для оформления логически законченных участков программы (рис. 2.11).

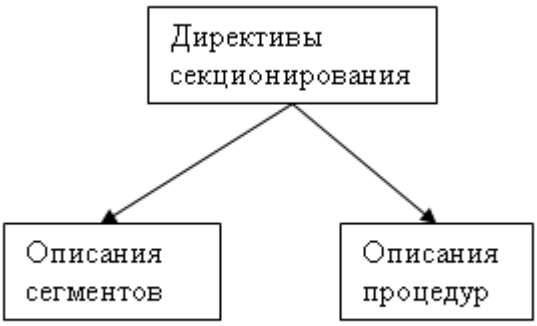


Рис. 2.11. Директивы секционирования программы

Сегменты программы могут описываться традиционными директивами segment и ends, как показано в приведенном примере, и директивами упрощенной сегментации. Примерами таких директив являются:

.code – открывает описание сегмента кодов команд;

.data – начинает описание сегмента данных;

.stack – описывает сегмент стека, в качестве аргумента этой директивы может указываться размер стека в байтах.

Приведенный выше пример программы с использованием директив упрощенной сегментации выглядит следующим образом:

```
        title   hello
        .model  small

        .stack  256

        .data
msg     db      'Hello',0dh,0ah,'world!$'

        .code
start:  mov     ax,@DATA
        mov     ds,ax

        mov     ah,9
        lea     dx,msg
        int     21h

        mov     ah,4ch
        int     21h
        end     start
```

Использование этих директив приводит к невозможности явного определения имен сегментов программистом, так как транслятор сам дает эти имена. Узнать эти имена можно из таблицы символов, включаемой в файл листинга. Использование упрощенной сегментации рекомендуется при разработке ассемблерных модулей, предназначенных для вызова из программ, написанных на языках высокого уровня.

Для описания процедур в ассемблере предусмотрены директивы proc и endp. Схематичное оформление процедуры с именем P:

```
P       proc
        ……
        ……; текст процедуры
        …….
P       endp
```

Детальное рассмотрение процедур дается далее при рассмотрении вопросов модульного программирования.

Директивы макросредств. Директивы макросредств позволяют при написании программы оперировать заранее заготовленными фрагментами текстов. Подробное рассмотрение макросредств приводится в дальнейших

лекциях, поэтому на данном этапе имеет смысл ограничиться их классификацией (рис. 2.12).

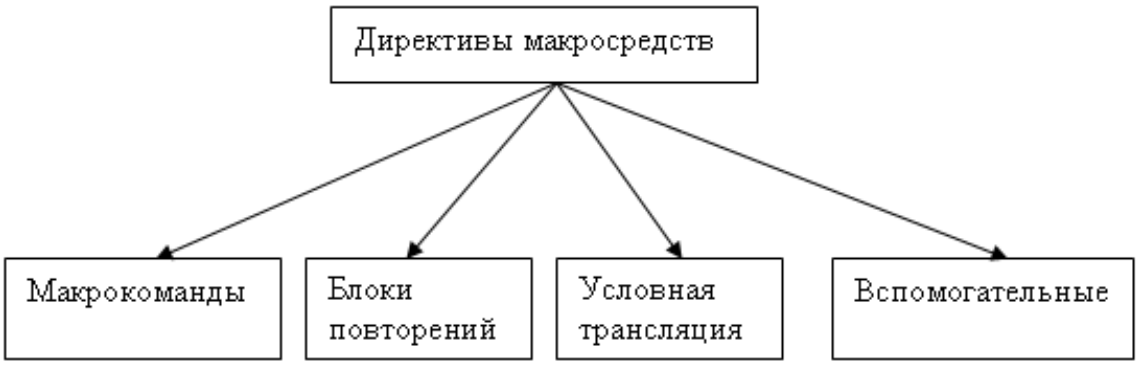


Рис. 2.12. Классификация директив макросредств

Директивы резервирования данных. Директивы резервирования данных размещают и в случае необходимости, инициализируют области памяти под данные программы. К ним относятся:

db (define byte) – резервирует область для байтовых данных;
dw (define word) – резервирует область для двухбайтовых данных;
dd (define double word) – резервирует область для четырехбайтовых данных;
df (define float) – резервирует область для шестибайтовых данных;
dq (define quakeук word) – резервирует область для восьмибайтовых данных;
dt (define ten bytes) – резервирует область для десятибайтовых данных.

Формат этих директив одинаковый (рис. 2.13).

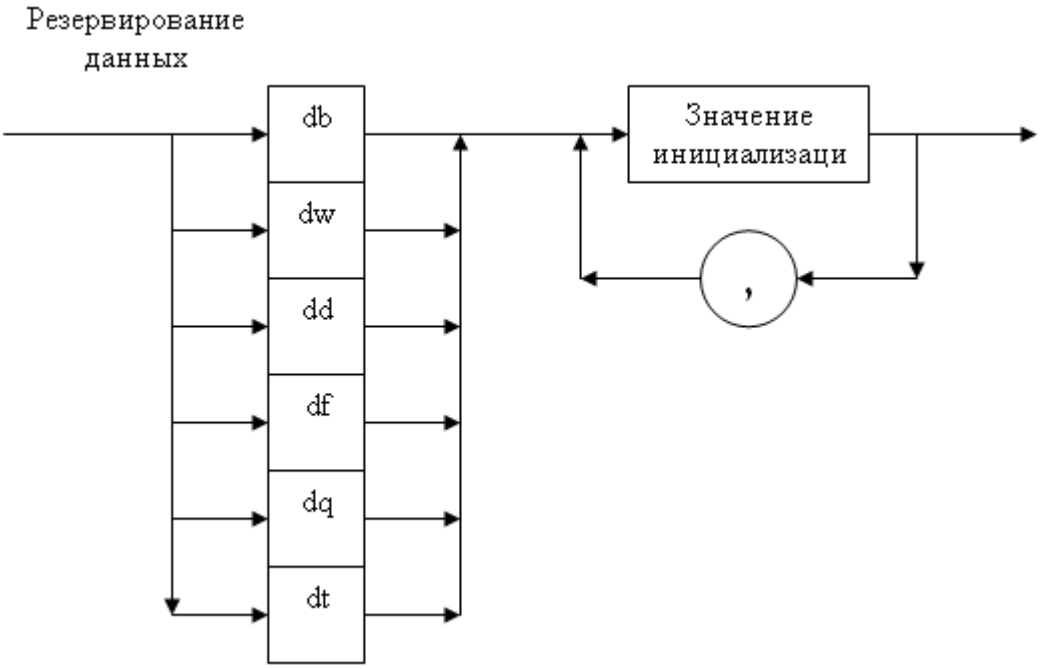


Рис. 2.13. Директивы резервирования данных

Синтаксис значения инициализации (рис. 2.14).

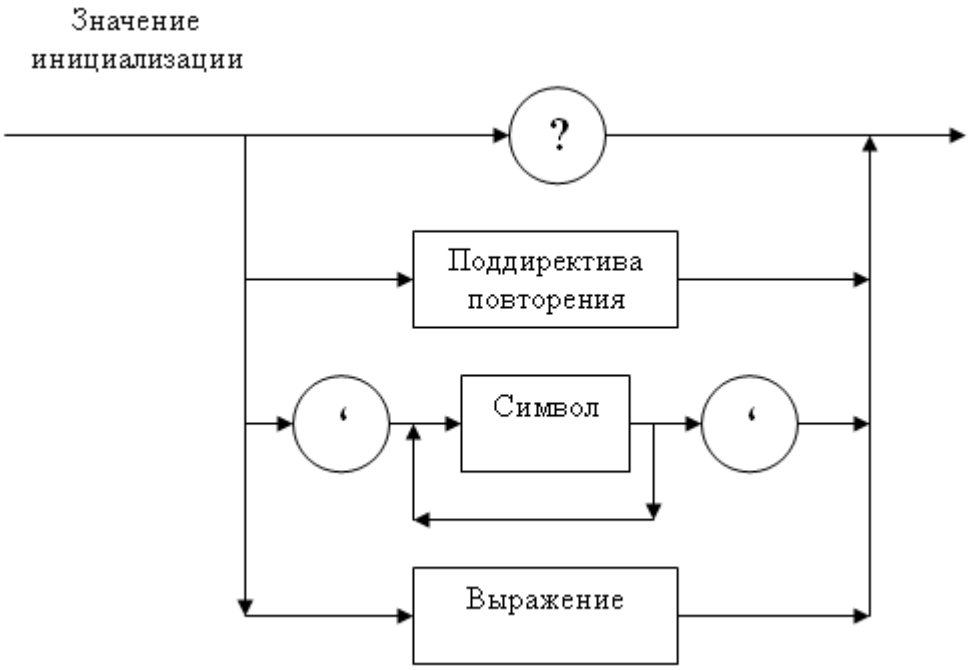


Рис. 2.14. Синтаксис значения инициализации

Здесь ? – означает произвольное значение (неинициализированная область); выражение – любое выражение, значение которого может подсчитать транслятор. Например, допустимыми выражениями являются:

15, 8Ch, 1010B – числа, записанные в десятичной, шестнадцатеричной и двоичной системах счисления;

(15+8*6)/(4+4) – эквивалент записи числа 7;

A+7*9+5/2-(A-B)/(4+2*5) – в выражениях могут участвовать символы, имеющие числовые значения (в частности метки, определенные в программе). Здесь A и B – такие символы.

Поддиректива повторения позволяет заполнить область повторяющимися значениями. Ее синтаксис (рис. 2.15).

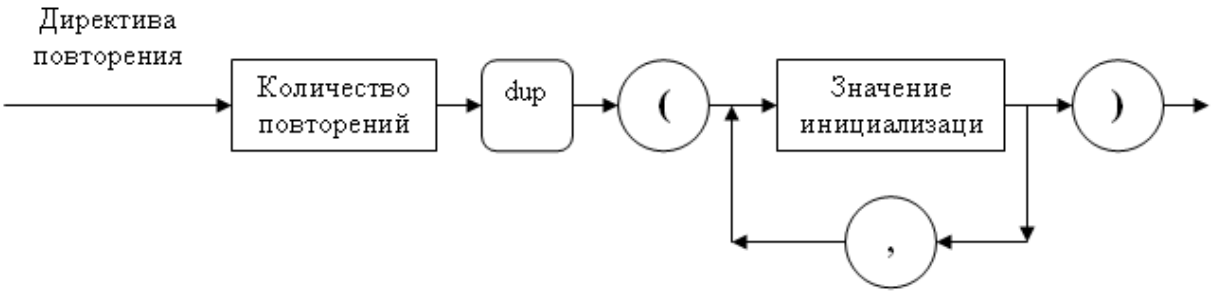


Рис. 2.15. Синтаксис поддирективы повторения

Здесь количество повторений – число, определяющее, сколько раз будет повторен список значений инициализации, указанный в круглых скобках.

Таким образом, директива, приведенная в примере программы

```
msg     db      'Hello',0dh,0ah,'world!$'
```

резервирует область памяти из 14 байтов, размещает в них коды символов и числа, а смещение от начала сегмента до первого байта этой области получает в качестве значения метка msg (рис. 2.16).

‘H’	‘e’	‘l’	‘l’	‘o’	0dh	0ah	‘w’	‘o’	‘r’	‘l’	‘d’	‘!’	‘$’
msg	msg+1	msg+2	msg+3	.....									

Рис. 2.16. Содержимое памяти

Директива dw отличается от db только тем, что размещает значения инициализации не в байтах, а в словах, dd – в двойных словах и т. д. Следует помнить, что при использовании в качестве значений инициализации строк символов транслятор будет разбивать их на последовательности байтов соответствующего размера и размещать правый символ в младшем байте, а левый –

в старшем. Например, директива dw ‘hello!’ разместит данные следующим образом:

‘e’	‘h’	‘l’	‘l’	‘!’	‘o’

Для директивы dd ‘hello!’ размещение данных будет выглядеть следующим образом:

‘l’	‘l’	‘e’	‘h’	‘!’	‘o’	0	0

4. Обзор системы команд процессора.

В ассемблерных программах наряду с директивами используются мнемонические записи команд (инструкций) процессора. Совокупность всех команд, которые может выполнять процессор, образует его систему команд. Естественно, процессоры разных архитектур имеют различные системы команд и, соответственно, различные ассемблеры, но ряд общих черт все-таки имеется. В большинстве случаев в системах команд процессоров выделяются группы:

команд передачи данных. Команды этой группы используются для перемещения данных между различными узлами компьютера;

арифметические команды – предназначены для выполнения сложений, вычитаний, умножений и делений чисел;

логические команды – выполняют логические сложение и умножение, инверсию и т. д.;

команды сдвигов – выполняют различные поразрядные сдвиги влево и вправо;

команды управления выполнением. В эту группу включаются команды, нарушающие естественный порядок выполнения команд программы, такие как программные прерывания и возврат из прерываний, вызов и возврат из подпрограмм, различные условные и безусловные переходы.

Как правило, часть команд различного назначения объединяется в группу прочих команд. В архитектуре x86 присутствуют еще:

строковые (цепочечные) команды;

команды сопроцессора x87;

команды MMX™ – расширения (появились в процессорах Intel Pentium©).

Дальнейшее рассмотрение системы команд является одним из вопросов следующей лекции.

ЛЕКЦИЯ 3
СРЕДСТВА ВЗАИМОДЕЙСТВИЯ АССЕМБЛЕРНЫХ ПРОГРАММ С ОС

План лекции

1. Взаимодействие ассемблерных программ с ОС.
2. Ассемблеры для x86.

1. Взаимодействие ассемблерных программ с ОС.

Неотъемлемой частью программирования на ассемблере является использование услуг, предоставляемых операционными системами. Способы взаимодействия прикладных программ с ОС отличаются в различных операционных системах.

В MS DOS для обслуживания прикладных программ предусмотрен набор процедур, называемых функциями MS DOS. Каждая функция имеет свой уникальный номер, который перед передачей управления в DOS прикладная программа обязана разместить в регистре Ah. В зависимости от вызываемой функции в остальные регистры помещается различная другая информация, описывающая передаваемый запрос. Передача управления из прикладной программы в DOS производится с помощью программного прерывания с вектором 21h: int 21h. Далее операционная система по содержимому регистра Ah определяет, какую операцию запросила программа, пытается ее выполнить, а если это невозможно, возвращает управление в прикладную программу с установленным флагом переноса (CF). В этом случае в регистр Ax она помещает код ошибки. Если запрос выполнен успешно, управление в программу возвращается со сброшенным CF. В этом случае, если запрос имеет результаты, они размещаются в регистрах процессора.

Основными группами функций DOS являются:

системная информация. В эту группу входят такие функции, как получение системных даты и времени, версии системы, текущего каталога и т. д.;

символьный ввод-вывод. В эту группу входят функции ввода и вывода символов и строк;

Handle – ориентированный ввод-вывод. Данная группа включает в себя функции для работы с файлами;

традиционные FCB-операции. Эти функции являются устаревшими аналогами Handle – ориентированного ввода-вывода

функции файловой системы. В эту группу собраны функции создания и удаления каталогов, переименования файлов;

управление процессами. Функции этой группы позволяют порождать и уничтожать процессы;

управление памятью. Позволяет динамически запрашивать и освобождать участки оперативной памяти;

смешанные операции. Эту группу образуют различные функции, не вошедшие в другие группы;

драйверы устройств. Группа функций для работы с драйверами внешних устройств.

2. Ассемблеры для x86.

На сегодняшний день в сере программирования DOS/WINDOWS наибольшее распространение нашли две среды программирования на ассемблере.

Borland Turbo Assembler (TASM).

Microsoft Macro Assembler (MASM).

Для Unix – подобных систем чаще всего используется:

Unix Assembler (AS).

GNU Assembler (GAS).

Характеристика пакета MASM. Основная концепция пакета MASM 6.1x – совместить удобства программирования, свойственные языкам высокого уровня, с традиционными достоинствами машинно-ориентированных языков.

В состав пакета ассемблера фирмы Microsoft входят следующие программы:

masm.exe – ассемблер;

ml.exe (Masm and Link)– ассемблер и компоновщик;

link.exe – компоновщик;

pwb.exe (Programmer's WorkBench) – интегрированная среда разработки ассемблерных программ, позволяющая редактировать, формировать, отлаживать и выполнять ассемблерную программу. Среда обладает макроязыком, с помощью которого существенно повышается гибкость управления ее работой;

cv.exe (CodeView) – отладчик ассемблерных программ, предназначенный для обнаружения ошибок в ассемблерных программах. Существуют два варианта отладчика – один для MS DOS, другой для Microsoft Windows;

вспомогательные утилиты lib.exe, nmake.exe и др.

Пакет MASM позволяет производить разработку программ двумя способами:

традиционным для ассемблера способом – запуском отдельных программ трансляции, компоновки и отладки;

с использованием интегрированной среды, запускаемой программой pwb.exe.

При традиционном способе разработки программы используются следующие средства пакета MASM: masm.exe, ml.exe, link.exe и cv.exe.

Программа **masm.exe**

Командная строка masm.exe имеет вид:

Masm[ключи]исх_файл[.[объектный_файл][.[файл_листинга][.[файл_перекрестных_ссылок]]]]

Ключи (некоторые из них):

/A – сегменты должны быть упорядочены в алфавитном порядке.

/H – вывести справку по ключам командной строки.

/HELP – вызвать QuicHelp (qh.exe)для справки по MASM.

/L – создать обычный листинг.

/LA – в листинг помещается вся возможная информация.

/ZI – включение в объектный файл информации о идентификаторах для отладчика CodeView.

Программа **ml.exe**

Транслирует и связывает один или более исходных ассемблерных файлов. Командная строка ml.exe имеет следующий вид:

ml [ключи]исх_файл_1[[ключи]исх_файл_2]…[/link ключи_link].

Ключи командной строки чувствительны к регистру (ниже приведены некоторые из них):

/c – только трансляция исходного файла.

/Feимя_файла – задать имя исполняемого файла.

/Flимя_файла – генерировать листинг.

/Fmимя_файла – создать файл карты компоновщика.

/Foимя_файла – имя объектного файла.

/Sa – включить в листинг всю доступную информацию.

/Sg – включить в листинг текст сгенерированного кода.

/Si – включение в объектный файл информации для отладчика CodeView.

/? – справка о синтаксисе командной строки ml.

Программа **link.exe**

Компоновщик link компонует (объединяет) объектные файлы и библиотеки в исполняемый файл или динамически компонуемую библиотеку (DLL). Командная строка link.exe имеет вид:

Link[ключи]объект_файлы[.[исполн_файл][.[файл_карты][файлы_библиотек][.[def_файл]]]][;].

Некоторые ключи приведены ниже:

/T – создать исполняемый файл для MS DOS формата .com.

/? – вывод информации о синтаксисе link.exe.

Программа **cv.exe**

Отладчик Microsoft CodeView Debugger – предназначен для выполнения исполняемого файла программы при одновременном отображении ее исходного текста, состояния переменных, регистров процессора, занимаемой памяти и другой сопутствующей информации.

Существуют две версии отладчика CodeView – для MS DOS (cv.exe) и Windows (cvw.exe).

Отладчик cv.exe запускается следующей командной строкой:

Cv[ключи]исполн_файл[параментры].

Некоторые значения ключей приведены ниже:

/2 – разрешает использование двух мониторов.

/25 – запуск отладчика с 25 строками.

/43 – запуск отладчика в режиме с 43 строками.

/50 – запуск отладчика в режиме с 50 строками.

/M – запретить CodeView использовать мышь.

Программа **lib.exe**

Программа lib.exe является библиотечным менеджером, который создает и обслуживает стандартные библиотеки объектных модулей. С его помощью программист может выполнить все действия с файлом библиотеки: создание, добавление, удаление и замену модулей.

Программа lib.exe запускается следующей командной строкой:

Lib библиотека[ключи][команды][.[файл_листинга][.[вых_библиотека]]][;]

Значения команд приведены ниже:

+name – добавить объектный код или библиотечный файл в конец библиотечного файла.

-name – удалить модуль.

-+name – заменить модуль, удаляя его и добавляя в конец с тем же самым именем.

*name – копировать модуль в новый объектный файл.

-*name – переместить модуль (удаляя его) из библиотеки в новый объектный файл.

Характеристика пакета TASM. В состав пакета TASM входят следующие программы и утилиты:

1. 16- и 32-разрядные трансляторы tasm.exe и tasm32.exe.
2. 16- и 32-разрядные компоновщики (редакторы связей) tlink.exe и tlink32.exe.
3. Turbo Debuger (TD) – отладчик, работающий на уровне исходного текста. Имеет 16- и 32-разрядные версии td.exe td32.exe. Существует отладчик tdw.exe, позволяющий производить отладку Windows-приложений.

Все исполняемые файлы утилит находятся в каталоге ..\bin пакета TASM.

Формат командной строки для запуска tasm.exe следующий:

TASM [ключи]имя исходного файла[, имя объектного файла]
[, имя_файла_листинга][, имя_файла_перекрестных_ссылок]

Формат командной строки для запуска tlink.exe следующий:

TLINK [ключи]список_объектных_файлов[,имя_загрузочного_модуля]
[,имя_файла_карты][,имя_файла_библиотки]
[,имя_файла_определений][,имя_ресурсного_файла]

Ниже приведены некоторые ключи транслятора TASM и редактора связей TLINK:

/h,/? – вывод на экран справочной информации.

/mn – установка количества (n) проходов транслятора TASM. По умолчанию – транслятор выполняет один проход. Максимально, при необходимости, можно задать выполнение до 5 проходов.

/zi – включить в объектный файл информацию для отладки.

/zd – поместить в объектный файл информацию о номерах строк, что необходимо для работы отладчика на уровне исходного текста программы.

/zn – запретить помещение в объектный файл отладочной информации.

/3 – поддержка 32-битного кода.

/ax – тип приложения:

/aa – оконное Windows-приложений.

/ap – консольное Windows-приложений.

/m – создать более полный файл карты (.map), чем это делается по умолчанию.

/s – то же, что /m, но дополнительно в файл карты включается информация о сегментах (адрес, длина в байтах, класс, имя сегмента и т. д.).

/t – создать файл типа .com (по умолчанию .exe).

/v – включить отладочную информацию в выполняемый файл.

ЛЕКЦИЯ 4
ОСНОВЫ АРХИТЕКТУРЫ ЭВМ ЛИНИИ X86

План лекции

1. Набор регистров процессора, их форматы, назначение, особенности использования.
2. Программируемые регистры.
3. Регистр флажков.

1. Набор регистров процессора, их форматы, назначение, особенности использования.

Для программирующих на языке ассемблер главным объектом при изучении любого процессора является набор доступных внутренних регистров, образующих программную или регистровую модель процессора. Она показывает те ресурсы процессора, которыми может пользоваться программист, привлекая его систему команд.

В разработке регистровых моделей современных процессоров имеются два крайних подхода, между которыми находятся и промежуточные варианты. В первом подходе почти все регистры процессора выполняют абсолютно одинаковые функции, т. е. являются полностью взаимозаменяемыми. Обычно такие регистры называются универсальными. Во втором подходе, характерном для процессоров фирмы Intel, многие регистры специализированы, т. е. в некоторых командах выполняют специфические функции. Считается, что закрепление регистров за часто используемыми функциями обеспечивает более компактное кодирование команд (или, как говорят, улучшает плотность кода) и приводит к повышению производительности процессора. Конкретные регистры применяются в командах умножения и деления двойной точности, преобразования, управления циклами, динамического сдвига, ввода-вывода, цепочечных командах и в стековых операциях.

Расширенный набор регистров процессора x86 объединяет лучшие стороны обоих подходов. В нем сохранены специализированные регистры для некоторых команд (в целях совместимости с предыдущими процессорами), но в большинстве команд при необходимости можно заменить предпочтительный регистр. Для использования другого регистра перед командой можно указать префиксы замены сегмента, размеров операнда и адреса; а также воспользоваться альтернативной, более длинной формой команды, в которой действует другой режим адресации. Различия в объектном коде, полученном при задании регистра, который участвует в команде вместо принимаемого по умолчанию, почти незаметны для программиста. Если же в конкретных командах привлекаются регистры по умолчанию, длина объектного кода обычно сокращается.

В программной модели процессора x86 имеется 31 регистр. Они подразделяются на 16 регистров прикладного программиста (пользовательские регистры) и 15 регистров системного программиста (системные регистры).

Напомним, что поле ss указывает масштабный коэффициент индекса, поле index определяет любой регистр общего назначения, кроме ESP, который служит индексным регистром, и поле base идентифицирует любой регистр общего назначения, участвующий в адресации как базовый регистр.

2. Программируемые регистры.

Большинство регистров процессора используется прикладными программистами, и только часть – системными программистами. Программируемые регистры включают в себя регистры общего назначения, сегментации, флажков и указателя команды.

Некоторые регистры входят в базовый регистровый набор. Это группа регистров, которые есть в каждом процессоре семейства x86. В нее входят младшие 16 бит прикладных регистров и слово состояния.

Регистров общего назначения всего восемь. Они содержат по 32 бита. Имена полных 32-битных регистров начинаются с буквы E (Extended — расширенный): EAX, EBX, ECX, EDX, EBP, ESI, EDI и ESP. Младшие 16 бит каждого из этих регистров можно указывать так же, как и в прежних CPU: AX, BX, CX, DX, BP, SI, DI и SP. Наконец, первые четыре 16-битных регистра можно адресовать парами байтных регистров: AH и AL для AX, DH и DL для DX, BH и BL для BX, CH и CL для CX. Формат регистров, на примере EAX, приведен на рис (рис. 4.1).

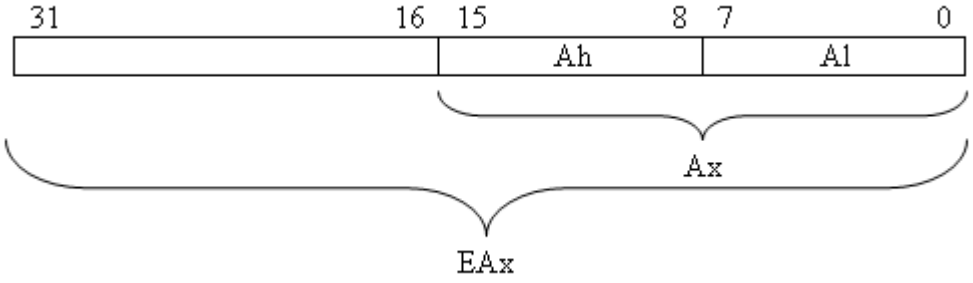


Рис. 4.1. Формат регистров EAX, EBX, ECX, EDX

Регистры EBP, ESI, EDI и ESP используются как указательные. Их формат приведен на рис. 4.2.

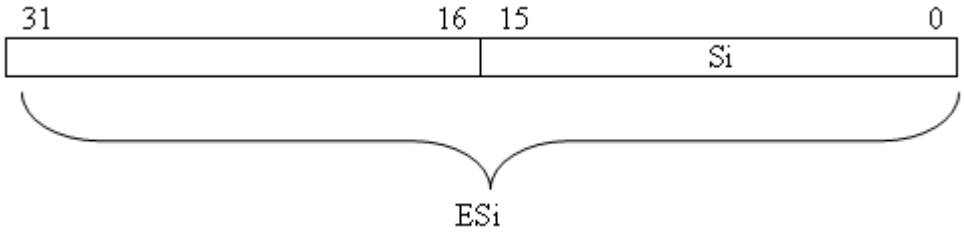


Рис. 4.2. Формат регистров EBP, ESI, EDI и ESP

Все регистры адресуются как двойные слова, т. е. как 32-битные регистры. В двухсловном регистре одна команда (например, MOV EAX,1) оперирует всем двойным словом. Младшая половина каждого регистра адресуется как слово (16-битный регистр). В двухсловный регистр можно загрузить слово в младшую половину, не влияя на старшие 16 бит. Когда нужно обращаться к регистру в 32- или 16-битной формах (как удобнее программисту), мы помещаем в скобках букву E. Например, (E)AX означает «EAX или AX – что нужно». Первые четыре регистра допускают в младшей половине адресацию байт.

Каждый из регистров общего назначения имеет специализированное применение. В командах типа ADD программист может указывать в качестве операндов любые два регистра. В операциях PUSH предполагается, что стек адресует регистр SP.

Покажем назначение каждого регистра:

(E)AX — аккумулятор, применяется в десятичной арифметике.

(E)BX — регистр базы, применяется как база при вычислении адреса.

(E)CX — счетчик, применяется как счетчик в циклических операциях.

(E)DX — регистр данных, хранит данные для нескольких операций.

(E)SP — указатель стека, содержит смещение вершины стека.

(E)BP — указатель базы, может содержит базу области данных.

(E)SI и (E)DI — индексы источника и получателя, применяются для адресации смещения.

Регистры сегментации применяются для определения начальных смещений в памяти области кода и данных. Каждый из шести этих регистров имеет свое назначение. CS адресует программный код, DS — данные программы, SS — ее стек. Дополнительные регистры ES, TS, FS и GS предназначены для структур данных. Из них только ES используется в конкретных командах. Все регистры 16-разрядные.

Прикладные программы обычно не обращаются к регистрам сегментации, ими полностью управляет операционная система. Содержимое регистров сегментации вместе с другими регистрами определяет, где находится следующая команда (CS плюс указатель команды), где вершина стека (SS плюс ESP) и т. д. Способ объединения содержимого регистров зависит от режима, в котором работает программа.

Еще два регистра — это регистр флажков EFlags, который управляет некоторыми операциями и показывает текущее состояние процессора x86, и указатель команды, который используется вместе с CS для адресации следующей команды. Программисту доступны только некоторые биты в регистре флажков, а указатель команды модифицируется только при переходах и вызовах.

3. Регистр флажков.

Регистр флажков EFlags процессора x86 содержит 32 бита. ПрикладныеРегистр флажков EFlags процессора x86 содержит 32 бита. Прикладные про-

граммы работают только с младшими 16 битами EFlags. Биты в EFlags отражают состояние x86 и управляют выполнением некоторых операций.

Различают три типа флажков: системные (они отражают текущее состояние компьютера в целом и чаще используются операционной системой, а не прикладными программами), состояния (показывают состояние конкретной программы) и управления (прямо влияют на некоторые команды).

Системные флажки. Флажок VM — виртуальный режим.

0 = защищенный режим; 1 = режим виртуального x86. Этот флажок показывает, работает ли ваша программа в режиме виртуального x86. Обычно вы не можете проверить этот бит (и следующий) в реальном режиме.

Флажок R — возобновление.

0 = нет ошибки; 1 = ошибка отладки.

Этот флажок временно включает средства отладки, когда программа возобновляет работу после особого случая отладки.

Флажок NT — вложенная задача.

0 = текущая задача не вложена; 1 = текущая задача вложена.

Этот флажок показывает, выполняется ли текущая задача «под» некоторой другой задачей. Он влияет на выполнение команды IRET.

Флажок IOPL — уровень привилегии ввода/вывода (биты 13 и 12).

0 = текущая задача имеет высший приоритет; 1 = следующий ниже; 2 = следующий ниже; 3 = низший приоритет.

Два бита IOPL используются процессором и операционной системой для определения прав доступа прикладной программы к средствам ввода/вывода. Допустимые уровни варьируются от 0 (наиболее привилегированные) до 3 (наименее привилегированные).

Флажок I — прерывание.

0 = внешние прерывания запрещены; 1 = разрешены.

Этот флажок определяет, будет ли CPU реагировать на внешние прерывания или игнорирует их. Он не влияет на особые случаи прерывания (порождаемые программой) и немаскируемые внешние прерывания. Программы в мультизадачных системах должны избегать воздействия на этот флажок.

Флажок T — трассировка.

0 = нет трассировки; 1 = прерывание после каждой команды.

Этот флажок вызывает генерирование особого случая прерывания по каждой команде. Применяется для покомандной работы при отладке

Флажки состояния. Флажки состояния (прямо используемые прикладными программами) устанавливаются многими командами x86, особенно арифметическими.

Программист может проверить флажок для определения последующих действий программы. Хотя каждый флажок имеет общую функцию, точный смысл его зависит от последней выполненной команды.

Флажок O — переполнение.

0 = нет переполнения; 1 = возникло переполнение.

Этот флажок устанавливается в 1, если результат арифметической операции превышает доступные пределы; если этого нет, флажок переполнения сбрасывается в 0.

Флажок S — знак.

0 = старший бит содержит 0; 1 = старший бит содержит 1.

Значение этого флажка совпадает со старшим битом результата. Для знаковых чисел этот бит показывает знак результата: 1 = отрицательный, 0 = положительный.

Флажок Z — нуль.

0 = последний ненулевой результат; 1 = последний результат был нулевым.

Этот флажок устанавливается в 1 (истина), если результат операции нуль, и сбрасывается в 0 (ложь), если результат ненулевой.

Флажок A — коррекция или вспомогательный перенос.

0 = нет внутреннего переноса; 1 = внутренний перенос.

Этот флажок показывает состояние «внутреннего» переноса или заема (при сложении и вычитании) из бита 3 в бит 4 — межтетрадный перенос или заем.

Флажок P — паритет.

0 – младший байт имеет четный паритет; 1 = младший байт имеет нечетный паритет.

Состояние этого флажка зависит от младшего байта результата: если он содержит четное число единиц, то P = 1, а в случае нечетного числа единиц P = 0.

Флажок C — перенос.

0 = нет переноса из старшего бита; 1 = есть перенос.

Показывает, вызвали ли сложение или вычитание перенос или заем из старшего бита результата.

Флажки управления. Флажки управления действуют только на циклические команды.

Флажок D — направление.

0 – автоинкремент; 1 = автодекремент в циклических командах.

Этот флажок управляет «направлением» операций. Когда D = 0, циклы обрабатываются от младших адресов к старшим, а когда D = 1, обработка производится от старших адресов к младшим.

ЛЕКЦИЯ 5
АДРЕСАЦИЯ ПАМЯТИ

План лекции

1. Режимы адресации.
2. Сегментация памяти в процессоре x86.
3. Система команд.

1. Режимы адресации.

Некоторые команды работают с данными, содержащимися в самой команде или в регистрах CPU, т. е. они не обращаются к памяти. Это примеры непосредственного и регистрового режима адресации. В команде MOV AX,7FH используются оба эти режима.

Когда программа обращается к памяти, она должна сообщить компьютеру, какую ячейку памяти использовать. В простейшей форме можно прямо указывать имя ячейки, например ADD AX,ANADDRESS, где ANADDRESS – ранее определенная ячейка.

Для эффективного программирования требуется использовать несколько способов именования ячеек памяти. В ассемблерном операторе для определения адреса можно объединять 4 возможных элемента. Вычисленный, или эффективный адрес, объединяется с соответствующим сегментным регистром, давая окончательный адрес. Вот эти элементы:

1. База. Это содержимое одного из регистров общего назначения. База считается начальной точкой, а другие элементы прибавляются к ней с образованием эффективного адреса. В регистр можно поместить начальную ячейку массива данных.

2. Смещение. Это адрес ячейки в сегменте памяти. Длина смещения 8, 16 или 32 бита.

3. Индекс. Как и в базовой адресации, для образования эффективного адреса используется содержимое какого-либо регистра. В случае 16-битных операндов для индексирования предназначены SI и DI, а в случае 32-битных операндов – любой регистр, кроме ESP.

4. Масштаб. Если индекс является 32-битной величиной, его можно умножить на 2, 4 или 8. Это удобно при обращении к массивам с элементами фиксированного размера. Эффективный адрес вычисляется следующим образом:

EA = База + (Индекс * Масштаб) + Смещение

При отсутствии некоторых элементов формула упрощается. Если, например, нет индекса, то EA = База + Смещение. Ниже приведены примеры использования каждого режима.

1. Прямая адресация включает регистровые и непосредственные операнды; обращение к памяти не требуется.

2. Используется только смещение. Смещение обычно указывается меткой; вычисляется расстояние от текущей команды до метки и используется как смещение при обращении к памяти.

3. В косвенной адресации адресом служит содержимое регистра. Имя регистра заключается в квадратные скобки. Например, команда MOV AX, BX передает в AX содержимое BX, а команда MOV AX,[BX] передает в AX содержимое ячейки памяти, адресуемой содержимым регистра BX.

4. Базовая адресация требует прибавить константу к значению в регистре и использовать сумму как эффективный адрес. Выражение «регистр + смещение» заключается в квадратные скобки. Например, в команде MOV AX,[BX+4] сумма содержимого BX и 4 служит эффективным адресом. Значение, содержащееся по данному адресу, передается в AX.

В остальных режимах используется индексирование. Индекс заключается в квадратные скобки и помещается после базы, с которой он суммируется. Масштабировать (умножать на 2, 4 или 8) можно только индексы.

5. В индексной адресации эффективный адрес равен сумме прямого адреса и индекса. В команде MOV ECX,TABLE[SI] эффективный адрес равен сумме SI и значения TABLE. Число, хранящееся по эффективному адресу, передается в ECX.

6. Индекс объединяет базу в регистре и индекс в регистре. В команде MOV ECX,[EDX][EAX] эффективный адрес равен сумме EDX и EAX. В случае 32-битных операндов индекс можно масштабировать.

7. Масштабированный индекс можно умножать на 2, 4 или 8. Например, в команде ADD ECX, TABLE [ESI*8] значение из ESI умножается на 8, суммируется с адресом TABLE и результат служит эффективным адресом. Такой прием особенно удобен для элементов данных длиной в 8 байт.

Кроме приведенных режимов можно использовать любую комбинацию базы, индекса (с масштабированием или нет) и смещения. Вычисление адреса ведется параллельно с другими действиями, поэтому даже в случае самого сложного режима адресации дополнительное время не требуется. Но есть одно исключение: при наличии базы, индекса и смещения время увеличивается на один такт.

2. Сегментация памяти в процессоре x86.

В процессоре x86 сегментация памяти реализована довольно простыми средствами. Под сегментом понимается блок смежных ячеек памяти (в адресном пространстве 1 Мбайт) с максимальным размером 64 Кбайт и начальным или базовым адресом, находящимся на 16-байтной границе (такая граница называется параграфом). Для обращения к памяти необходимо определить базу сегмента и 16-битное расстояние от базы, называемое смещением (offset) или относительным адресом. Базовые адреса четырех одновременно доступных программе сегментов находятся в сегментных регистрах кода CS, данных DS, стека SS и дополнительных данных ES. Каждый из них имеет длину 16 бит, но можно считать сегментный регистр 20-битным, так как че-

тыре младших бита базового адреса содержат нули. Таким образом, две 16-битных величины (указателя), соответствующие базовому адресу сегмента (с подразумеваемыми младшими нулями) и смещению, позволяют обращаться ко всему адресному пространству 1 Мбайт, как показано на рис. 5.1.

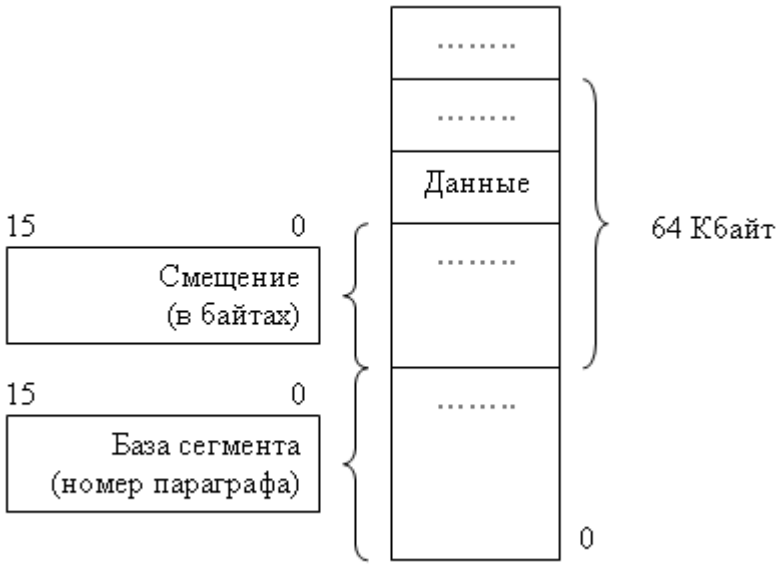


Рис. 5.1. Сегментация памяти в процессоре x86

Отметим, что сегментные регистры содержат физические адреса памяти, т. е. значение в каждом сегментном регистре прямо указывает на границу параграфа в адресном пространстве 1 Мбайт. В защищенном режиме последующих процессоров значение в сегментном регистре не имеет прямой взаимосвязи с реальным адресом памяти.

Чтобы упростить обращения к памяти, за каждой командой закреплен сегментный регистр по умолчанию, который привлекается без специальных указаний. Так, команды всегда выбираются из сегмента кода (указателями служат регистры CS и IP), все стековые операции выполняются в сегменте стека (SS:SP), данные (переменные) находятся в сегменте данных (база находится в регистре DS, а смещение, называемое эффективным адресом EA, определяется режимом адресации). Для обращения к другим сегментам перед командой помещается префикс замены сегмента. Например, команда MOV AX,[100H] загружает в регистр AX слово из сегмента, базовый адрес которого находится в регистре DS, а смещение равно 100H. Команда с префиксом замены сегмента MOV AX,ES:[100H] будет обращаться к дополнительному сегменту данных, базовый адрес которого находится в регистре ES.

Преобразование пары сегмент:смещение, называемой также логическим или виртуальным адресом, в физический адрес осуществляется довольно просто: содержимое сегментного регистра сдвигается влево на четыре бита (что эквивалентно умножению на 16), а затем суммируется со смещением. Если, например, регистр DS содержит 1234H, а регистр SI содержит 5678H, то физический адрес в команде MOV AX,[SI] следующий:

(DS) * 16 + (SI) = 12340H + 5678H = I79B8H.

Отметим два обстоятельства, связанные с сегментацией памяти в процессоре x86. Во-первых, преобразование логического адреса в физический всегда однозначно, т. е. каждому логическому адресу соответствует уникальный физический адрес. Однако обратное преобразование неоднозначно: каждому физическому адресу соответствуют 4К логических адресов. Поэтому во всех процессорах фирмы Intel манипуляции физическими адресами осуществляются довольно сложно. Во-вторых, в этом способе формирования физического адреса может возникнуть своеобразное переполнение. Пусть, например, регистр DS содержит 0FFFFH, тогда в команде MOV AX,[3000H] физический адрес

(DS) * 16 + 3000H = 0FFFF0H + 3000H = 102FF0H.

В такой ситуации процессор просто отбрасывает старший 21-й бит адреса и выдает на шину адрес 02FF0H. Другими словами, здесь троисходит так называемое «заворачивание» или «закругление» (wrap around) адреса: от адреса 0FFFFFH происходит переход к нулевому адресу. Заворачивание адреса приходится учитывать при работе процессора x86 в R- и V-режимах.

Простая сегментация памяти в процессоре 8086 обладает следующими особенностями, которые в известном смысле можно считать ее недостатками, усложняющими разработки мультизадачных систем.

1. Сегменты памяти, определяемые только одними сегментными регистрами, имеют всего два атрибута: начальный адрес, находящийся на границе параграфа, и максимальный размер 64 Кбайт. Никаких аппаратных средств контроля правильности использования сегментов нет.

2. Размещение сегментов в памяти произвольно: сегменты могут частично или полностью перекрываться или не иметь общих частей.

3. Программа может обращаться к любому сегменту для производства как считывания, так и записи данных, а также для выборки команд. В более общем плане программа может обратиться по любому физическому адресу, а для защиты определенных областей памяти от несанкционированного доступа требуются внешние схемы.

4. Нет никаких препятствий для обращения даже к физически несуществующей памяти. Когда программа выдает адрес несуществующей памяти, результат зависит только от особенностей схем дешифрирования адреса и управления внешней шиной: процессор может получить фиктивные данные, будет сформирован сигнал прерывания или система может просто зависнуть в ожидании данных, которые никогда не будут возвращены.

С учетом этих недостатков сегментация памяти сначала в процессоре 80286, а затем и в процессорах 80386 и i486 была значительно усовершенствована. В этих процессорах имеются гибкие средства организации сегментов с большим числом атрибутов и эффективные механизмы контроля и защиты доступа к сегментам. Кроме того, «ниже» сегментации действует внутренний механизм страничной организации памяти, который позволяет еще более гибко управлять ресурсами памяти компьютера.

3. Система команд.

Команды группируются по выполняемым функциям. Кратко опишем каждую команду и общие правила ее применения.

Очень важным для команд является понятие операнда. Это то значение данных, которое «обрабатывает» команда. Хотя каждая команда имеет свою структуру операндов, но имеется и много общих особенностей.

Операнд обычно рассматривается как источник или получатель (в зависимости от того, берет команда данные или помещает данные в него). Получателями могут быть только регистры или ячейки памяти, а источниками — еще и непосредственные данные. Одновременно и источником, и получателем ячейки памяти быть не могут.

Обычно операнды могут иметь любой размер (байт, слово или двойное слово). В большинстве команд с несколькими операндами все операнды имеют одинаковый размер.

Команды передачи данных. Команды этой группы пересылают данные из одного места компьютера в другое: между регистрами, между регистрами и памятью, между регистром или памятью и стеком.

MOV — пересылает один элемент данных из одного места в другое.

XCHG — обменивает содержимое двух регистров или содержимое регистра и ячейки памяти. Часто применяется для синхронизации нескольких процессов, так как ее нельзя прерывать другим устройством, использующим шину данных.

PUSH — копирует операнд-источник в вершину стека. Применяется для помещения параметров в стек перед вызовом процедуры. Полезна также для временного сохранения данных в стеке.

POP — берет верхний элемент из стека и пересылает его в операнд-получатель. Применяется для возвращения из стека значений, включенных в него командой PUSH.

PUSHA и PUSHAD — применяются для помещения содержимого всех 8 регистров общего назначения в стек (PUSHA — оперирует 16-битными регистрами, PUSHAD — 32-битными регистрами). Эти команды используются перед вызовом процедур.

POPA и POPAD — восстанавливают из стека содержимое 8 регистров общего назначения, являясь дополнением команд PUSHA и PUSHAD.

Арифметические команды. Эти команды применяются для выполнения арифметических операций над знаковыми или беззнаковыми числами. Они часто встречаются в программах.

ADD — суммирует два операнда, помещая результат в первый операнд (получатель).

SUB — вычитает один операнд (источник) из другого (получатель). Разность замещает место получателя.

INC — увеличивает операнд на 1, не воздействуя на флажок переноса. Применяется в циклах для инкремента индекса.

DEC — уменьшает операнд на 1, не воздействуя на флажок переноса. Применяется в циклах для декремента индекса.

MUL — эта простая команда умножает беззнаковые целые числа. Она имеет один операнд-источник. Еще два операнда умножения подразумеваются размером операнда-источника.

IMUL — умножает знаковые целые числа. Это более гибкая и сложная команда. Она имеет 4 базовые формы по числу и типу операндов (один, два или три операнда).

DIV — осуществляет деление целых беззнаковых чисел. В делении есть фактически 4 операнда: делимое, делитель, частное и остаток. Определяется только местоположение делителя. Все остальные операнды определяются неявно, в зависимости от размера делителя.

IDIV — аналогична команде DIV, но работает со знаковыми числами.

NEG — изменяет знак операнда, находящегося в регистре или памяти.

CMP — аналогична команде SUB, но не сохраняет результат. Команда сравнивает два числа для последующего условного перехода.

ADC — действует как команда ADD, но прибавляет значение переноса к сумме. Удобна для арифметики повышенной точности.

SBB — действует как команда SUB, но вычитает из разности значение переноса. Удобна для арифметики повышенной точности.

Команды преобразования данных. Команды этой группы применяются для преобразования типов данных. Большинство из них работает со знаковыми числами, а одна пригодна и для беззнаковых чисел.

MOVSX — команда пересылки с расширением знака, передает операнд-источник в получатель, расширяя знаковый бит источника в старшую часть получателя.

MOVZX — аналогична предыдущей команде, но старшая часть получателя устанавливается равной нулю.

CBW — команда преобразования байта в слово, имеет источником регистр AL, а получателем – регистр AH, действует аналогично команде MOVSX. Знаковый бит AL расширяется в AH, преобразуя знаковый байт из AL в знаковое слово в AX.

CWDE — преобразует знаковое слово из AX в знаковое двойное слово в EAX, расширяя знаковый бит AX в старшую половину EAX.

CWD — также преобразует знаковое слово в знаковое двойное слово, но результат, в отличие от команды CWDE помещается в два регистра. Знаковый бит из AX расширяется в регистр DX.

CDQ — эта команда заполняет регистр EDX знаковым битом регистра EAX.

Команды десятичной арифметики. Прямой поддержки десятичной арифметики процессор не обеспечивает, но имеет команды десятичной коррекции, действующие с обычными арифметическими командами. Есть два типа команд десятичной коррекции: команды ASCII-коррекции работают с одной цифрой в байте, а команды десятичной коррекции — с двумя цифрами.

AAA, AAS, AAM, AAD — команды ASCII-коррекции для арифметических операций. Три из них применяются после обычной арифметической команды, корректируя полученный результат: AAA — после сложения, AAS — после вычитания и AAM — после умножения. Команда AAD — коррекции деления, выполняется до операции и подготавливает операнды к делению.

DAA, DAS — команды десятичной коррекции для сложения и вычитания, применяются аналогично предыдущим.

Логические команды. Команды этой группы применяются для производства булевых операций и обеспечивают работу с двоичными полями в байтах, словах и двойных словах.

AND — выполняет логическую функцию И для двух операндов и удобна для установки двоичного поля в нуль.

OR — выполняет логическую функцию ИЛИ для двух операндов. Обычно применяется для установки двоичного поля в нужное состояние (поле предварительно очищается командой AND).

NOT — инвертирует биты своего операнда.

TEST — команда логического сравнения аналогична команде AND, но результат не сохраняется. Применяется для проверки двоичного поля на нуль (или не нуль).

XOR — выполняет логическую функцию ИСКЛЮЧАЮЩЕГО ИЛИ для двух операндов. Применяется для инвертирования в двоичном поле только определенных бит.

SETxx — эти команды применяются для сохранения результата некоторого сравнения. Значение «xx» определяет условие сравнения. Если сравнение истинно, то получатель устанавливается в 1, а если сравнение ложно, получатель устанавливается на 0.

Команды сдвига и циклического сдвига. Эти команды позволяют передвигать биты внутри любого из стандартных типов данных. Они применяются для ускорения операций умножения и деления целых чисел и образования двоичных полей.

SHR, SHL — команды логических сдвигов «вдвигают» нули с одного конца операндов, а биты с другого конца «выдвигаются».

SAR, SAL — команды арифметического сдвига. Арифметический сдвиг влево аналогичен логическому сдвигу, а при арифметическом сдвиге вправо происходит копирование знакового бита. Это удобно для деления знаковых чисел на степени числа 2.

ROR, ROL — при циклическом сдвиге биты данных не теряются: выдвигаемый бит помещается на место освобождающегося (вдвигаемого).

RCL, RCR — это команды циклического сдвига, включающего перенос. Выдвигаемый бит операнда помещается во флажок переноса, а старое значение флажка переноса передается в освобождающийся бит. Обычно применяются в операциях повышенной (кратной) точности.

SHRD, SHLD — это команды сдвигов двойной точности, но их действие отличается от команд циклического сдвига, включающих перенос. Эти команды имеют три операнда: источник, получатель и счетчик сдвигов. По-

лучатель сдвигается, и выдвигаемые биты теряются, а освобождающиеся биты заполняются битами из источника.

Команды операций над битами. Команды этой группы очень удобны для манипуляций отдельными битами, допуская их установку и проверку.

BT, BTS, BTR, BTC. Команда проверки бита BT просто помещает значение указанного бита во флажок переноса. Три другие команды выполняют эту же функцию, но позволяют адресуемый бит установить в 1 (BTS), сбросить в 0 (BTR) или инвертировать (BTC).

BSF, BSR — две команды сканирования бита: BSF (вперед) и BSR (назад) — находят первую 1 в операнде, начиная с младшего бита (BSF) или старшего бита (BSR). Обе команды удобны при работе с двоичными образами. Команда BSR применяется также при вычислении двоичных логарифмов.

Команды управления флажками. Команды этой группы позволяют проверять и изменять флажки процессора. Одни команды работают с отдельными флажками, другие — со всеми одновременно.

На отдельные флажки воздействуют 7 команд. Команды CLD и STD сбрасывают и устанавливают флажок направления. Команды CLI и STI сбрасывают и устанавливают флажок направления. Команды CLC и CTC сбрасывают и устанавливают флажок переноса, а команда CMC инвертирует его.

Команда LAHF загружает младший байт флажков в регистр AH. Команда SAHF производит обратную передачу. Команды PUSHF и PUSFD включают регистры Flags и EFlags в стек. Соответствующие операции извлечения из стека реализуют команды POPF и FOPFD.

Циклические команды. Часто приходится обрабатывать большие блоки данных, и в этом помогают команды данной группы. Одна циклическая команда может реализовать цикл, требующий обычно нескольких команд.

Каждая из этих команд обычно выполняет одну операцию пересылки, сравнения, загрузки, запоминания или сканирования. Для адресации источника применяется регистр (E)SI, а получателя — регистр (T)DI. После команды оба регистра модифицируются для адресации следующего элемента цикла. Данные команды наиболее полезны при использовании с одним из префиксов повторения.

REP, REPE, REPZ, REPNE, REPNZ — префиксы повторения, обеспечивают выполнение следующей за ними команды заданное число раз. Префикс повторения REP вызывает выполнение команды столько раз, сколько определено содержимым регистра ECX. Аналогичные префиксы REPE и REPZ вызывают повторение команды либо до исчерпания счетчика, либо до установки флажка Z в нуль. Префиксы REPNE и REPNZ действуют аналогично REPE, но выход происходит при установке флажка Z в 1.

MOVS — эта команда просто пересылает фрагмент данных из одной области памяти в другую. Необходимо тщательно проанализировать возможность перекрытия фрагментов и соответственно определить состояние флажка направления.

CMPS — сравнивает два фрагмента (массива данных). Для окончания сравнения в нужной точке применяются префиксы REPE или REPNE.

STOS — полезна для заполнения фрагмента константой. Она передает содержимое соответствующей части регистра EAX в каждый элемент фрагмента.

SCAS — сравнивает каждый элемент данных с частью регистра EAX. Обычно с этой командой применяется префикс REPE или REPNE.

LODS — эта команда необычна в том, что она бесполезна с префиксом повторения. Она просто загружает следующий элемент данных в некоторую часть EAX. Следующий пример показывает общие принципы группы циклических команд.

Команды управления программой. При создании программ часто приходится изменять последовательный ход исполнения команд. Команды данной группы обеспечивают необходимые средства управления ходом выполнения программы.

JMP — команда безусловного перехода. Обычно в ней указывается метка, определяющая следующую выполняемую команду. Ассемблер формирует нужную форму JMP, зная местонахождение именованного (отмеченного) оператора.

Jxx — команда условного перехода, в которой xx означает код проверяемого условия. Обычно перед ней находится команда сравнений или другая команда, которая устанавливает некоторые флажки.

CALL — команда вызова. Применяется для реализации процедур (называемых еще подпрограммами и функциями). Команда вызова передает управление так же, как команда JMP. Однако до ее выполнения в стек включается необходимая информация, чтобы вызванная процедура могла возвратить управление команде, находящейся после команды CALL. Для выполнения возврата управления применяется команда RET.

LOOP, LOOPE, LOOPZ, LOOPNE, LOOPNZ — команды зацикливания. Помогают в реализации программных циклов. Команда LOOP повторяет цикл такое число раз, которое определяется содержимым регистра ECX. В командах LOOPE и LOOPZ имеется дополнительное ограничение: цикл заканчивается, когда флажок Z будет нулевым. Команды LOOPNE и LOOPNZ аналогичны предыдущим, но цикл заканчивается, когда флажок Z будет содержать 1.

INT, INTO, IRET, IRETD — последняя группа команд передачи управления. Связана с прерыванием. Команда INT инициирует выполнение системной процедуры. Команда INTO вызывает процедуру прерывания 4, если установлен флажок переполнения. Обычно процедуры обработки прерываний входят в операционную систему. Возврат в вызывающую программу осуществляет одна из команд возврата из прерывания IRET или IRETD.

Команды поддержки языка высокого уровня. Обычно команды этой группы генерируются компиляторами, а программисты на ассемблере их не применяют. Такие команды упрощают разработку компиляторов и обеспечивают аппаратную поддержку некоторых общих операций.

BOUND — применяется для контроля границ массива. Операндами служат значение, нижняя граница и верхняя граница. Если значение не находится внутри границ, возникает прерывание.

ENTER, LEAVE — парные команды, которые сокращают подготовительное время в начале процедуры. Команда ENTER организует стек в начале исполнения процедуры для упрощения доступа к аргументам и к переменным во взаимосвязанных процедурах. Команда LEAVE восстанавливает стек, подготавливая возврат управления.

Команды управления процессором. Команды данной группы управляют действиями CPU.

ESC — префикс ESC информирует о том, что следующая команда предназначена для сопроцессора.

WAIT — заставляет CPU остановить выполнение команд до получения сигнала BUSY. Применяется при ожидании результата от сопроцессора.

LOCK — префикс блокировки. Резервирует системную шину за процессором 80386 на время выполнения следующей команды. Применяется для устранения конфликтов на шине в мультипроцессорных системах. Примечание: префикс LOCK в 80386 имеет несколько ограничений.

NOP — пустая команда; иногда применяется при отладке.

HLT — блокирует работу процессора до получения сигнала сброса. Применяется редко.

Команды операций с адресами. Команды этой группы применяются для загрузки указателей. Имеются два типа команд. К первому типу относится одна команда LEA, которая загружает в регистр смещение указанной ячейки памяти. Она удобна для загрузки индексного регистра. Ко второму типу относятся команды загрузки полного указателя: они загружают сегментный регистр и индексный регистр. В сегментный регистр загружается селектор сегмента ячейки памяти, а в индексный регистр — смещение ячейки памяти. Мнемоника команд имеет вид Lxx, где xx есть имя одного из сегментных регистров.

Команда преобразования. Команда преобразования XLAT выделена в отдельную группу. Она осуществляет табличное преобразование. Предполагается, что AL содержит байтный индекс таблицы, адресуемой регистром (E)BX. Байт в AL заменяется элементом таблицы. Команда применяется для преобразования символьного кода и синтаксического разбора команд.

ЛЕКЦИЯ 6
РЕШЕНИЕ ВЫЧИСЛИТЕЛЬНЫХ ЗАДАЧ В АССЕМБЛЕРЕ X86

План лекции

1. Ввод и вывод информации на ассемблере.
2. Вычисление выражений.
3. Реализация многоразрядной арифметики.
4. Организация циклов в ассемблере. Реализация вложенных циклов.

1. Ввод и вывод информации на ассемблере.

В языках ассемблера отсутствуют готовые процедуры ввода-вывода. Для выполнения этих операций существуют следующие варианты:

1. Напрямую обращаться к устройствам ввода-вывода. Этот способ является единственным в случае программирования «голой» машины, т. е. когда полностью отсутствуют готовые процедуры для работы с внешними устройствами.
2. Использование процедур BIOS, размещенных в ПЗУ и соответственно постоянно присутствующих в ПК (обращение к функциям BIOS). Этот способ используется при программировании в отсутствие операционной системы.
3. Обращение к сервисам ОС с запросами на ввод-вывод для соответствующих устройств. Этот вариант является наиболее предпочтительным и часто используемым, и в дальнейшем будет рассматриваться именно он.

Любая операционная система обязательно обеспечивает символьный ввод-вывод и операции для работы с файлами.

Символьный ввод-вывод. Для ввода символа достаточно воспользоваться функцией 01 DOS:

```
Mov     ah,01 ; номер функции помещаем в ah
Int     21h   ; передаем управление DOS
```

Далее ОС помещает код нажатой клавиши в регистр al и возвращает управление программе.

Для вывода символа предназначена функция 02:

```
Mov     ah,02 ; номер функции помещаем в ah
Mov     dl,'*' ; код выводимого символа помещаем в dl
Int     21h   ; передаем управление DOS
```

Далее ОС выводит в текущую позицию экрана символ, код которого находится в dl (в данном случае на экран будет выведена звездочка) и возвращает управление программе.

Вывод строки. Для вывода строки на экран можно организовать цикл посимвольного вывода с помощью рассмотренной ранее функции 02:

```
Msg     db      'Этот текст должен появиться на экране' ; размещаем (в
; сегменте данных) выводимую строку
Len = $-Msg         ; Len присваиваем значение длины строки (разница между
;текущим смещением в сегменте и смещением до Msg)

        Mov  cx,Len         ; инициализируем счетчик цикла
  Lea   si,Msg        ; устанавливаем si на начало области Msg (на
; первый байт выводимой строки)
M:      Mov  ah,02          ; номер функции помещаем в ah
        Mov  dl,[si]        ; код выводимого символа помещаем в dl
        Int  21h            ; передаем управление DOS
        Inc  si             ; продвигаем si к следующему символу (байту)
        Loop M              ; организуем цикл с меткой M
```

Другим способом вывода строки на экран является использование функции 09:

```
Msg   db    'Этот текст должен появиться на экране$'      ; размещаем
;(в сегменте данных) выводимую строку, последний знак $ - ограничитель
;строки для DOS

        Mov     ah,09         ; номер функции помещаем в ah
        Leadx,Msg     ;смещение в сегменте до Msg помещаем в dx
        Int  21h            ; передаем управление DOS
```

Далее ОС выводит, начиная с текущей позиции экрана, символы, коды которых адресуются ds:dx до тех пор, пока не встретит код знака $, после чего управление возвращается программе.

Ввод строки. Для ввода строки можно использовать цикл посимвольного ввода с помощью функции 01. Приведенный ниже пример предполагает завершение строки нажатием на клавишу «Enter» и предназначен для ввода строк не длиннее Len байтов (все последующие символы игнорируются):

```
        Len = 256           ; пусть для определенности строка не может
; превышать 256 байт
Str             db    Len dup (?) ; резервируем область памяти под вводи-
мую
; строку (в сегменте данных)
```

```
        cr  = 0dh           ; присваиваем cr значение кода клавиши «Enter»

        Mov  cx,Len         ; инициализируем счетчик цикла
        Lea  di,str         ; устанавливаем di на начало области Str
M:      Mov  ah,01          ; номер функции помещаем в ah
        Int  21h            ; передаем управление DOS
        Mov  [di],al        ; код  введенного символа помещаем в
; очередной байт области Str
        Cmp  al,cr          ; проверяем, не является ли полученный код
; кодом клавиши «Enter»
        Je   Ex             ; если да – ввод завершен, выходим из цикла
        Inc  di             ; продвигаем di к следующему байту Str
        Loop M              ; организуем цикл
Ex:                         ; при выходе из цикла cx содержит количество
; незаполненных байтов в Str
```

Ввести строку символов не длиннее 255 байтов можно также с помощью функции 0a:

```
Str         db   255, ?, 255 dup (?); резервируем область памяти под
; вводимую строку (в сегменте данных). Первый байт – максимальная длина
; строки, второй – фактическая длина (заполняется операционной системой
; в процессе ввода), далее – собственно строка символов

        Mov  ah,0a          ; номер функции помещаем в ah
        Lea  dx,Str         ;смещение в сегменте до Str помещаем в dx
        Int  21h            ; передаем управление DOS
```

Ввод-вывод чисел. Для ввода-вывода числовых данных операционная система не предоставляет никаких возможностей, поэтому преобразование вводимых символов в число и обратно полностью возлагается на программу.

Преобразование числа в строку символов производится последовательным целочисленным делением на основание системы счисления до получения нуля и сохранением остатков от деления. Полученные остатки и являются цифрами, которыми записывается число. Далее необходимо преобразовать цифры в коды соответствующих символов (добавить 48 – код символа ‘0’) и в обратном порядке вывести их на экран. Рассмотрим это преобразование на примере числа 6543 и десятичной системы счисления (табл. 6.1).

Таблица 6.1

Деление	Частное	Остаток	Код символа	Символ
6543 : 10	654	3	51	‘3’
654 : 10	65	4	52	‘4’
65 : 10	6	5	53	‘5’
6 : 10	0	6	54	‘6’

Вывод беззнаковых чисел. Данная операция иллюстрируется приведенной ниже процедурой, которая содержимое регистра ax преобразует в строку символов и выводит ее на экран в десятичной системе счисления, интерпретируя как число без знака (число в диапазоне от 0 до 65535):

```
UnsignedOut  proc
             xor  cx,cx    ;обнуляем счетчик цифр
             mov  bx,10    ;в bx помещаем делитель
     ;(основание системы счисления)
m:           inc  cx       ;считаем количество
     ;получающихся цифр
             xor  dx,dx    ;преобразуем делимое к 32
     ;разрядам
             div  bx       ;получаем очередную цифру
             pushdx        ;сохраняем ее в стеке
             or   ax,ax    ;проверяем, есть ли еще цифры
             jnz  m        ;если да - на метку m
;при выходе из цикла в стеке лежат цифры, в cx - их
;количество
m1:          pop  dx       ;извлекаем цифру из стека
             add  dx,'0'   ;преобразуем в код символа
             mov  ah,2     ;функцией 02 выводим на экран
             int  21h
             loopm1        ;повторяем cx раз
             ret           ;возвращаемся из процедуры
UnsignedOut  endp
```

Вывод целых чисел. Вывод целых чисел отличается от вывода беззнаковых чисел тем, что число может быть отрицательным. В этом случае на экране вначале должен появиться символ ‘-’, а затем цифры, показывающие абсолютную величину числа. Данная операция иллюстрируется приведенной ниже процедурой, которая содержимое регистра ax преобразует в строку символов и выводит ее на экран в десятичной системе счисления, интерпретируя как целое число (число в диапазоне от – 32768 до 32767):

```
IntegerOut   proc
             xor  cx,cx    ;обнуляем счетчик цифр
             mov  bx,10    ;в bx помещаем делитель
             cmp  ax,0     ;проверяем знак числа
             jge  m        ;если неотрицательное - на m
             neg  ax       ;иначе - меняем знак числа
             pushax        ;сохраняем число перед вызовом
                           ;функции, использующей ax
             mov  ah,2     ;функцией 02 выводим знак '-'
             mov  dl,'-'
```

```
                int 21h
                pop ax          ;восстанавливаем число в ax
m:              inc cx          ;считаем количество
    ;получающихся цифр
                xor dx,dx       ;преобразуем делимое к 32
    ;разрядам
                div bx          ;получаем очередную цифру
                pushdx          ;сохраняем ее в стеке
                or  ax,ax       ;проверяем есть ли еще цифры
                jnz m           ;если да - на метку m
;при выходе из цикла в стеке лежат цифры, в cx - их
;количество
m1:             pop dx          ;извлекаем цифру из стека
                add dx,'0'      ;преобразуем в код символа
                mov ah,2        ;функцией 02 выводим на экран
                int 21h
                loopm1          ;повторяем cx раз
                ret             ;возвращаемся из процедуры
IntegerOut      endp
```

Ввод чисел сопровождается преобразованием строки символов в число. Данное преобразование производится получением последовательности цифр из символов (вычитанием 48 – кода символа ’0’) и последующим сложением цифр с соответствующими весами. Данная операция описывается формулой (для числа из пяти цифр):

m=((((s1-48)*n+(s2-48))*n+(s3-48))*n+(s4-48))*n+(s5-48) , где
m – получаемое число,
n – основание системы счисления,
s1,s2,s3,s4,s5 – коды первого, второго, третьего, четвертого и пятого символов строки соответственно.

Например строка ‘31562’ для десятичной системы счисления:
s1=51, s2=49, s3=53, s454=, s5=50
s1-48=3
(s1-48)*10+(s2-48)=31
((s1-48)*10+(s2-48))*10+(s3-48)=315
(((s1-48)*10+(s2-48))*10+(s3-48))*10+(s4-48)=3156
((((s1-48)*10+(s2-48))*10+(s3-48))*10+(s4-48))*10+(s5-48)=31562

Ввод беззнаковых чисел. Ввод беззнаковых чисел в десятичной системе счисления в диапазоне от 0 до 65535 иллюстрируется нижеприведенной процедурой, размещающей введенное число в регистре ax:

```
;Данная прцедура требует ввода числа в виде строки, содержащей только
;цифры и оканчивающейся клавишей «Enter»
cr = 0dh ;cr присваиваем значение кода символа
;возврата каретки (клавиши «Enter»)
lf = 0ah ;lf присваиваем значение кода символа
;перевода строки

UnsignedIn  proc
start:      mov ah,0ah        ;функцией 0a вводим строку
                              ;символов и размещаем ее в
                              ;области string
    lea dx,string
    int 21h

            xor ax,ax         ;обнуляем ax, в котором
    ;будем формировать число
            lea si,string+2   ;устанавливаем si на
;первый символ введенной ;строки
;анализируем текущий символ
m:          cmp byte ptr [si],cr ;если это cr – строка
    ;закончилась, выходим
            je  ex
            cmp byte ptr [si],'0';если код символа
;меньше кода '0' – ;это не цифра
            jb  err               ;прыгаем на метку err
            cmp byte ptr [si],'9';если код символа
;больше кода '9' – ;это не цифра
            ja  err               ;прыгаем на метку err

            mov bx,10             ;домножаем полученное
;число на основание ;системы счисления
            mul bx
            sub byte ptr [si],'0';вычитаем код символа
    ;'0' (получаем
    ;очередную цифру)
            add al,[si]           ;добавляем цифру к
    ;числу
            adc ah,0
            inc si                ;продвигаем si к
    ;следующему символу
            jmp m                 ;организуем цикл
;функцией 09 выводим сообщение об ошибке
err:        lea dx,errmsg
            mov ah,9
            int 21h
```

```
                jmp start          ;повторяем ввод
ex:             ret
UnsignedIn      endp

;в сегменте данных описываем область string для
;вводимой строки и сообщение об ошибке errmsg
string          db  255, 0, 255 dup (?)
errmsg          db  'Недопустимый символ, можно'
    db  'использовать только  цифры',cr,lf,'$'
```

Ввод целых чисел. Ввод целых чисел отличается от ввода беззнаковых чисел тем, что перед ними может указываться знак числа. Само преобразование строки цифр в число производится по обычным правилам, однако если перед первой цифрой указан знак минус, у полученного числа надо сменить знак. Ввод целых чисел в десятичной системе счисления в диапазоне от – 32768 до 32767 иллюстрируется нижеприведенной процедурой, размещающей введенное число в регистре ax:

```
;Данная процедура требует ввода числа в виде строки, в первой позиции
;которой может присутствовать знак '+' или '-', далее - только
;цифры и оканчивающейся клавишей Enter.
    cr = 0dh ;cr присваиваем значение кода символа
    ;возврата каретки (клавиши «Enter»)
    lf = 0ah ;lf присваиваем значение кода символа
    ;перевода строки

IntegerIn   proc
start:      mov ah,0ah          ;функцией 0a вводим строку
                        ;символов и размещаем ее в
                                ;области string
    lea dx,string
    int 21h

                xor ax,ax       ;обнуляем ax, в котором
    ;будем формировать число
                lea si,string+2 ;устанавливаем si на
;первый символ введенной ;строки

                mov negflag,ax  ;обнуляем флаг
    ;отрицательности числа
    ;(предполагаем, что оно
    ;будет неотрицательным)
                cmp byte ptr [si],'-';первый символ – это
    ;минус?
                jne m2                ;если нет – на m2
```

```
                not negflag             ;отмечаем, что число
;отрицательное ;(negflag не равен 0)
                inc si                  ;продвигаем si со
    ;знака числа к первой
    ;цифре
                jmp m                   ;прыгаем на разбор
    ;строки цифр
m2:             cmp byte ptr [si],'+';первый символ – это
    ;плюс?
                jne m                   ;если нет – на m
                inc si                  ;продвигаем si со
    ;знака числа к первой
    ;цифре
;анализируем текущий символ
m:              cmp byte ptr [si],cr ;если это cr – строка
;закончилась, выходим ;из цикла разбора ;символов
                je  ex1
                cmp byte ptr [si],'0';если код символа
;меньше кода '0' – ;это не цифра
                jb  err                 ;прыгаем на метку err
                cmp byte ptr [si],'9';если код символа
;больше кода '9' – ;это не цифра
                ja  err                 ;прыгаем на метку err

                mov bx,10               ;домножаем полученное
;число на основание ;системы счисления
                mul bx
                sub byte ptr [si],'0';вычитаем код символа
    ;'0' (получаем
    ;очередную цифру)
                add al,[si]             ;добавляем цифру к
    ;числу
                adc ah,0
                inc si                  ;продвигаем si к
    ;следующему символу
                jmp m                   ;организуем цикл

;функцией 09 выводим сообщение об ошибке
err:            lea dx,errmsg
                mov ah,9
                int 21h
                jmp start           ;повторяем ввод

ex1:            cmp negflag,0       ;число положительное?
                je  ex              ;если да – выходим
```

```
            neg ax              ;меняем знак числа
ex:         ret
IntegerIn   endp

;в сегменте данных описываем область string для
;вводимой строки, сообщение об ошибке errmsg и флаг
;отрицательности числа negflag (0 – неотрицательное,
;0ffffh - отрицательное)
string      db  255, 0, 255 dup (?)
errmsg      db  'Недопустимый символ, можно'
    db  'использовать только  цифры',cr,lf,'$'
negflag     dw  ?
```

Обратите внимание: перед практическим использованием приведенных процедур ввода чисел их надо доработать: ввести контроль переполнения разрядной сетки.

2. Вычисление выражений.

В ассемблере понятие выражения существенно отличается от языков высокого уровня. Выражения в ассемблере рассчитываются при трансляции, и в них могут использоваться только имена, значения которых известны транслятору. При этом следует помнить, что значениями меток (и имен областей памяти) являются смещения от начала сегмента, и они могут участвовать в выражениях. В ассемблерных выражениях можно использовать четыре целочисленные арифметические операции - +, -, *, /, и четыре логические – and, or, not, xor. В исполняемом модуле вместо выражений будут присутствовать константы, являющиеся результатом вычисления значений этих выражений транслятором.

Выражения, в которых участвуют значения переменных, реализуются с помощью соответствующей последовательности команд. Например, программа вычисления

$$Y = ((a+b*c)^2+10)/3,$$

где a, b, c – целочисленные переменные, на ассемблере может выглядеть следующим образом:

```
title     prim
assume    cs:cod, ds:d, ss:s

;Обратите внимание. Данная программа не контролирует
;переполнение разрядной сетки
```

```
s           segment     stack
            dw    128 dup (?)
s           ends

d           segment

;резервируем области под переменные a, b, c
a           dw    ?
b           dw    ?
c           dw    ?

;размещаем строки – подсказки для ввода переменных
msga        db    ‘Введите a:$’
msgb        db    ‘Введите b:$’
msgc        db    ‘Введите c:$’

;описываем данные для процедур ввода и вывода целых чисел
                  .
                  .
                  .
d           ends

c od        segment

;описываем процедуры ввода и вывода целых чисел
IntegerIn   proc     ;процедура ввода целого числа
                  .
                  .




                  .
IntegerIn   endp

IntegerOut  proc     ;процедура вывода целого числа
                  .
                  .
                  .
IntegerOut  endp

;устанавливаем ds на сегмент данных
start:      mov   ax,d
            mov   ds,ax
;вводим значение переменныой a
```

```
                Mov   ah,09         ;выводим строку – подсказку для a
                Lea   dx,msga
                Int   21h
                Call  IntegerIn     ;вводим число
                Mov   a,ax          ;помещаем его в область a

;вводим значение переменныой b
                Mov   ah,09         ;выводим строку – подсказку для b
                Lea   dx,msgb
                Int   21h
                Call  IntegerIn     ;вводим число
                Mov   b,ax          ;помещаем его в область b

;вводим значение переменныой c
                Mov   ah,09         ;выводим строку – подсказку для c
                Lea   dx,msgc
                Int   21h
                Call  IntegerIn     ;вводим число
                Mov   c,ax          ;помещаем его в область c

;рассчитываем значение выражения
                mov   ax,b          ;умножаем b на c
                mov   bx,c
                imul  bx
                add   ax,a          ;добавляем a
                imul  ax            ;возводим в квадрат
                add   ax,10         ;увеличиваем на 10
                mov   bx,3          ;делим на 3
                cwd
                idiv  bx
                call  IntegerOut    ;выводим результат

;завершаем работу программы
                mov   ax,4c00h
                int   21h
c od            ends
                end   start
```

Еще один пример: программа вычисления

$$Y = (a+b)/c - 3a^2/(b+c),$$

где a, b, c – целочисленные переменные.

```
                title   prim
```

```
                assume      cs:cod, ds:d, ss:s

;Обратите внимание. Данная программа не контролирует
;переполнение разрядной сетки

s               segment     stack
                dw      128 dup (?)
s               ends

d               segment

;резервируем области под переменные a, b, c
a               dw      ?
b               dw      ?
c               dw      ?

;размещаем строки – подсказки для ввода переменных
msga            db      'Введите a:$'
msgb            db      'Введите b:$'
msgc            db      'Введите c:$'

;размещаем строки – сообщения об ошибках
err1            db      'Деление на ноль (первая дробь)$'
err2            db      'Деление на ноль (вторая дробь)$'

;описываем данные для процедур ввода и вывода целых чисел
                .
                .
                .
d               ends

c od            segment

;описываем процедуры ввода и вывода целых чисел
IntegerIn       proc    ;процедура ввода целого числа
                .
                .
                .
IntegerIn       endp

IntegerOut      proc    ;процедура вывода целого числа
                .
                .
                .
IntegerOut      endp
```

```
;устанавливаем ds на сегмент данных
start:      mov   ax,d
            mov   ds,ax
;вводим значение переменныой a
            Mov   ah,09       ;выводим строку – подсказку для a
            Lea   dx,msga
            Int   21h
            Call  IntegerIn   ;вводим число
            Mov   a,ax        ;помещаем его в область a

;вводим значение переменныой b
            Mov   ah,09       ;выводим строку – подсказку для b
            Lea   dx,msgb
            Int   21h
            Call  IntegerIn   ;вводим число
            Mov   b,ax        ;помещаем его в область b

;вводим значение переменныой c
            Mov   ah,09       ;выводим строку – подсказку для c
            Lea   dx,msgc
            Int   21h
            Call  IntegerIn   ;вводим число
            Mov   c,ax        ;помещаем его в область c

;проверяем знаменатели на равенство 0
            Cmp   c,0         ;проверяем первый знаменатель
            Jnz   m           ;если не 0 – на m
            Mov   ah,09       ;выводим сообщение об ошибке
            Lea   dx,err1
            Int   21h
            Jmp   err         ;выходим
m:          Mov   ax,b        ;считаем второй знаменатель
            Add   ax,c
            Jnz   m1          ; если не 0 – на m1
            Mov   ah,09       ;выводим сообщение об ошибке
            Lea   dx,err2
            Int   21h
            Jmp   err         ;выходим

;рассчитываем значение выражения
m1:         Mov   bx,ax       ;помещаем второй знаменатель в bx
            Mov   ax,a        ;считаем второй числитель
            Imul  ax
            Mov   cx,3
```

```
                Imul   cx
                Idiv   bx          ;считаем значение второй дроби
                Push   ax          ;сохраняем его в стеке
                Mov    ax,a        ;считаем первый числитель
                Add    ax,b
                Mov    bx,c        ;считаем значение первой дроби
                Cwd
                Idiv   bx
                Pop    bx          ;извлекаем из стека значение второй
    ;дроби
                Sub    ax,bx       ;вычитаем
                call   IntegerOut  ;выводим результат

    ;завершаем работу программы
                mov    ax,4c00h    ;с кодом завершения 0 – без ошибок
                int    21h
    err:        mov    ax,4cffh    ;с кодом завершения 0ffh (-1) – с ошибкой
                int    21h
    c od        ends
                end    start
```

3. Реализация многоразрядной арифметики.

При решении вычислительных задач необходимо учитывать допустимый диапазон значений обрабатываемых данных. В языках высокого уровня этот вопрос решается выбором соответствующего типа данных. В ассемблере тип данных фактически сводится к разрядности операндов:

байт – резервируется директивой db;
слово (2 байта) – резервируется директивой dw;
двойное слово (4 байта) – резервируется директивой dd;
квадрослово (8 байтов) – резервируется директивой dq.

Иногда возникает ситуация, когда требуемая разрядность операндов превышает разрядность процессора. В этом случае нужно программно реализовать выполнение операций соответствующей разрядности. Для примера рассмотрим выполнение 32-разрядных операций в 16-разрядной сетке процессора.

В основе многоразрядных сложения, вычитания и умножения лежит представление операндов в виде двух слагаемых:

$$X = Xh*2^{16}+Xl,$$

где X – двойное слово; Xh – старшее слово X; Xl – младшее слово X.

Тогда ***сложение*** двухсловных X и Y выглядит следующим образом:

$$X+Y = Xh*2^{16}+Xl+Yh*2^{16}+Yl = (Xh+Yh)*2^{16} +(Xl+Yl),$$

и фактически сводится к попарному сложению младших и старших слов слагаемых. При этом следует учитывать возможность появления переноса при суммировании младших слов, который необходимо добавить к сумме старших слов. Соответствующий фрагмент программы:

```
;в сегменте данных резервируем области под операнды
X               dd      65538
Y               dd      76543
                ……..
;выполняем сложение X+Y
                Mov   ax, word ptr X      ; помещаем в ax младшее слово X
                Mov   dx, word ptr X+2    ; помещаем в dx старшее слово X
                Add   ax, word ptr Y      ; добавляем к ax младшее слово Y
                Adc   dx, word ptr Y+2    ; добавляем к dx старшее слово Y
```

В результате в регистровой паре ax,dx получится 32-разрядная сумма: в ax – младшее слово, в dx – старшее.

Вычитание двухсловных X и Y выглядит аналогично:

$$X–Y = (Xh*2^{16}+Xl) – (Yh*2^{16}+Yl) = (Xh – Yh)*2^{16} +(Xl – Yl),$$

и фактически сводится к попарному вычитанию младших и старших слов операндов. При этом следует учитывать возможность появления заема при вычитании младших слов, который необходимо вычесть из разности старших слов. Соответствующий фрагмент программы:

```
;в сегменте данных резервируем области под операнды
X               dd      65538
Y               dd      76543
                ……..
;выполняем вычитание X–Y
                Mov   ax, word ptr X      ; помещаем в ax младшее слово X
                Mov   dx, word ptr X+2    ; помещаем в dx старшее слово X
                Sub   ax, word ptr Y      ; вычитаем из ax младшее слово Y
                Sbc   dx, word ptr Y+2    ; вычитаем из dx старшее слово Y
```

В результате в регистровой паре ax,dx получится 32-разрядная разность: в ax – младшее слово, в dx – старшее.

Умножение двухсловных X и Y содержит больше операций

$X*Y = (Xh*2^{16}+Xl)*(Yh*2^{16}+Yl) = Xh*2^{16} *Yh*2^{16}+ Xh*2^{16} *Yl+ Xl* Yh*2^{16}+ Xl*Yl = Xh *Yh*2^{32}+ (Xh*Yl+ Xl* Yh)*2^{16}+ Xl*Yl$

и результат в общем случае состоит из 4-х слов. Соответствующий фрагмент программы для беззнаковых X и Y:

```
;в сегменте данных резервируем области под операнды
X           dd    65538
Y           dd    76543
tmp1        dd    0
tmp2        dd    0
tmp3        dd    0

            ……..
;выполняем умножение X*Y
            Mov   ax, word ptr X+2
            Mov   bx, word ptr Y+2
            Mul   bx
            Mov   word ptr tmp1,ax
            Mov   word ptr tmp1+2,dx

            Mov   ax, word ptr X+2
            Mov   bx, word ptr Y
            Mul   bx
            Mov   word ptr tmp2,ax
            Mov   word ptr tmp3+2,dx

            Mov   ax, word ptr X
            Mov   bx, word ptr Y+2
            Mul   bx
            Mov   word ptr tmp2,ax
            Mov   word ptr tmp3+2,dx

            Mov   ax, word ptr X
            Mov   bx, word ptr Y
            Mul   bx

            Xor   bx,bx
            Xor   cx,cx

            Add   dx, word ptr tmp3
            Adc   cx, word ptr tmp3+2
            Add   dx, word ptr tmp2
            Adc   cx, word ptr tmp2+2
            Adc   bx,0
```

```
Add   cx, word ptr tmp1
Adc   bx, word ptr tmp1+2
```

Результат разместился в регистровой четверке bx, cx, dx, ax.

Деление многоразрядных чисел организуется путем последовательного вычитания делителя из делимого с подсчетом количества вычитаний до получения остатка, меньшего, чем делитель.

4. Организация циклов в ассемблере. Реализация вложенных циклов.

В ассемблере нет операторов циклов, характерных для высокоуровневого программирования, поэтому циклы организуются с помощью команд условных переходов:

C++	Pascal	Ассемблер
while (a>-3)	while a>-3 do	m: Cmp a,-3
{	begin	Jle ex
…	…	…
}	end	Jmp m
…	…	ex: …

В системе команд процессоров x86 имеется специальная команда для организации циклов с заданным числом повторений – loop. С ее помощью обычно организуются циклы, аналогичные высокоуровневым циклам for. Типичная структура такого цикла:

```
            Mov  cx,n   ; в cx помещаем количество повторений
m:          ……
            ……
            Loop m      ; cx уменьшается на 1 и, если он не равен 0, на
; метку m
```

В случае когда необходимо организовать вложенные циклы с использованием одного регистра cx, типичным приемом является использование стека:

```
            Mov  cx,m
m1:         push cx     ;тело внешнего цикла с m повторениями
            mov  cx,n
m2:         ……          ;тело внутреннего цикла с n повторениями
            ……
  loop  m2
            pop  cx
            loop m1
```

Обработка целочисленных массивов и матриц. Приведенные способы организации циклов обычно применяются при обработке массивов и матриц. Рассмотрим несколько примеров.

Цикл отыскания местоположения максимального элемента одномерного целочисленного массива (n – количество элементов, a – массив):

```
                Mov   cx,n
                Xor   si,si
                Mov   ax,a
                Xor   bx,bx
m:      cmp     ax,a[si]
                jge   m1
                mov   ax,a[si]
                mov   bx,si
m1:     add     si,2
                loop  m
```

После выполнения данного фрагмента в ax останется значение максимального элемента, в bx – смещение до него от начала массива.

Цикл сортировки одномерного целочисленного массива (n – количество элементов, a – массив):

```
m:      Mov   cx,n
        Dec   cx
        Xor   si,si
        Xor   bp,bp
m1:     mov   ax,a[si]
        mov   bx,a+2[si]
        cmp   ax,bx
        jge   m2
        mov   a+2[si],ax
        mov   a[si],bx
        mov   bp,1
m2:     inc   si
        inc   si
        loop  m1
        or    bp,bp
        jnz   m
```

После выполнения этого цикла элементы массива расположатся по невозрастанию.

При обработке матриц требуется организовать вложенные циклы и использовать более сложную адресацию. В качестве примера рассмотрим цикл сортировки строк целочисленной матрицы a из m строк и n столбцов, m<=10, n<=15 :

```
a       dw    10 dup (15 dup (?))

              Mov   cx,m
              Xor   bx,bx
m1:           Push  cx
m2:           Mov   cx,n
              Dec   cx
              Xor   si,si
              Xor   bp,bp
m3:           Mov   ax,a[bx+si]
              Mov   bx,a+2[bx+si]
              Cmp   ax,bx
              Jge   m4
              Mov   a[bx+si],bx
              Mov   a+2[bx+si],ax
              Inc   bp
m4:           Inc   si
              Inc   si
              Loop  m3
              Or    bp,bp
              Jnz   m2
              Add   bx,15*2
              Pop   cx
              Loop  m1
```

После выполнения этого цикла элементы каждой строки матрицы расположатся по невозрастанию.

ЛЕКЦИЯ 7
РАБОТА С ФАЙЛАМИ В АССЕМБЛЕРЕ X86

План лекции

1. Основные понятия файловых систем.
2. Средства взаимодействия программ с ОС.
3. Пример программы

1. Основные понятия файловых систем.

Файловая система – это совокупность:
файлов;
системных таблиц;
методов доступа к файлам;
способов организации файлов;
процедур доступа к файлам.

Файлы. Существует два понятия файла: общепринятое и используемое в программировании. Общепринятое понятие файла – это область ВЗУ, имеющая собственное имя и содержащая логически законченную совокупность однородной информации. В программировании особенно в высокоуровневом файл – это линейная последовательность логических записей. Логическая запись – это минимальный объем информации, который может быть считан или записан в файл. В большинстве операционных систем в качестве логической записи выступает байт.

В каждой файловой системе существует как минимум две разновидности таблиц: каталоги и таблицы открытых файлов. Поскольку на данном этапе изучения рассматриваются 16-разрядные приложения, для определенности будем ориентироваться на простейшую 16-разрядную ОС – MS DOS.

В ней используются 3 разновидности системных таблиц: каталоги, FAT (Files Allocation table – таблица размещения файлов), SFT (System File Table – системная таблица файлов).

Каталоги. Логически каталог – это таблица, содержащая записи о файлах. В любой файловой системе запись о файле обязательно содержит:
имя и расширение имени файла;
размер (длину) файла;
адрес начала файла на ВЗУ;
атрибуты файла.

Для организации иерархии каталогов достаточно разрешить хранить каталоги в файлах со специальным атрибутом. В этом случае появляется понятие корневого каталога, располагающегося в фиксированном месте ВЗУ, и подкаталогов, размещенных в файлах.

Размещение файлов на ВЗУ. С точки зрения ОС большинство ВЗУ представляются как линейная последовательность блоков (в MS DOS – кластеров). Блок – минимальный объем внешней памяти, выделяемый под файл.

Для учета размещения файлов по кластерам используется FAT, каждый элемент которой описывает состояние соответствующего кластера. Взаимосвязь каталогов, FAT и размещения файлов по кластерам иллюстрируется следующей схемой (рис. 7.1).

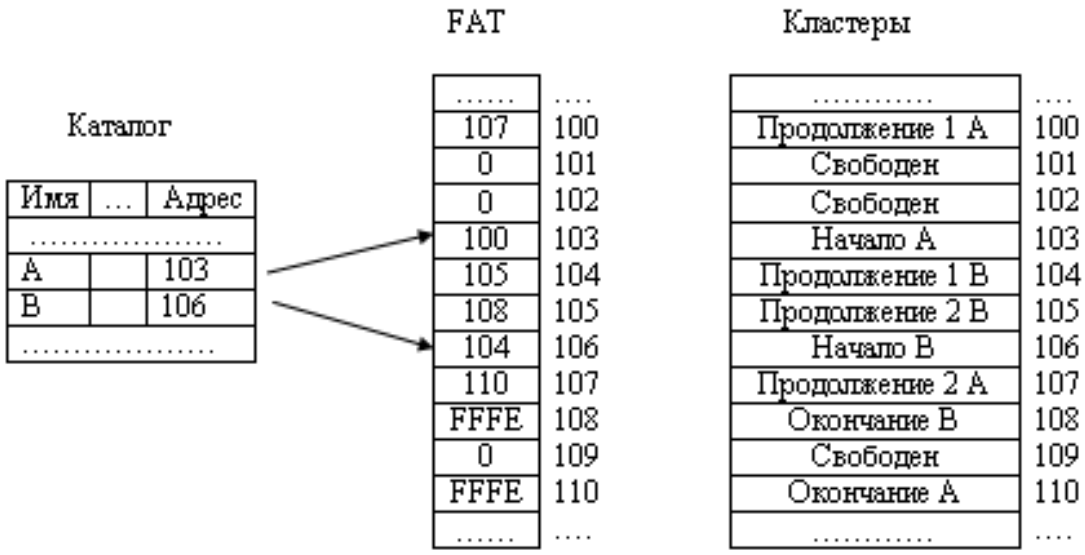


Рис. 7.1. Взаимосвязь каталогов, FAT и размещения файлов

Для ускорения доступа к файлам при первом обращении к устройству FAT считывается в оперативную память и хранится там в течение всего времени работы с устройством.

Системная таблица открытых файлов. SFT (таблица открытых файлов) содержит записи о файлах, с которыми работают прикладные программы. Каждая запись имеет уникальный числовой идентификатор (порядковый номер), присваиваемый операционной системой (handle файла). В состав записи входят данные, достаточные для выполнения всех операций с файлом без обращения к каталогам и FAT:

handle файла;
счетчик записей о файле;
все данные из записи о файле в каталоге;
режим доступа;
указатель текущей позиции;
указатель на копию FAT в оперативной памяти;
указатель на текущий каталог;
указатель на драйвер устройства и т. д.

2. Средства взаимодействия программ с ОС.

Прикладные программы обычно напрямую не работают с файлами, а обращаются с запросами к операционной системе, используя handle файла. При этом следует помнить, что в SFT постоянно присутствуют записи с handle 0, 1, 2, 3, 4:

0 – стандартный ввод (чаще всего клавиатура), только для чтения;
1 – стандартный вывод (чаще всего монитор), только для записи;
2 – стандартный вывод ошибок (чаще всего монитор), только для записи;
3 – устройство с логическим именем AUX (чаще всего COM1);
4 – устройство с логическим именем PRN (чаще всего LPT1),
что позволяет работать с перечисленными устройствами как с файлами.

Каждая ОС может выполнять определенный набор запросов (в DOS – функций). Набор запросов является одним из системных соглашений и определяется разработчиками ОС. Другим системным соглашением является способ передачи запросов из программ в ОС возврата результатов выполнения этого запроса в программу. Для DOS характерно размещение данных, описывающих запрос, в регистрах: в ah помещается номер функции, в остальные регистры – остальные данные, необходимые для выполнения этой функции. Передача управления в ОС производится с помощью программного прерывания (командой int 21h). Если в процессе выполнения запроса операционной системой возникает ошибка (система не в состоянии выполнить запрос), управление в программу возвращается с установленным флагом переноса CF. Результаты выполнения запросов, как правило, возвращаются в программу через регистр ax: если CF = 1, в нем лежит код ошибки, иначе – данные, соответствующие запросу. Таким образом, при взаимодействии программ с DOS, типичной является последовательность команд:

```
        Mov   ah, номер функции
        Mov   …….. размещаем все остальные данные
        Mov   …….. в других регистрах
        ……
        Int   21h
        Jnc   m
        …….. обработка ошибки
m:      …….. продолжение работы программы
```

Типовой набор запросов к ОС для работы с файлами. Любая операционная система поддерживает следующий набор запросов на доступ к файлам:

Создание.
Удаление.
Открытие.
Закрытие.
Чтение.
Запись.
Изменение характеристик файла.

Для обеспечения доступа к произвольному месту файла в большинстве ОС присутствует возможность позиционирования указателя файла.

Соответствующие функции DOS:
3ch Create, 39h Mkdir.
41h Delete, 3ah Rmdir.
3dh Open.

3eh Close.
3fh Read.
40h Write.
56h Rename, 57h Time/Date, 43h Chmod.
Для позиционирования предназначена функция 42h Lseek.
Смысл перечисленных операций:
Открытие файла – размещение записи о файле в SFT.
Закрытие файла – удаление записи о файле из SFT.
Создание файла – размещение записи о файле в каталоге и выделение ему места на устройстве. Создание файла в большинстве случаев сопровождается его открытием.
Удаление файла – удаление записи о файле в каталоге и освобождение выделенного ему места на устройстве.
Чтение файла – копирование содержимого файла в оперативную память.
Запись в файл – копирование содержимого оперативной памяти в файл.

3. Пример программы.

Работа с файлами иллюстрируется программой – копировщиком файлов:

```
    title   copy
    assume  cs:c, ds:d, ss:s

s   segment stack
    dw  128 dup ('ss')
s   ends

d   segment
mes1db  10,13,'Имя входного файла:$'
mes2db  10,13,'Имя выходного файла:$'
fname   db  255,0, 255 dup (?)
inhan   dw  ?
outhan  dw  ?
er1 db  10,13,'Файл не открылся$'
er2 db  10,13,'Файл не создан$'
buf db  256 dup (?)
er3 db  10,13,'Ошибка чтения файла$'
er4 db  10,13,'Ошибка записи файла$'
d   ends

c   segment
start:  mov ax,d
    mov ds,ax

m2: lea dx,mes1
```

```
     mov ah,9
     int 21h

     mov ah,0ah
     lea dx,fname
     int 21h

     lea di,fname+2
     mov al,-1[di]
     xor ah,ah
     add di,ax
     mov [di],ah

     mov ah,3dh
     lea dx,fname+2
     xor al,al
     int 21h
     jnc m1

     lea dx,er1
     mov ah,9
     int 21h
     jmp m2

m1:  mov inhan,ax

m3:  lea dx,mes2
     mov ah,9
     int 21h

     mov ah,0ah
     lea dx,fname
     int 21h

     lea di,fname+2
     mov al,-1[di]
     xor ah,ah
     add di,ax
     mov [di],ah

     mov ah,3ch
     lea dx,fname+2
     xor cx,cx
     int 21h
     jnc m4
```

```
     lea dx,er2
     mov ah,9
     int 21h
     jmp m3

m4:  mov outhan,ax

m7:  mov bx,inhan
     mov ah,3fh
     lea dx,buf
     mov cx,256
     int 21h
     jnc m5

     lea dx,er3
     mov ah,9
     int 21h

m6:  mov ah,3eh
     mov bx,inhan
     int 21h

     mov ah,3eh
     mov bx,outhan
     int 21h

     mov ah,4ch
     int 21h

m5:  cmp ax,0
     jz  m6

     mov ah,40h
     mov bx,outhan
     lea dx,buf
     int 21h
     jnc m7

     lea dx,er4
     mov ah,9
     int 21h
     jmp m6

c    ends
```

```
end start
```

В самом начале оперативной памяти располагаются 256 двухсловных векторов прерываний, содержащих адреса процедур обработки прерываний (обработчиков прерываний). За ними следует нижняя память (Lower memory), используемая для хранения программ и данных. Нижние 640 Кбайт ОЗУ образуют так называемую Conventional memory (основную или базовую память), за которой следует видеопамять, физически расположенная в видеоадаптере. В ней хранится образ экрана монитора. Выше видеопамяти находится верхняя память (Upper memory), состоящая из UMB (Upper Memory Block), расширенной памяти (Extended Memory Specification) и дополнительной памяти (eXpanded Memory Specification). Под самыми верхними адресами располагается ПЗУ (ROM BIOS), хранящее программы, исполняемые сразу после включения питания компьютера (точнее – после начального сброса аппаратуры). Этими программами являются:

POST (Power On Self Test) – самотестирование основных узлов компьютера после включения питания;

Setup (конфигурирование аппаратуры), предназначен для настройки режимов работы различных узлов компьютера;

Абсолютный загрузчик, отыскивающий на дисках операционную систему и инициирующий ее загрузку.

Помимо этого в ПЗУ располагается BIOS (Basic Input Output system) – базовая система ввода-вывода, представляющая собой набор стандартных процедур для работы с внешними устройствами. Между верхней памятью и ПЗУ может находиться неиспользуемая область адресов, в которой отдельные участки принадлежат расширениям BIOS – ПЗУ, физически расположенные во внешних устройствах и содержащие идентификационные данные этих устройств и иногда – процедуры для работы с ними.

ЛЕКЦИЯ 8
РАБОТА С ПАМЯТЬЮ В АССЕМБЛЕРЕ X86

План лекции

1. Распределение памяти, системные структуры данных, набор запросов к ОС.

2. Пример программы.

1. Распределение памяти, системные структуры данных, набор запросов к ОС.

В реальном режиме ЭВМ x86 использует следующее распределение памяти (рис. 8.1):

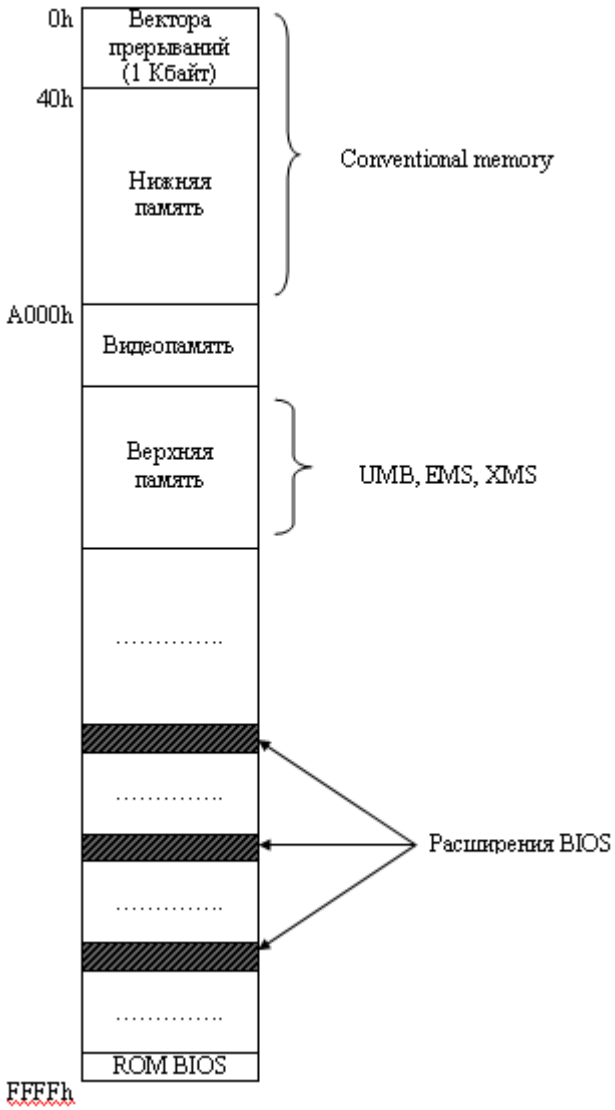


Рис. 8.1. Распределение памяти DOS

Сразу после загрузки DOS распределение Conventional Memory выглядит следующим образом (рис. 8.2):

Рис. 8.2. Распределение Conventional Memory

MCB (Memory Control Block – блок управления памятью) – структура данных длиной в 1 параграф, используемая системой для отслеживания состояния памяти. MCB описывает блок памяти, непосредственно следующий за ним. В нем документировано три поля:

Адрес PSP владельца блока памяти.

Размер блока в параграфах.

Признак последнего блока (M/Z).

На рис.8.2 присутствует пять блоков памяти, из которых 1, 3 и 4 заняты программами, 2 – хранит переменные окружения, 5 – считается свободным. Последний блок памяти, непосредственно предшествующий видеопамяти, используется для загрузки и выполнения прикладных программ.

Блоки памяти, занимаемые программами, имеют следующую структуру.

MCB	PSP	Сегмент стека	Сегмент данных	Сегмент кодов команд

PSP – служебная структура данных длиной 256 байт содержит описание программы.

Загружаемая прикладная программа помещается в свободный блок памяти, и этот блок целиком принадлежит ей. Поэтому, в общем случае, в сис-

теме может не остаться свободной памяти. Программы, которые используют динамически выделяемую память, должны освободить неиспользуемую часть выделенного им блока. Для работы с памятью DOS предоставляет 3 функции:

48h Allocate – выделить блок памяти.

49h Free – освободить выделенный блок памяти.

4ah Setblock – изменить размер выделенного блока памяти.

При запуске программы сегментные регистры ds и es содержат номер параграфа начала PSP.

2. Пример программы.

Работа с памятью иллюстрируется программой, которая в цикле запрашивает новый блок памяти и перемещает себя в него (за исключением сегмента стека).

```
     title   tstmov
     assume  cs:c,ds:d,ss:s

s    segment stack 'STACK'
     dw  128 dup('ss')
s    ends

d    segment
eoln db  10,13,'$'
er2  db  10,13,'Нет памяти$'
er1  db  10,13,'Ошибка 4ah$'
msg  db  10,13,'Мой текущий адрес:$'
msg1 db  10,13,'Запуск номер $'
cnt  dw  0
d    ends

c    segment

UnsignedOut  proc
             .....
UnsignedOut  endp

start:  mov bx,z
     mov ax,es
     sub bx,ax
     mov ah,4ah
     int 21h
     jnc mm1
```

```
     mov cx,d
     mov ds,cx
     mov ah,9
     lea dx,er1
     int 21h

     mov ah,4ch
     int 21h

mm1: mov ax,cs
     sub ax,c
     add ax,d
     mov ds,ax
     inc cnt

     mov ah,9
     lea dx,msg
     int 21h

     mov ax,cs
     callUnsignedOut

     mov ah,9
     lea dx,msg1
     int 21h

     mov ax,cnt
     callUnsignedOut

     mov ah,9
     lea dx,eoln
     int 21h

     mov bx,z
     mov ax,d
     sub bx,ax
     mov ah,48h
     int 21h
     jnc mm2

     mov ah,9
     lea dx,er2
     int 21h
```

```
    mov ah,4ch
    int 21h

mm2:mov es,ax
    mov cl,3
    shl bx,cl
    mov cx,bx
    xor si,si
m3: mov ax,[si]
    mov es:[si],ax
    inc si
    inc si
    loop m3

    mov ax,es
    add ax,c
    sub ax,d
    mov si,offset mm1
    pushf
    push ax
    push si
    iret
c   ends

z   segment
z   ends

    end start
```

ЛЕКЦИЯ 9
СИСТЕМА ПРЕРЫВАНИЙ X86

План лекции

1. Понятие прерывания. Классификация прерываний.
2. Аппаратная поддержка системы прерываний.

1. Понятие прерывания. Классификация прерываний.

Прерывание – это реакция вычислительной системы на некоторое асинхронное событие, которая заключается в том, что выполнение текущей программы временно прекращается, выполняется некоторая подпрограмма (обработчик прерывания), после чего чаще всего продолжается выполнение прерванной программы. Классификация прерываний (рис. 9.1).

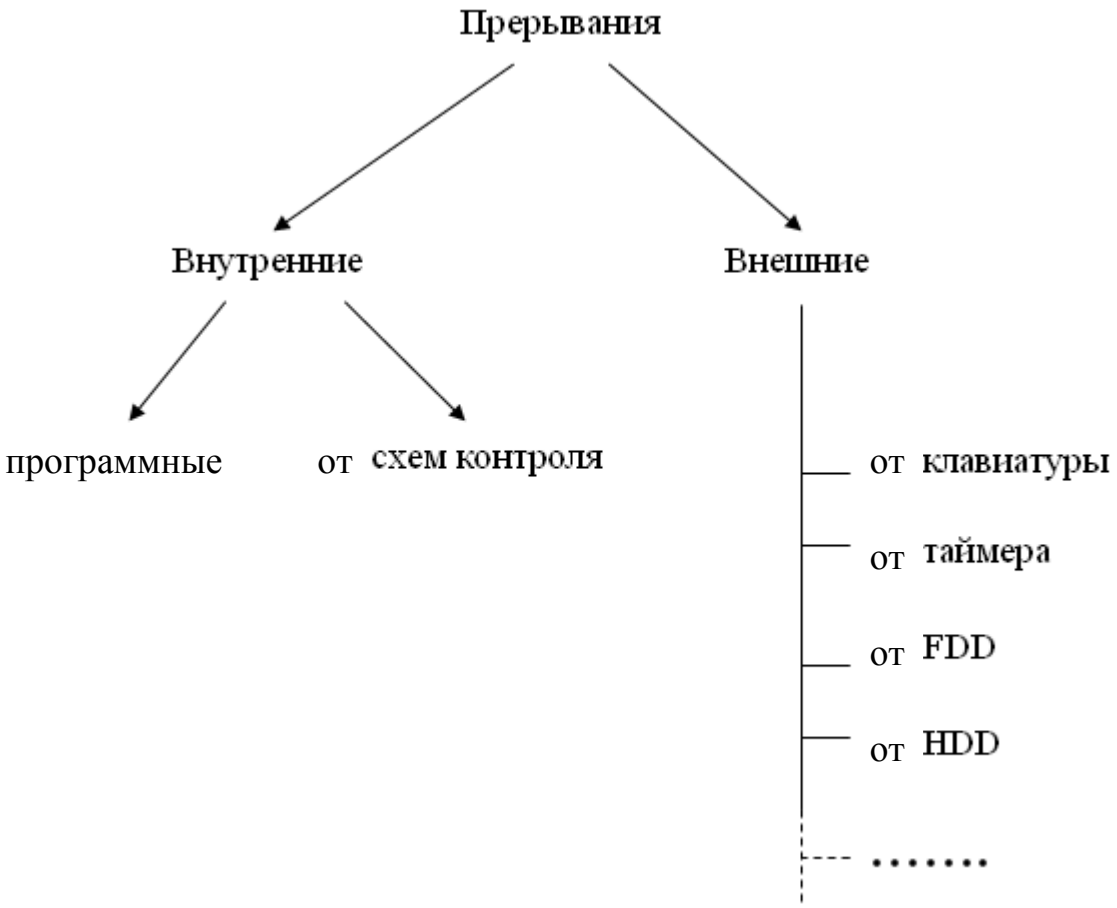


Рис. 9.1. Классификация прерываний

Программные прерывания вызываются командой int и используются для передачи управления из прикладных программ в ОС или BIOS.

Прерывания от схем контроля возникают в тех случаях, когда невозможна корректная работа процессора. Примерами таких прерываний являются:

прерывание по резервной инструкции;
прерывание по нарушению защиты памяти;
прерывание по нарушению вида доступа к памяти и т. д.

Поскольку эти прерывания в отличие от остальных прерывают выполнение команды, в последнее время их стали называть исключениями.

Внешние прерывания возникают по требованию внешних устройств в тех случаях, когда им требуется обслуживание. Исключением из этого правила является системный таймер – устройство, запрограммированное на периодическую выработку запросов на прерывание через фиксированные интервалы времени (в DOS – приблизительно18,2 ms). Он используется для отслеживания системного времени, интервалов времени при работе с внешними устройствами, тайм-аутов, исключающих «зависания» системы.

Фактически вся работа операционной системы основана на прерываниях.

2. Аппаратная поддержка системы прерываний.

Для обеспечения возможности работы с прерываниями аппаратная часть компьютера обязательно содержит соответствующие средства поддержки прерываний. В реальном режиме работы линии x86 такими средствами являются:

опрос процессором входной линии запроса прерывания перед исполнением очередной команды;
вектора прерываний;
команды процессора int, into, iret;
флаг разрешения прерывания (IF) в регистре флагов процессора;
контроллер прерываний;
схемы выработки сигналов прерываний во внешних устройствах.

До настоящего времени нами рассматривался упрощенный вариант цикла работы процессора (рис. 9.2).

Рис. 9.2. Упрощенный цикл работы процессора

На самом же деле он выглядит следующим образом (рис. 9.3).

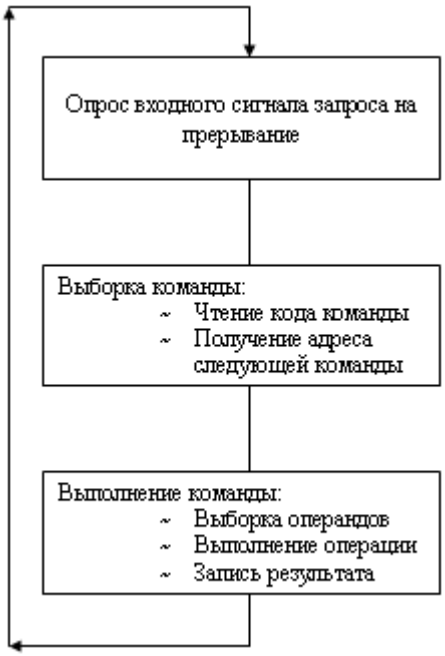


Рис. 9.3. Полный цикл работы процессора

Векторы прерываний в реальном режиме работы располагаются в самом начале оперативной памяти и занимают первые 2^{10} байт. Всего векторов 256, и они пронумерованы от 0 до 255. За каждым вектором закреплены свои прерывания. По своему содержанию вектор прерывания – это два слова, содержащие адрес обработчика прерывания. В старшем слове располагается номер параграфа (сегментная часть адреса), в младшем – смещение относительно сегментной части адреса.

Команда int вызывает прерывание с вектором, указанным в качестве аргумента команды, в частности int 21h (или int 33) – передает управление в DOS, int 13h и int 17h – в BIOS и т. д. Команда iret используется для возврата управления из обработчиков прерываний назад в прерванную программу.

Флаг IF в регистре флагов процессора определяет, будет ли процессор воспринимать запросы на прерывания от внешних устройств. Если IF равен 0, то процессор не воспринимает запросы внешних прерываний, если IF = 1, то запросы вызывают прерывания. Назначение этого флага иллюстрируется следующей логической схемой (рис. 9.4).

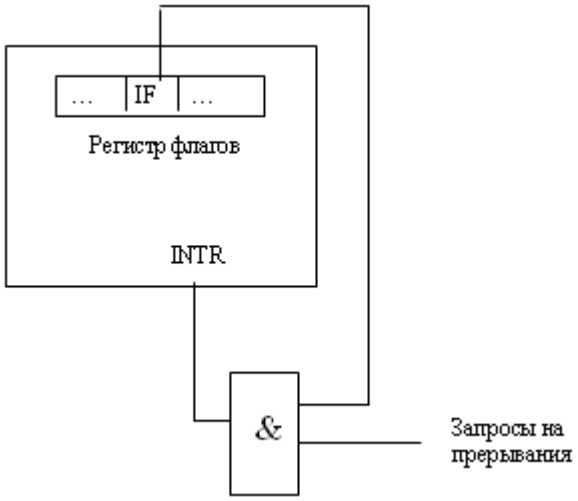


Рис. 9.4. Логика работы с флагом IF

INTR – входная линия процессора для запросов на прерывания (**int**er-rupt **r**equest).

ЛЕКЦИЯ 10
КОНТРОЛЛЕР ПРЕРЫВАНИЙ

План лекции

1. Понятие контроллера прерываний, схемы его построения.
2. Работа контроллера прерываний.

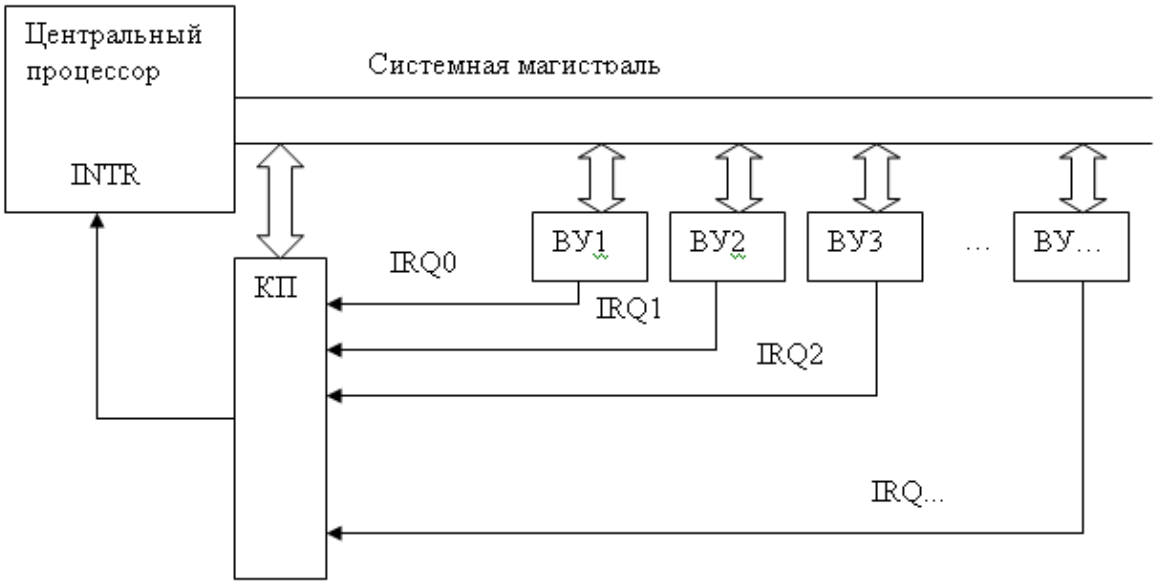
1. Понятие контроллера прерываний, схемы его построения.

Контроллер прерываний – узел компьютера, логически располагающийся между процессором и внешними устройствами (рис. 10.1).

Сигналы запросов на прерывания от внешних устройств поступают по линиям IRQx в контроллер прерываний, а далее он отправляет запросы в центральный процессор.

Узнать, какое внешнее устройство использует тот или иной вход IRQ, можно из свойств этого устройства (рис. 10.2).

Исторически первым контроллером прерываний была микросхема i8259A, имевшая 8 входов IRQ (с IRQ0 по IRQ7). Однако уже тогда было явно видно, что восьми линий запросов прерываний недостаточно, поэтому указанная микросхема позволяла строить двухуровневую схему контроллера прерываний с использованием каскадного соединения. Простейшая схема такого контроллера состоит из двух микросхем (ведущего и ведомого контроллеров) и увеличивает количество входов IRQ до 15 (рис. 10.3).



КП – контроллер прерываний; ВУ – внешние устройства.

Рис. 10.1. Место контроллера прерываний

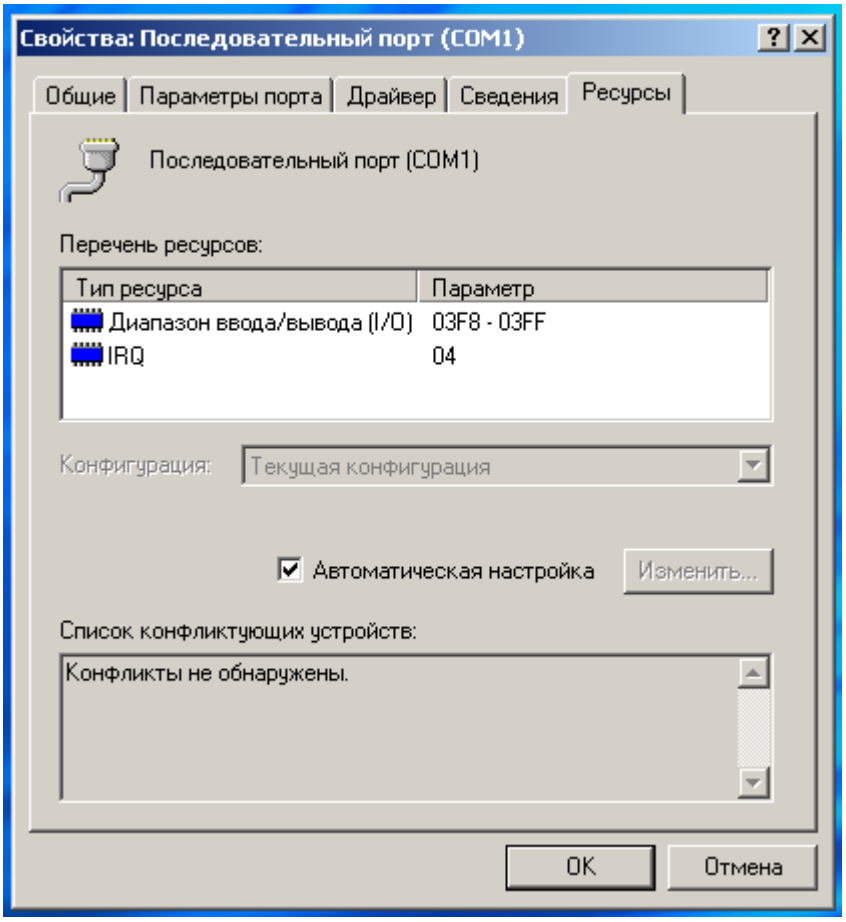


Рис. 10.2. Окно ресурсов внешнего устройства

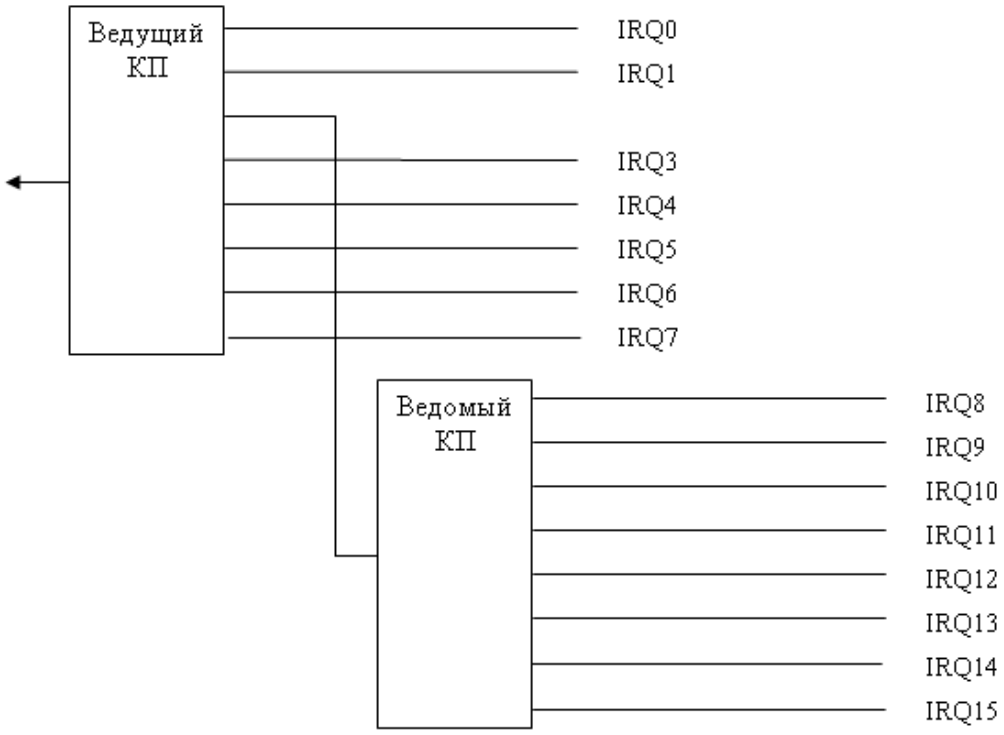


Рис. 10.3. Каскадное соединение двух микросхем

Предельное количество входов IRQ составляет 64 и получается при соединении девяти микросхем (рис. 10.4).

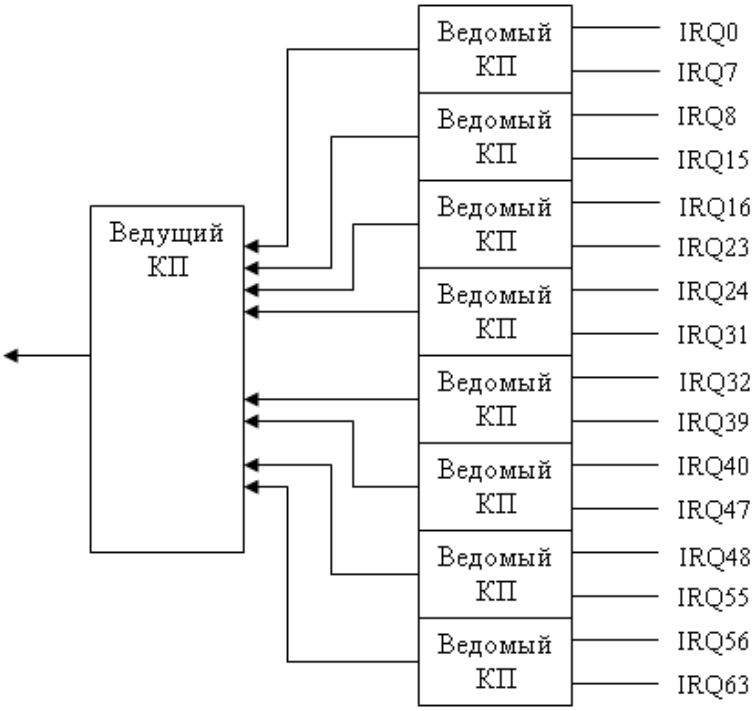


Рис. 10.4. Каскадное соединение девяти микросхем

Современные контроллеры прерываний, интегрированные в чипсет (в «южный мост»), сохраняют этот старый принцип каскадного построения.

2.Работа контроллера прерываний

Контроллер прерываний выполняет следующие функции:

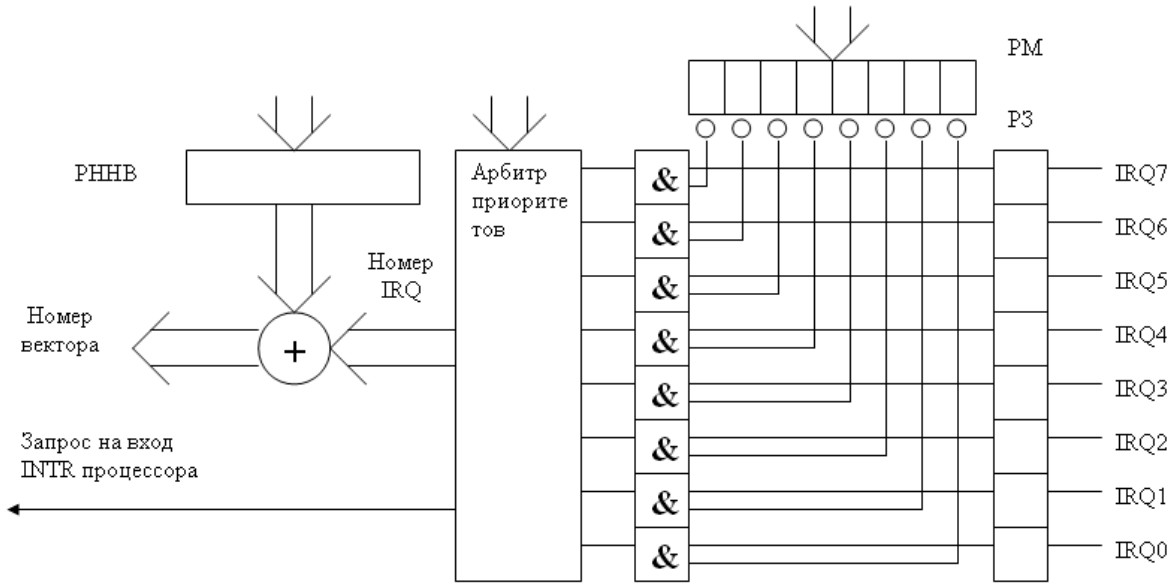
фиксирует запросы на прерывания от внешних устройств;

маскирует поступившие запросы (разрешает или запрещает прохождение запросов с отдельных IRQ входов);

реализует обслуживание запросов в соответствии с их приоритетами (в случае, когда одновременно присутствуют несколько запросов);

передает процессору номер вектора прерывания, который следует использовать при обработке прерывания.

Логика работы контроллера прерываний иллюстрируется следующей схемой (рис. 10.5).



РМ – регистр маски; РЗ – регистр-защелка; РННВ – регистр начального номера вектора.

Рис. 10.5. Логика работы контроллера прерываний

Запросы со входов IRQ запоминаются в регистре-защелке до тех пор, пока не произойдет соответствующее прерывание. Далее сигналы запросов логически перемножаются с проинвертированным содержимым регистра маски (единичное значение соответствующего разряда не пропускает запросы – маскирует их). Незамаскированные сигналы поступают в арбитр приоритетов, который, в случае наличия нескольких запросов, выбирает для обслуживания самый высокоприоритетный. Типовое распределение приоритетов между IRQ дает понижение приоритетов при увеличении номера IRQ. Номер IRQ, по которому пришел запрос, выбранный для обслуживания, складывается с начальным номером вектора прерывания из РННВ, и полученный номер вектора впоследствии передается процессору.

В контроллере прерываний программируются: маска (в РМ), арбитр приоритетов, начальный номер диапазона номеров векторов прерываний (в РННВ), режим работы, схема соединения «ведущий – ведомые».

Сам переход к обработчику прерывания процессором всегда выполняется одинаково и может быть проиллюстрирован следующей последовательностью операций.

```
Pushf
Push  cs
Push  ip
Mov   cs, 0:[n*4+2]
Mov   ip, 0:[n*4]
cli
```

Возврат из обработчика прерывания выполняется командой iret и сводится к трем пересылкам:

```
Pop   ip
Pop   cs
Popf
```

ЛЕКЦИЯ 11
ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ СИСТЕМЫ ПРЕРЫВАНИЙ

План лекции

1. Состав и размещение обработчиков прерываний.
2. Общие принципы функционирования обработчиков прерываний и требования к ним.
3. Пример программы.

1. Состав и размещение обработчиков прерываний.

Программное обеспечение системы прерываний образует совокупность всех обработчиков прерываний. Обработчиком программных прерываний является операционная система, точнее та ее часть, которая занимается выполнением запросов прикладных программ.

Прерывания от схем контроля также обрабатываются процедурами операционной системы. Типичным их действием является принудительное аварийное завершение программы, во время исполнения которой возникло прерывание, сопровождающееся выводом сообщения приблизительно такого содержания: «Программа выполнила недопустимую операцию и будет закрыта».

Прерывания от внешних устройств обрабатываются процедурами, физически расположенными в драйверах внешних устройств. Очень упрощенный алгоритм функционирования драйвера устройства ввода-вывода, работающего по прерываниям, выглядит следующим образом (рис. 11.1).

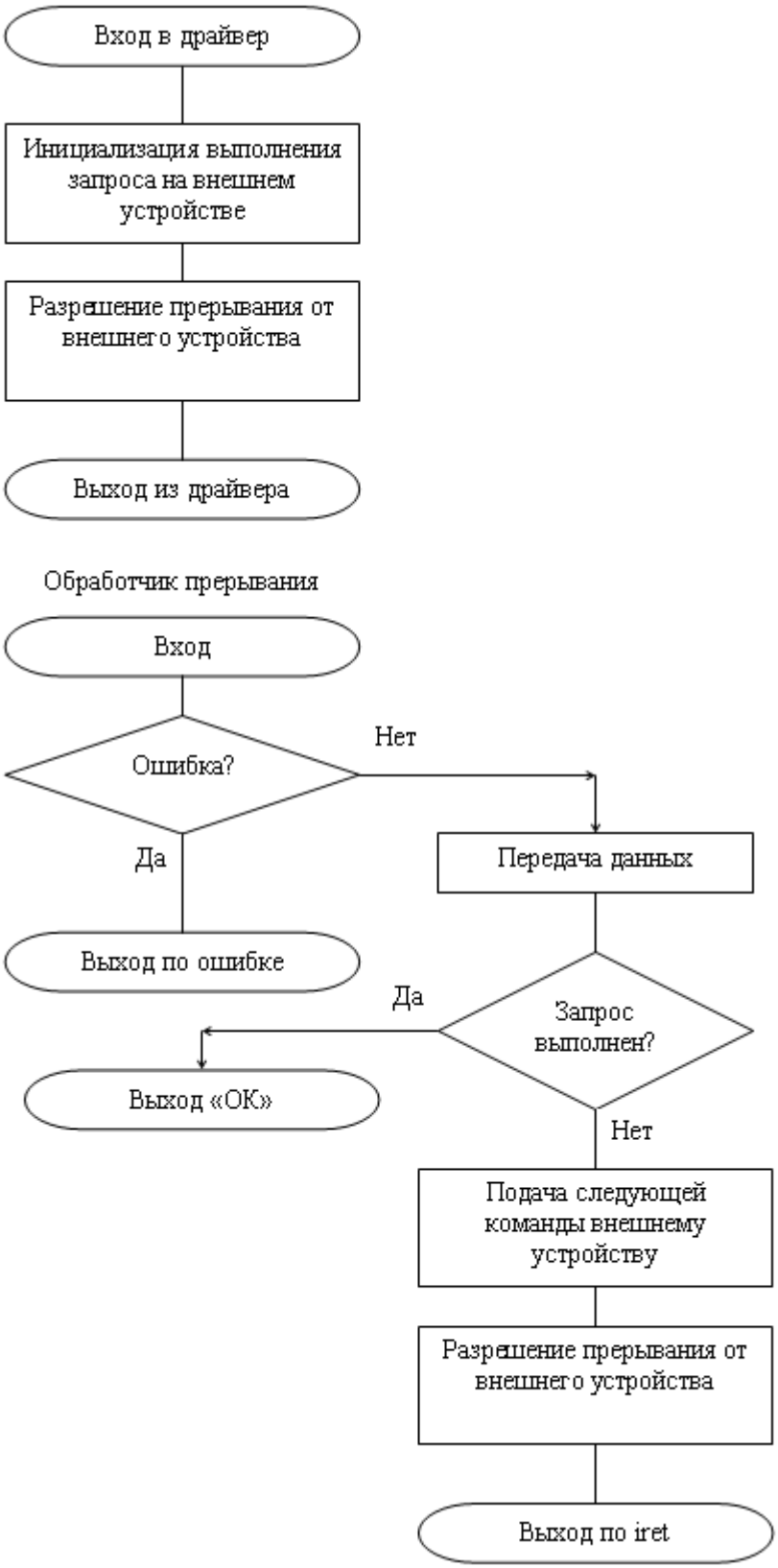


Рис. 11.1. Упрощенный алгоритм работы драйвера

Вход в драйвер. Когда прикладная программа передает запрос на ввод-вывод операционной системе, та, в свою очередь, формирует запрос к соот-

ветствующему драйверу: заполняет структуру данных, описывающую запрос, и передает управление в драйвер.

Инициализация выполнения запроса на внешнем устройстве. Получив управление, драйвер анализирует запрос (полученную структуру данных) и подает команду внешнему устройству (помещает соответствующие коды в регистры этого устройства). С этого момента внешнее устройство начинает выполнять операцию в соответствии с полученной командой.

Разрешение прерывания от внешнего устройства в общем случае требует разрешения прерывания на внешнем устройстве (записи соответствующего кода в регистр управления) и разрешения прохождения запросов с соответствующего IRQ входа (записи 0 в необходимый разряд регистра маски контроллера прерываний).

Выход из драйвера. После разрешения прерывания драйвер передает управление в операционную систему, чтобы та могла загрузить процессор полезной работой (например, выполнением одной из прикладных программ), пока внешнее устройство выполняет запущенную операцию.

Далее, когда внешнее устройство заканчивает выполнение операции, выполняющаяся программа прерывается и управление попадает в обработчик прерываний драйвера. Он проверяет успешность выполнения операции внешним устройством (состояние соответствующих разрядов регистра управления ВУ) и, если операция завершилась неуспешно, передает управление в операционную систему с признаком ошибки. В случае успешного завершения операции выполняется передача данных между внешним устройством и областью памяти прикладной программы, выдавшей запрос.

После этого драйвер проверяет объем переданных данных и, если он не совпадает с запрошенным объемом, подает следующую команду внешнему устройству, разрешает прерывания и возвращает управление назад в прерванную программу.

Когда очередная команда будет выполнена, опять произойдет прерывание и описанный процесс повторится.

Однажды обработчик прерываний обнаружит, что запрос выполнен (переданы все данные, предписанные запросом), после чего он запретит прерывания от внешнего устройства и передаст управление в операционную систему с признаком успешного завершения выполнения запроса.

2.Общие принципы функционирования обработчиков прерываний и требования к ним.

Большинство обработчиков прерываний работают по следующей общей схеме (рис. 11.2).

Рис. 11.2. Обобщенный алгоритм обработчика прерываний

Поиск источника прерывания наиболее явно выражен в тех случаях, когда к одному IRQ входу контроллера прерываний подключено несколько внешних устройств. При возникновении исключений (прерываний от схем контроля) требуется определить программу, выполнение которой привело к возникновению исключения. Определение причины прерывания:

для внешних устройств характерны две причины возникновения прерывания – успешное и неуспешное завершение операции;

для программных прерываний требуется определить, какой запрос необходимо выполнить.

Конкретные действия по обслуживанию прерывания зависят от результатов предыдущих операций.

При разработке процедур обработки прерываний в общем случае следует учитывать ряд требований и правил. Обработчик прерываний должен:

сохранять контекст прерванной программы;

держать запрещенными прерывания только в случае крайней на то необходимости;

выполняться за минимально возможное время;

предусматривать возможность повторного входа. Для этого быть либо реентерабельным, либо защищенным от повторного входа;

обеспечивать срабатывание обработчика, использовавшего данный вектор ранее.

Запросами к операционной системе можно пользоваться только в том случае, если заведомо известно, что данный запрос разрешен к использованию в обработчиках прерываний.

3. Пример программы.

С учетом вышеприведенного для прикладных программ, использующих собственные обработчики прерываний можно предложить следующую схему взаимодействия (рис. 11.3).

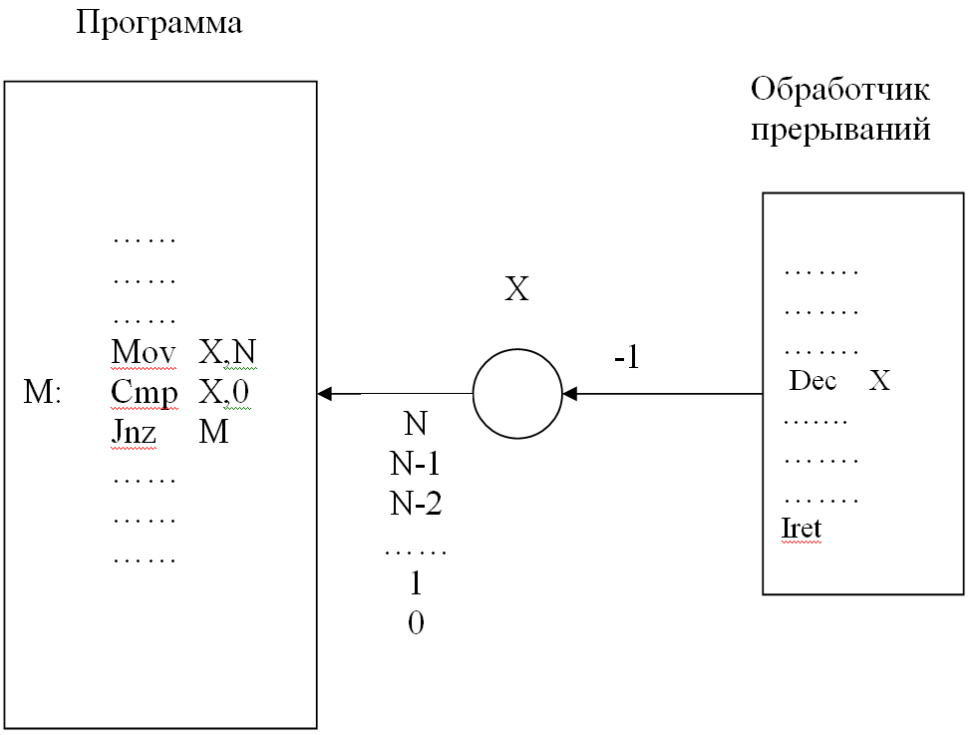


Рис. 11.3. Схема взаимодействия программы с обработчиком прерываний

Приведенная схема иллюстрирует организацию цикла ожидания N прерываний (N срабатываний обработчика).

Примером такой схемы взаимодействия является программа, выводящая звездочки на экран монитора с заданной скоростью. Данная программа использует прерывания с вектором номер 1Ch – пользовательское прерывание от таймера.

```
        assume cs:c,ss:s,ds:d
        delay = 3

s       segment     stack
        dw    128 dup(?)
s       ends

d       segment
old1c   dw    0,0
x       dw    ?
d       ends

c       segment
start:  mov   ax,d
```

```
        mov   ds,ax

        mov   ax,351ch
        int   21h
        push  bx
        push  es

        mov   ax,251ch
        lea   dx,tim
        push  ds
        push  cs
        pop   ds
        int   21h
        pop   ds

        mov   cx,10

        mov   x,18*delay
m:      cmp   x,0
        jnz   m
        mov   x,18*delay

      mov     ah,2
        mov   dl,'*'
        int   21h
        loop  m

        pop   ds
        pop   dx
        mov   ax,251ch
        int   21h

        mov   ax,4c00h
        int   21h

tim:    push  ax
        push  ds

        mov   ax,d
        mov   ds,ax

        dec   x

        pop   ds
        pop   ax
        iret
c       ends
        end   start
```

ЛЕКЦИЯ 12
ВЗАИМОДЕЙСТВИЕ ПРОГРАММ С ОПЕРАЦИОННЫМИ СИСТЕМАМИ

План лекции

1. Принципы взаимодействия ассемблерных программ с ОС.
2. Общие вопросы взаимодействия программ с операционной системой.

1. Принципы взаимодействия ассемблерных программ с ОС.

Большинство прикладных программ в своей работе не обходятся без услуг операционных систем. В этом мы убедились в предыдущих лекциях: ОС предоставляет набор услуг по выполнению различных операций, таких как ввод-вывод, работа с файлами и каталогами, работа с памятью и т. д. Чем обширнее набор таких сервисов и чем они крупнее, тем проще разрабатывать программы. Обращение из прикладной программы к операционной системе для выполнения какой-либо операции называется запросом к ОС или системным вызовом. После выполнения запроса операционная система возвращает в прикладную программу сведения об успешности выполнения запроса и, если это предусмотрено, результаты его выполнения.

Общие принципы взаимодействия. Способ передачи запросов в ОС и получения от нее результатов (интерфейс системных вызовов) зависит от конкретной операционной системы и является частью системных соглашений (стандартные соглашения о связях). Различные ОС используют разные интерфейсы системных вызовов, но в общем случае последовательность действий программы состоит из следующих операций (рис. 12.1):

подготовка данных, описывающих запрос;

передача управления в операционную систему;

после возврата управления из операционной системы – анализ успешности выполнения запроса и его результатов.

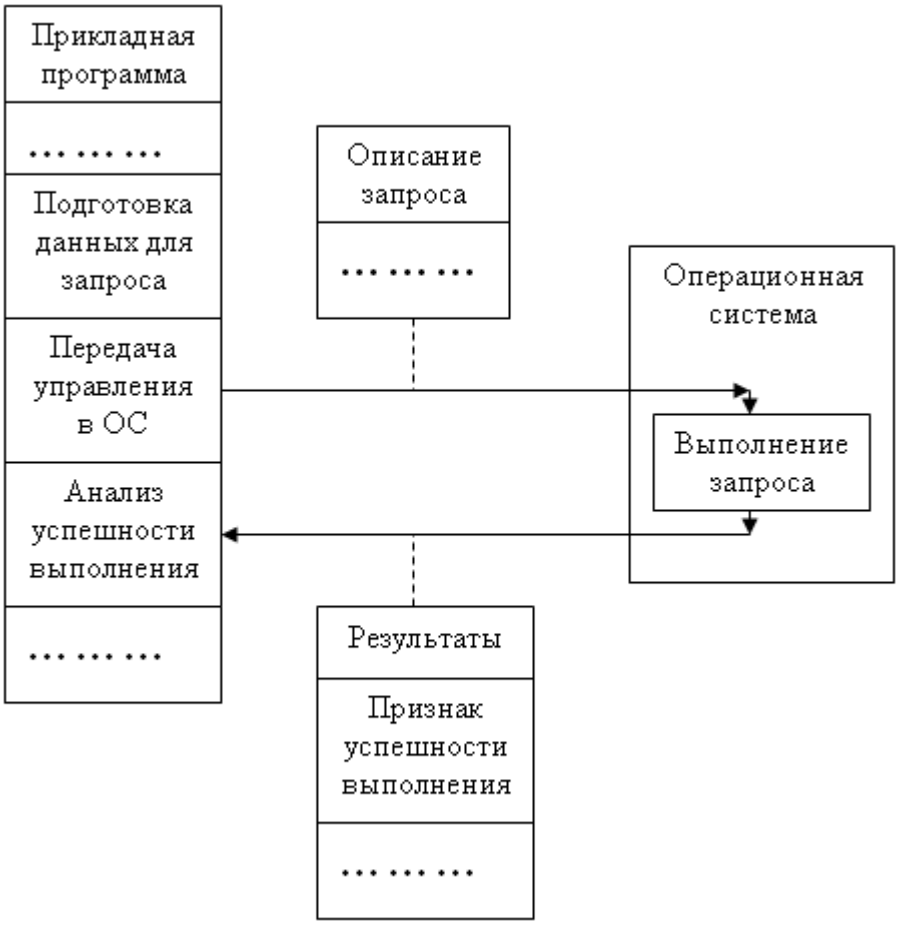


Рис. 12.1. Схема взаимодействия прикладной программы с ОС

На сегодняшний день наиболее распространенными являются две разновидности системных вызовов, рассматриваемых далее.

Программные прерывания. *Первый вариант интерфейса системных вызовов* прекрасно виден при рассмотрении вызовов функций DOS (который, кстати, она унаследовала из других операционных систем, существовавших на момент ее появления). Кратко перечислим его основные особенности:

данные, составляющие описание запроса, размещаются в регистрах процессора. Для передачи операционной системе данных большого объема используются указатели, которые размещаются опять-таки в регистрах процессора;

передача управления в операционную систему осуществляется с помощью программных прерываний с определенным вектором, команды которых присутствуют в системах команд подавляющего большинства процессоров.

В DOS для этих целей используется команда int с вектором номер 21h.

После возврата управления из операционной системы с помощью команды возврата из прерывания (или аналогичной ей последовательности команд) признаком неудачного завершения выполнения системного вызова является единичное состояние флага переноса в регистре состояния процессора (в DOS – флаг CF регистра флагов процессора). Результаты выполнения за-

проса возвращаются в прикладную программу через регистры процессора (для DOS – чаще всего регистры Ax, Bx, ES; в случае установленного флага переноса в регистре Ax размещается код ошибки).

Данный вариант системных вызовов появился в системах с защитой памяти, в которых прикладные программы исполняются в пользовательском режиме и им запрещено обращаться в не принадлежащие им области памяти. При передаче запроса из программы в ОС требуется передать управление операционной системе, и, следовательно, процессор должен начать выбирать команды из «чужой» области памяти – памяти, принадлежащей ОС. Но аппаратура защиты памяти не разрешает этого. Таким образом, возникает конфликт, хотя, по-существу, это обычная ситуация. Для устранения этого конфликта необходимо одновременно с передачей управления менять состояние процессора с пользовательского на системное, в котором снимаются все ограничения. Для этих целей наилучшим решением является прерывание, которое и передает управление, и меняет состояние процессора на системное.

API-функции. *Второй вариант организации системных вызовов* на сегодняшний день стал стандартом. Это так называемые API-функции (Application Programming Interface – интерфейс программирования приложений). Наиболее известной в этой сфере является группа стандартов, имеющих общее наименование POSIX (Portable Operation System Interface – переносимый интерфейс операционной системы). Впервые этот интерфейс полностью был реализован в ОС Linux.

Данный вариант передачи запросов к операционной системе предполагает оформление системных вызовов в виде функций. Описанием запроса в этом случае выступает имя (идентификатор) функции и передаваемые в нее аргументы. Результаты выполнения запроса возвращаются в прикладную программу в виде результата функции и в случае необходимости через переданные в функцию аргументы (в тех случаях, когда результаты запроса представляют собой некоторую структуру данных достаточно большого объема). Фактически API-функции используют стандартные средства языка программирования для вызова процедур и функций, в которых передача аргументов в вызываемые подпрограммы производится через стек, а результат функции возвращается через регистры процессора (чаще всего – регистры Al, Ax, EAx, Dx, EDx).

Если совместить стандартные механизмы работы с процедурами и функциями, используемые в современных языках программирования, с вышеприведенной последовательностью действий прикладной программы для использования услуг операционной системы, то получается следующий интерфейс системных вызовов:

данные, составляющие описание запроса, размещаются в стеке. Состав этих данных определяется составом аргументов API-функции;

передача управления в операционную систему осуществляется с помощью команды вызова подпрограммы Call, которая присутствуют в системах команд подавляющего большинства процессоров;

возврат управления из операционной системы производится с помощью команды возврата из подпрограммы (или аналогичной ей последовательности команд). Результаты выполнения запроса возвращаются в прикладную программу через регистры процессора и в случае необходимости через переданные в функцию аргументы, содержащие указатели на переменные с результирующими данными. Признаком неудачного завершения выполнения системного вызова обычно является отрицательное или нулевое значение регистра EAX.

Программы, написанные на разных языках программирования, даже под управлением одной операционной системы, используют различные реализации механизмов работы с подпрограммами, и вследствие этого без специальных мер нельзя из процедуры, написанной на одном языке, вызвать процедуру, написанную на другом языке. Операционная система тоже имеет свой интерфейс вызова, не совместимый со стандартными механизмами вызова подпрограмм, используемыми по умолчанию в различных языках программирования. В общем случае различия заключаются в виде вызова, порядке размещения аргументов в стеке и способе удаления их из стека. Для решения этого вопроса в состав языков программирования включаются средства согласования интерфейсов вызова разноязыковых подпрограмм. Рассмотрение этих средств является одним из вопросов модульного программирования, которое является следующей темой настоящего курса лекций.

2. Общие вопросы взаимодействия программ с операционной системой.

На сегодняшний день наиболее распространенными являются три разновидности приложений: DOS-приложения и консольные и оконные Windows-приложения.

Dos-приложения достаточно хорошо рассмотрены в предыдущих лекциях и практически не накладывают никаких ограничений на алгоритмическое построение программ.

Консольные Windows-приложения в принципе построены так же, как и 16-разрядные приложения MS DOS. При их запуске внешне происходит то же самое, открывается «черное» окно, в котором видны текстовые сообщения из программы и в котором пользователь может набирать строки. Отличительной особенностью консольных Windows-приложений является в первую очередь то, что в них используется 32-разрядная архитектура: 32-разрядные регистры, 32-разрядные адреса. Следующим отличием является то, что консольные приложения используют «плоскую» (FLAT) модель памяти, в которой отсутствует понятие сегментов (точнее началом всех сегментов является один и тот же начальный адрес памяти). Третьим отличием являются запросы к операционной системе, которые выполняются не с помощью программных прерываний, как в DOS-функциях, а в виде вызова подпрограмм (API-функций).

Оконные приложения существенно отличаются от консольных. Эти отличия связаны с использованием средств операционной системы для работы с окнами. В оконных приложениях окно является самостоятельным объектом операционной системы и отделено от самой программы (рис. 12.2).

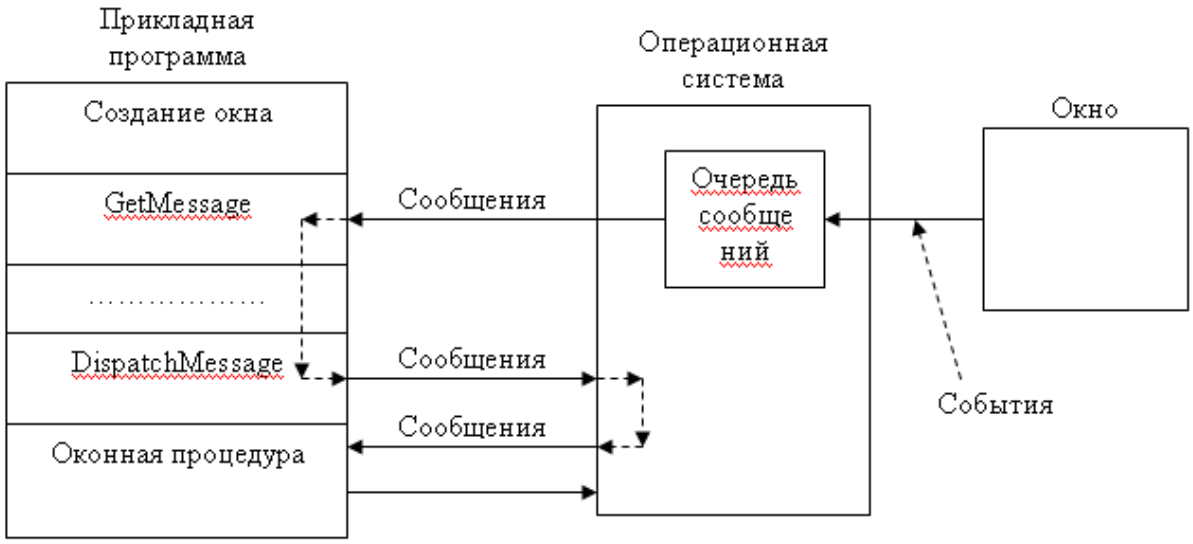


Рис. 12.2. Схема оконного приложения

Воздействия пользователя на окно являются событиями, которые обрабатываются операционной системой, и для каждого события формируется сообщение, описывающее это событие. Все сообщения выстраиваются в очередь сообщений, находящуюся в ведении операционной системы. Задачей прикладной программы является получение сообщений из системной очереди с помощью функции GetMessage и передача их через операционную систему оконной процедуре прикладной программы (функция DispatchMessage). Оконная процедура выполняет все действия, необходимые для обработки действий пользователя (событий). Для того чтобы приложение получило свое окно, оно должно само позаботиться о его создании (вызвать функции CreateWindow и ShowWindow).

ЛЕКЦИЯ 13
МОДУЛЬНОЕ ПРОГРАММИРОВАНИЕ

План лекции

1. Виды вызовов подпрограмм и способы их указания.
2. Стили вызовов подпрограмм.
3. Языковые средства ассемблера для поддержки модульного программирования.

При разработке больших программ часто используется принцип модульного программирования, который предполагает использование готовых модулей, часто написанных на разных языках программирования.

В такой ситуации возникает ряд вопросов, связанных с согласованием особенностей генерации кодов модулей при трансляции с различных языков программирования.

Первый вопрос связан с согласованием имен и атрибутов сегментов модулей. Этот вопрос решается довольно просто: на сегодняшний день большинство компиляторов с языков высокого уровня дают устоявшиеся имена сегментам. Чаще всего сегмент кода носит имя _TEXT, сегмент данных – _DATA, сегмент стека – STACK. Чтобы не задумываться над именами сегментов в ассемблерных программах рекомендуется пользоваться упрощенными директивами сегментации: .CODE, .DATA, .STACK. В этом случае транслятор сам даст типовые имена сегментам модуля и присвоит им все необходимые атрибуты.

Поскольку разные языки используют различные реализации вызова подпрограмм, важнейшим вопросом обеспечения корректного взаимодействия разноязыковых модулей является согласование интерфейсов вызовов подпрограмм. Первым вопросом здесь является согласование вида вызова подпрограммы возврата из нее.

Линия процессоров x86 имеет в своей системе команд две разновидности команд вызова подпрограмм и возврата из них.

Команды вызовов:

Call near (близкий или внутрисегментный вызов);

Call far (далекий или межсегментный вызов).

Команды возвратов:

ret near (близкий или внутрисегментный возврат);

ret far (далекий или межсегментный возврат).

Отличия близких от далеких вызовов и возвратов заключаются в размерности изменяемых адресов:

близкие вызовы и возвраты изменяют только смешения в сегменте и позволяют передавать управление в пределах сегмента. Фактически в этом случае изменяется только содержимое регистра IP/EIP;

далекие вызовы и возвраты изменяют не только смещения в сегменте, но и сегментную часть адреса и позволяют передавать управление между

сегментами. В этом случае изменяется не только содержимое регистра IP/EIP, но и содержимое сегментного регистра CS.

Содержимое стека, получаемое вызываемой процедурой, для near и far вызовов отличается (рис. 13.1).

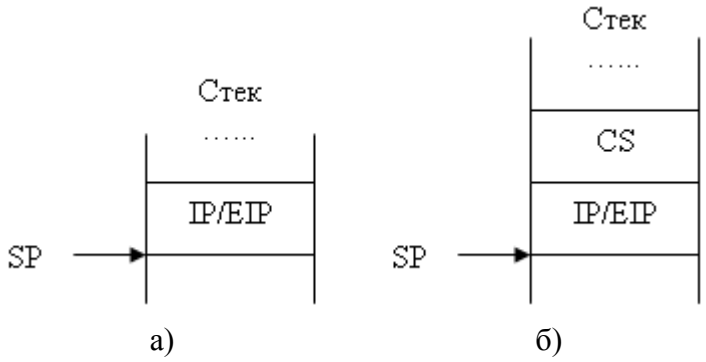


Рис. 13.1. Содержимое стека: а) для Call near, б) для Call far

Таким образом при согласовании разноязыковых модулей необходимо иметь возможность определять виды вызовов и возвратов из подпрограмм.

В ассемблерных программах для этого существуют следующие средства:

явное задание вида вызова и возврата в командах Call и Ret;

указание вида вызова в директиве Proc;

определение вида вызова с помощью модели памяти (директивы .Model).

Команды Call и Ret позволяют задавать виды вызовов следующими способами: команда Ret имеет несколько мнемоник – Ret, Retn, Retf. Первые две мнемоники заставляют транслятор генерировать коды команд близких возвратов, последняя мнемоника – код далекого возврата. На практике пользоваться этой возможностью довольно сложно, поэтому лучше задавать вид возврата с помощью вида процедуры – в этом случае транслятор сам определит вид возврата и сгенерирует соответствующий код.

Команда Call имеет одну мнемоническую запись, но в ней указывается операнд, по смыслу определяющий адрес подпрограммы, к которой производится переход. По этому операнду и определяется вид вызова, если адрес перехода описан в том же сегменте, что и сама команда, транслятор генерирует код близкого вызова, если же адрес перехода и команда вызова расположены в разных сегментах – генерируется код далекого вызова.

Другой вариант указания вида вызова заключается в явном описании типа адреса перехода с помощью директивы label:

```
        ……..
A       label   near
        ……..
B       label   far
        ……..
```

```
        Call   A
        ……..
        Call   B
```

В этом случае Call A будет близким вызовом, Call B – далеким.

В случае, когда переход осуществляется с использованием указателя на адрес подпрограммы, необходимо явное указание размерности адреса:

```
Call word ptr [Bx] – для близкого вызова,
Call dword ptr [Bx] – для далекого вызова.
```

По умолчанию используются близкий вызов и возврат.

Вышеописанные способы используются, когда подпрограмма представляет собой последовательность команд, начинающуюся с некоторой метки.

В ассемблере имеются директивы для оформления подпрограмм в виде процедур – директивы proc и endp:

```
A       proc   near
        …….
        ret
A       endp

B       proc   far
        ……..
        ret
B       endp
```

Директива proc позволяет явно задавать вид вызова и заставляет транслятор генерировать вместо команды ret соответствующий код возврата. В приведенном примере процедура A – близкая, B – далекая.

Данный способ указания вида вызова и возврата является более предпочтительным по сравнению с предыдущими, так как директива proc дает еще ряд возможностей, рассматриваемых далее.

Также имеется возможность задавать вид вызова путем указания модели памяти с помощью директивы model, одним из аргументов которой является указание модели памяти (табл. 13.1).

Таблица 13.1

Модель	Описание модели памяти
Tiny	Один сегмент кода и один сегмент данных, физически наложенные друг на друга (16-разрядная)
Small	Один сегмент кода и множество сегментов данных (16-разрядная). Используется по умолчанию в DOS - приложениях
Medium	Множество сегментов кода и множество сегментов данных (16-разрядная)
Large	Множество сегментов кода и множество сегментов данных (16-разрядная)
Flat	Один сегмент кода и один сегмент данных, физически наложенные друг на друга (32-разрядная). Используется по умолчанию в Windows-приложениях

Из этой таблицы видно, что различные модели памяти предполагают разное количество сегментов кода, а это, в свою очередь, приводит к неявному указанию вида вызовов подпрограмм: для моделей Tiny, Small и Flat характерен один сегмент кодов, и вызовы будут near; для моделей Medium и Large, имеющих множество сегментов кодов, вызовы будут far.

В языках высокого уровня вид вызова и модель памяти обычно определяются опциями компилятора, либо в опциях среды программирования, либо в тексте программы.

Помимо вида вызовов и возвратов из подпрограммы в организации межмодульных связей обязательным вопросом является согласование способов передачи аргументов в подпрограммы. На сегодняшний день стандартом стала передача аргументов через стек. В этом случае рассмотрению подлежат следующие вопросы: порядок размещения аргументов в стеке и очистка стека. Решение этих двух вопросов образует понятие «стиль вызова». В табл. 13.2 приведены некоторые стили вызова.

Таблица 13.2

Стиль	Передача аргументов	Очистка стека
Nolanguage	Слева направо	Вызываемая
Basic	Слева направо	Вызываемая
Fortran	Слева направо	Вызываемая
C	Справа налево	Вызывающая
C++	Справа налево	Вызывающая
Pascal	Слева направо	Вызываемая
Stdcall	Справа налево	Вызываемая

В языках высокого уровня стиль вызова задается в описании процедур и функций.

В ассемблерных программах стиль вызова может задаваться в директиве proc:

```
A       proc   pascal, near
        ……..
A       endp

B       proc   c, near
        ……
B       endp
```

В приведенном примере процедура A использует стиль вызова Pascal, процедура B – стиль C.

Директива model также позволяет указывать стиль вызова:

```
.model      small,  c
.model      small, pascal.
```

Стандартный вопрос согласования типов аргументов в ассемблерных программах решается путем указания их размерности при перечислении в директиве proc:

```
A       proc   c, far,          b:byte, c:word, d:dword.
```

Для описания аргументов процедур в ассемблере также может использоваться директива Arg, которая должна следовать за заголовком процедуры:

```
A       proc   c, far,          b:byte, c:word, d:dword.
        arg    e:dword, f:word, =g
```

Директива arg позволяет не только описывать аргументы, но и присваивать значение их длины некоторому имени (в данном примере g получит значение 11).

При передаче параметров по ссылке в стеке размещаются их адреса, размерность которых зависит от используемой модели памяти. Если модель памяти предполагает один сегмент данных, то в стек помещается только смещение до аргумента, если имеется множество сегментов данных – в стеке для каждого аргумента передаваемого по ссылке будет размещен адрес, состоящий из двух частей: смещения и сегментной части.

Поскольку модульное программирование предполагает раздельную компиляцию модулей, неизбежно появляются неопределенные имена, приводящие к ошибкам компиляции. Для исключения этих ошибок предназначены директивы Extrn и Public. В директиве Extrn перечисляются все имена, описанные в других модулях, а в директиве Public – имена, которые должны быть доступны из других модулей.

Для хранения внутренних переменных процедур и функций рекомендуется использовать стек. В языках высокого уровня это реализуется описанием локальных переменных в процедурах (функциях). В ассемблерных подпрограммах для этих целей используется директива Local, которая должна следовать за строками с описанием аргументов:

```
Local a:word, b:dword, c:byte
```

В случаях, когда необходимо сохранить регистры неизменными используется директива Uses, которая сохраняет регистры в стеке при входе в процедуру и восстанавливает их при выходе. Эта директива записывается после local:

```
uses  ax,bx,cx
```

Для иллюстрации механизма работы с процедурами рассмотрим простой пример:

```
                .model      small, c

                .code

                extrn       f:far
                public      p

p               proc        a:dword
                arg         b:word,=c
                local       d:byte, e:word
                uses        ax,bx,cx,dx

                call        f

                ret
p               endp
                end
```

Наибольший интерес представляет листинг, в котором видно все, что появилось в процедуре при трансляции:

```
Turbo Assembler    Version 4.1    26/11/07 01:14:20          Page 1

     1  0000                          .model  small, c
     2  0000                          .code
     3                                extrn   f:far
     4                                public  p
     5  0000                    p     proc    a:dword
     6       =0006                    arg     b:word,=c
     7                                local   d:byte, e:word
     8                                uses    ax,bx,cx,dx
1    9  0000  55                      PUSH    BP
1    10 0001  8B EC                   MOV     BP,SP
1    11 0003  83 EC 04                SUB     SP,00004h
1    12 0006  50                      PUSH    AX
1    13 0007  53                      PUSH    BX
1    14 0008  51                      PUSH    CX
1    15 0009  52                      PUSH    DX
1    16 000A  0E E8 0000e             call    f
1    17 000E  5A                      POP     DX
1    18 000F  59                      POP     CX
1    19 0010  5B                      POP     BX
1    20 0011  58                      POP     AX
1    21 0012  8B E5                   MOV     SP,BP
1    22 0014  5D                      POP     BP
```

```
1    23 0015  C3                  RET    00000h
     24 0016                  p   endp
     25                           end

Turbo Assembler   Version 4.1    26/11/07 01:14:20        Page 2
Symbol Table

Symbol Name              Type   Value                    Cref(defined at #)
         ……..
A                        Number [DGROUP:BP+0004]         #5
B                        Number [DGROUP:BP+0008]         #6
C                        Number 0006                  #6
D                        Number [DGROUP:BP-0002]         #7
E                        Number [DGROUP:BP-0004]         #7
F (_F)                   Far    _TEXT:---- Extern    #3  16
P (_P)                   Near   _TEXT:0000              4  #5
```

Строки 9 и 10 начинают исполняемую часть процедуры. В них содержимое BP помещается в стек, BP устанавливается на верхушку стека. Эти две команды являются следствием директив proc и arg и необходимы для дальнейшего обращения к аргументам и локальным переменным.

Обратим внимание на таблицу символов. Из нее видно, что обращение к аргументам A и B производится через регистр BP со смешениями, учитывающими наличие в стеке старого содержимого BP и адреса возврата, занимающего одно слово в соответствии с видом вызова, предписанным моделью памяти small. Имя C получило значение 6 – размер области стека, занимаемой аргументами.

Строка 11 появилась вследствие использования директивы local. Она отвела в стеке место под локальные переменные (обратите внимание: под байтовую переменную в стеке отводится слово в соответствии с разрядностью модели памяти). Из таблицы символов видно, что для доступа к ним будет использоваться тот же BP, но с отрицательными смещениями.

Строки с 9 по 11 образуют так называемый пролог процедуры, формируемый директивами .model, proc, arg, local.

Эти же директивы привели к появлению перед командой ret в строках 21 и 22 эпилога, в котором освобождается место, отведенное для локальных переменных и восстанавливается содержимое BP.

Директива uses привела к генерации строк с 12-й по 15-ю и с 17-й по 20-ю, которые обеспечивают сохранность регистров.

В строке 16 происходит вызов процедуры F. В коде команды вместо адреса присутствует 0000e, а в таблице символов – extern, свидетельствующие о том, что данный адрес будет корректироваться компоновщиком при сборке исполняемого модуля.

В таблице символов появились имена _F и _P вследствие указания стиля C (компилятор C к пользовательским именам добавляет подчеркивание).

Имя P описано как Near в соответствии с моделью small, а F – far как явно указано в исходном тексте.

Теперь внесем изменения:

```
                .286

                .model      medium, pascal

                .code

                extrn  f:near
                public      p

p               proc   a:dword
                arg    b:word,=c
                local  d:byte, e:word
                uses   ax,bx,cx,dx

                call   f

                ret
p               endp
                end
```

Посмотрим на листинг:

```
Turbo Assembler   Version 4.1    26/11/07 02:44:53        Page 1

     1                                  .286
     2  0000                            .model  medium, pascal
     3  0000                            .code
     4                                  extrn   f:near
     5                                  public  p
     6  0000                       p    proc    a:dword
     7       =0006                      arg     b:word,=c
     8                                  local   d:byte, e:word
     9                                  uses    ax,bx,cx,dx
1    10 0000  C8 0004 00                ENTERW  00004h,0
1    11 0004  50                        PUSH    AX
1    12 0005  53                        PUSH    BX
1    13 0006  51                        PUSH    CX
1    14 0007  52                        PUSH    DX
1    15 0008  E8 0000e                  call    f
1    16 000B  5A                        POP     DX
1    17 000C  59                        POP     CX
1    18 000D  5B                        POP     BX
```

```
1    19 000E  58                    POP     AX
1    20 000F  C9                    LEAVEW
1    21 0010  CA 0006               RET     00006h
     22  0013                   p   endp
     23                             end
Turbo Assembler    Version 4.1    26/11/07 02:44:53          Page 2
Symbol Table

Symbol Name              Type   Value                        Cref(defined at #)

A                        Number [DGROUP:BP+0006]             #6
B                        Number [DGROUP:BP+000A]             #7
C                        Number 0006                    #7
D                        Number [DGROUP:BP-0004]             #8
E                        Number [DGROUP:BP-0002]             #8
F                        Near   SEG_TEXT:---- Extern    #4  15
P                        Far    SEG_TEXT:0000           5  #6
```

Что изменилось: исчезли имена _F и _P (Pascal не изменяет пользовательских имен), F стало near как и предписано (у команды вызова изменился код), P стало far в соответствии с моделью medium, у команды ret появился аргумент, освобождающий дополнительно 6 байт стека (удаление аргументов из стека) – как предписано стилем вызова.

Радикальные изменения произошли с прологом и эпилогом: появились команды Enter и Leave, что является следствием использования системы команд 80286. Первый аргумент команды Enter – размер стека под локальные переменные, второй – уровень (для ассемблерных подпрограмм обычно равен 0).

ЛЕКЦИЯ 14
ОСОБЕННОСТИ ПРОГРАММИРОВАНИЯ В МУЛЬТИПРОГРАММНОЙ И МУЛЬТИЗАДАЧНОЙ СРЕДАХ

План лекции

1. Реентерабельные программы.
2. Синхронизация процессов и потоков.
3. Критические секции.

Перед рассмотрением данной темы необходимо определиться с несколькими понятиями операционных систем.

Задание – описание работы, которую должна выполнить система. Типичным примером задания может служить команда «copy a.txt b.txt». Для выполнения задания в системе обычно порождается процесс.

Процесс – абстрактное понятие, обозначающее владельца ресурсов вычислительной системы, использующихся для выполнения задания. Перечислим основные разновидности ресурсов, закрепляемых за процессом:

области оперативной памяти;
код исполняемой программы;
рабочие файлы;
внешние устройства.

Поскольку процесс порождается для выполнения какого-либо задания, то часто говорят о выполнении процесса. На самом деле выполняется поток, порождаемый для каждого процесса. Процесс может иметь несколько потоков.

Поток (в оригинале thread – нить, поток) – последовательность команд, выполняемых процессором над определенным набором данных, принадлежащих одному процессу.

1. Реентерабельные программы.

В современных операционных системах практически повсеместно реализованы мультипрограммные среды, одновременно выполняющие множество потоков. В таких системах весьма заманчивым вариантом является вынесение типовых рутинных операций в библиотеки стандартных процедур, используемых всеми процессами. Для сокращения объемов занимаемой памяти эти библиотеки не включаются в состав исполняемого модуля при его компоновке, а динамически подключаются во время выполнения. Такие библиотеки получили название DLL – Dynamical Linking Libraries. Большинство системных DLL содержат процедуры и функции, используемые всеми потоками, причем используется единственный загруженный в память экземпляр такой библиотеки.

1. Реентерабельные программы.

Поскольку процессорное время в мультипрограммных системах распределяется между потоками операционной системой по своему усмотрению, заранее невозможно предсказать, в какой точке выполнение потока прервется и начнется выполнение другого потока. В заданном случае возможна ситуация, когда выполнение одного потока прервется после его входа в одну из процедур DLL, а поток, получивший управление, также выполнит вход в ту же самую процедуру DLL. Такая ситуация называется повторным входом в процедуру. Для обеспечения работоспособности процедур в случае повторного входа необходимо, чтобы они были реентерабельными.

Для обеспечения реентерабельности процедур нужно выдерживать ряд требований.

Код процедуры не может быть самомодифицирующимся. Данное требование почти очевидное: если при первом входе код модифицировался, то повторный вход получит модифицированный код, и в общем случае о корректной работе говорить просто не приходится.

Реентерабельная процедура не может вызывать нереентерабельную процедуру.

Процедура не может иметь собственных статических областей памяти для хранения переменных. Проще говоря, сегмент данных такой процедуры может содержать только константы.

Области памяти под переменные процедура должна получать из вызывающего модуля с помощью заранее оговоренного регистра процессора, указывающего на эту область. При соблюдении этого требования переключение контекстов операционной системой приведет к переустановке указателей между областями памяти, содержащими данные разных входов в процедуру.

Реентерабельная процедура может использовать динамически выделяемые области памяти, но указатели на них должны располагаться либо в регистрах процессора, либо в области памяти, полученной из вызывающего модуля.

Результаты из процедур должны возвращаться либо через регистры процессора, либо через область данных, полученную при вызове.

Как это ни странно, но выполнение вышеперечисленных довольно строгих правил реализуется очень простыми средствами: стандартный механизм вызова подпрограмм с передачей параметров через стек и размещением локальных переменных в стеке отвечает почти всем требованиям. Его необходимо только дополнить запретом на использование переменных, расположенных вне стека.

Требование реентерабельности программ для организации их совместного использования не является обязательным: достаточно предпринять меры, исключающие повторный вход. Решение этого вопроса тоже хорошо известно и лежит оно в области синхронизации (взаимного исключения) работы процессов и потоков.

2. Синхронизация процессов и потоков.

При работе с потоками или процессами и организации их параллельной работы часто требуется синхронизация этих потоков или процессов, чтобы они работали друг с другом согласованно. Для синхронизации можно использовать следующее:

1) критические секции;
2) объекты ядра, такие как:
   события;
   мьютексы;
   семафоры;
   ожидающие таймеры.

Критические секции не являются объектами ядра. Их нельзя использовать для синхронизации процессов, их можно использовать только для синхронизации потоков внутри одного процесса.

Синхронизация нужна в случае доступа к ресурсам, не допускающим совместное использование и в некоторых других случаях, таких как доступ к нереентерабельному коду. Общая идеология синхронизации такова: если один поток собирается синхронизироваться с другим, он сообщает операционной системе, что хочет получить доступ к ресурсу, коду или чему бы то ни было еще. ОС помещает этот поток в спящее состояние, пока другой поток, который владеет затребованным ресурсом, не сообщит ей, что этот ресурс свободен для других потоков. В этот момент ожидающий поток находится в критической секции (CS). Конечно, если ресурс свободен в момент запроса, поток получит доступ немедленно.

Обратите внимание: выражение *«ОС помещает этот поток в спящее состояние»*, означает, что поток не будет тратить системное время, пока он ожидает ресурс. Это обратно так называемому busy waiting, когда поток ждет, но на это потребляется системное время. Пример кода:

```
_CSflag        dd 012345678h
  ...
@nextCheck:
   mov eax,012345678h          ; загружаем ненулевое значение
   xchg eax,dword ptr [_CSflag] ; меняем со значением флага
   or eax,eax                  ; проверяем, сброшен  ли CS
   jnz @nextCheck              ; нет, тратим еще время ОС

; начало критической секции

   xor eax,eax                 ; оставляя флаг CS равным нулю
   mov dword ptr [_CSflag],eax
```

```
; конец критической секции
   ...
```

Один поток ждет в цикле, пока другой (только один) в критической секции. Взаимоисключение подтверждено инструкцией xchg.

CS может быть частью программы, которая работает с уникальными ресурсами, и если более чем один поток работает с ней в одно и то же время, он может потерпеть неудачу. Поэтому эта часть код критична, а потоки, работающие с ней, должны взаимно исключать друг друга при работе с этой критической секцией. При использовании IPC в вирусах, требуется синхронизировать каждый зараженный процесс.

Представьте, что у нас есть два потока в каждой зараженной системе. Первый ищет на HDD файлы, которые можно заразить, а второй – заражает их. У нас есть разделяемая область памяти, где ищущий поток сохраняет найденные имена файлов, а заражающий поток загружает их из этой области.

В момент времени у нас может быть любое количество этих потоков. Мы должны убедиться, что только один ищущий поток будет писать в защищенную память и только один заражающий поток будет ждать, пока новые имена не будут загружены в память.

Windows – это преимущественная мультизадачная система, значит запущенные процессы переключаются тогда, когда им это говорит делать система. Поэтому мы никогда не можем знать, насколько быстры наши процессы. Создавая синхронизированный код, мы не можем предполагать скорость выполнения потоков.

2. Критические секции.

Как уже сказано, критические секции не являются объектами ядра и не могут быть использованы для синхронизации процессов. Приведенный выше busy-waiting код бесполезно тратит процессорное время. Ниже приведен тот же код, который использует критические секции.

Перед использованием критической секции мы должны создать ее, применив

```
push указатель на структуру CRITICAL_SECTION
call InitializeCriticalSection
```

Функция ничего не возвращает.

Если имеется мультипроцессорная система, можно использовать:

```
push spin count
push указатель на структуру CRITICAL_SECTION
call InitializeCriticalSectionAndSpinCount
```

Функция ничего не возвращает.

```
CRITICAL_SECTION  struc
        DebugInfo          dd ?
        LockCount          dd ?
        RecursionCount     dd ?
        OwningThread       dd ?
        LockSemaphore      dd ?
        SpinCount          dd ?
CRITICAL_SECCTION  ends
```

После создания критической секции можно использовать ее разделяемый код:

```
        ...
        push указатель на структуру CRITICAL_SECTION
        call EnterCriticalSection
        ; начало критической секции
        ...
        ; здесь может быть только один поток
        ;..
        push указатель на структуру CRITICAL_SECTION
        call LeaveCriticalSection
        ; конец критической секции
        ...
```

Код отличается от того, который был приведен выше, только методом ожидания. Если в критической секции находится один поток, а другой пытается в нее попасть, ОС помещает его в сон.

Если критическая секция больше не нужна, удаляем ее:

```
push указатель на структуру CRITICAL_SECTION
call DeleteCriticalSection
```

Функция ничего не возвращает.

Также существует путь, чтобы узнать, можно ли войти в критическую секцию так, чтобы ОС не поместила поток в сон:

```
push указатель на структуру CRITICAL_SECTION
call TryEnterCriticalSection
```

Функция возвращает ненулевое значение, если удалось войти в критическую секцию.

Последняя функция, которая работает с CS – это SetCriticalSectionSpinCount.

ЛЕКЦИЯ 15
СИНХРОНИЗАЦИЯ ЧЕРЕЗ ОБЪЕКТЫ ЯДРА

План лекции

1. Общие принципы.
2. События.
3. Мьютексы.
4. Семафоры.
5. Ждущие таймеры.

1. Общие принципы.

Предполагается, что все объекты ядра могут использоваться как синхронизационные. Это значит, что они могут находиться в сигнализирующем или несигнализирующем состоянии. Если объект не используется, то он находится в несигнализирующем состоянии. Любой поток, который хочет получить доступ к таком объекту, должен подождать, пока объект не будет просигнализирован (например, процессы и потоки сигнализируются, когда оканчивают свою работу). Доступ к объектам ядра можно получить через handle, который вы получаете как результат работы какой-нибудь функции с примерным названием CreateXOBJECTX.

Чтобы поместить поток в сон и подождать, пока объект будет просигнализирован, можно использовать следующий код:

```
push время ожидания
push handle наблюдаемого объекта
call WaitForSingleObject
```

```
push время ожидания
push булевое значение – ждать всех
push указатель на массив объектов, которые нужно ждать
call WaitForMultipleObjects
```

Булевое значение – установка в TRUE(1) укажет функции ждать все handle, в противном случае (0) функция возвратится после сигнализации одного объекта.

Время ожидания – время в ms, которое функция будет ожидать сигнализацию.

Возможный результат работы этих функций:

WAIT_OBJECT_0 = 0 – объект был просигнализирован – при использовании функции XMultipleX с установкой ждать один объект, мы получим индекс этого handle в переданном массиве.

WAIT_ABANDONED = 080h – описан далее в пункте о мьютексах.

WAIT_TIMEOUT = 0102h – время ожидания вышло, но сигнализации объекта не произошло

WAIT_FAILED = -1 – вызов функции завершен с ошибкой

Вышеприведенный код показывает, как ждать объект. Теперь рассмотрим сами синхронизационные объекты.

2. События.

События – это простые объекты ядра, у которых нет специальных условий, при которых они переключаются в сигнализирующее состояние. Представим, что один из процессов ждет события. Другой процесс может переключить событие с помощью функции SetEvent.

Создать событие можно следующим образом:

```
push указатель на имя события
push булевое значение – начальное состояние
push булевое значение – режим события
push указатель на аттрибуты безопасности
call CreateEvent(A/W)
```

Начальное состояние – событие будет создано в сигнализирующем состоянии (1) или несигнализирующем (0).

Режим события – событие может переводиться в несигнализирующее состояние автоматически (после того, как будет выполнена какая-нибудь из функций WaitX) или вручную (необходимо десигнализировать объект самостоятельно):

```
auto reset   = 0
manual reset = 1
```

Если один процесс создает событие, другой может получить доступ к этому событию, используя его имя:

```
push указатель на имя события
push булевое значение – наследуется или нет
push доступ
call OpenEvent(A/W)
```

Булевое значение – если установлено в TRUE(1), handle события может быть унаследован другими созданными процессами.

Доступ:

EVENT_ALL_ACCESS = 01f0003h – полный доступ к событию

EVENT_MODIFY_STATE = 2 – можно использовать только функции SetEvent и ResetEvent

SYNCHRONIZE = 0100000h – Windows NT: можно использовать только функции WaitX

Функции CreateEvent и OpenEvent возвращают handle события. Есть три функции для работы с состоянием событий. Чтобы установить событие в сигнализирующее состояние:

```
push handle события
call SetEvent
```

Чтобы установить событие в несигнализирующее состояние:

```
push handle события
call ResetEvent
```

```
push handle события
call PulseEvent
```

Эта функция устанавливает событие, пробуждает ждущий поток, а затем сбрасывает событие. Если функция используется на событии, сбрасываемом вручную, все ждущие потоки будут пробуждены, а если событие сбрасывается автоматически, то будет пробуждена только одна ветвь.

Все эти функции возвращают TRUE(1) в случае успеха.

3. Мьютексы.

Слово мьютекс означает mutual exclusion (взаимное исключение). Это один из легких в использовании и очень полезных синхронизационных объектов. Он похож на критические секции, но его можно использовать во взаимодействии между процессами.

Можно создать мьютекс следующим образом:

```
push указатель на имя мьютекса
push булевое значение – инциализация
push указатель на SECURITY_ATTRIBUTES
call CreateMutex(A/W)
```

Инициализация – если TRUE(1), тогда мьютекс создается несигнализированным, в противном случае он доступен кому угодно.

Существующий мьютекс можно открыть так:

```
push указатель на имя мьютекса
```

push булевое значение – наследование

Наследование – если установлено в TRUE(1), handle события может быть унаследован другими созданными процессами.

Доступ:

MUTEX_ALL_ACCESS = 01f0001h полный доступ к мьютексу

SYNCHRONIZE = 0100000h Windows NT: можно использовать только функции WaitX и функцию ReleaseMutex, рассматриваемую ниже.

Обе эти функции возвращают handle мьютекса, если вызов функции прошел успешно.

Предположим, что поток создал мьютекс и владеет им. Другие потоки ждут мьютекса одной из функций WaitX. Если мьютекс потоку больше не нужен – следует освободить его с помощью функции ReleaseMutex. Один ждущий поток проснется и мьютекс снова будет в несигнализированном состоянии. Поток может освободить только тот мьютекс, которым владеет.

```
push handle мьютекса
call ReleasMutex
```

В случае успеха функция возвращает ненулевое значение.

Может случиться, что какой-то поток не освободит мьютекс и закончит работу. Такой мьютекс считается заброшенным, и через некоторое время система проверит его и освободит.

4. Семафоры.

Семафоры – это очень мощные синхронизационные объекты. Как и мьютексы, семафоры наблюдают за входом в критическую секцию, но разница заключается в том, что в одной критической секции может быть больше одного потока. Семафоры могут использоваться для ресурсов с ограниченным количеством.

Можно создать семафор следующим образом:

```
push указатель на имя семафора
push максимальное количество
push начальное количество
push указатель на SECURITY_ATTRIBUTES
call CreateSemaphore(A/W)
```

Максимальное количество – максимальное количество потоков, которое может быть внутри критической секции.

Начальное количество – начальное количество потоков внутри критической секции.

Каждый раз, когда вход в критическую секцию осуществляется через семафоры, Windows уменьшает количество свободных ресурсов, понижает

количество доступов в критическую секцию. Если счетчик семафора равен нулю, вход в критическую секцию закрывается и входящий поток помещается в сон.

Конечно, есть функция API, чтобы открыть существующий семафор:

```
push указатель на имя семафора
push булевое значение – наследование
push доступ
call OpenSemaphore(A/W)
```

Наследование – если установлено в TRUE(1), handle событие может быть унаследовано другими созданными процессами.

Доступ:

SEMAPHORE_ALL_ACCESS = 01f0003h - полный доступ к семафору

SEMAPHORE_MODIFY_STATE = 2 – позволяет использование ReleaseSemaphore

SYNCHRONIZE = 0100000h – позволяет использовать функции WaitX.

Обе эти функции возвращают handle семафора в случае успеха.

Если нет надобности в использовании ресурса, за которым наблюдает семафор, используется функция ReleaseSemaphore, чтобы увеличить количество возможных доступов к ресурсу:

```
push указатель на двойное слово – получаем предыдущее значение счетчика семафора
push насколько увеличить значение семафора
push handle семафора
call ReleaseSemaphore
```

Семафоры находятся в сигнализирующем состоянии, если значение его счетчика не равно нулю, в противном случае он устанавливается в несигнализирующее состояние и входящий поток помещается в сон.

5. Ждущие таймеры.

Ждущие таймеры появились в Windows начиная с NT 4. До сих пор у нас были объекты, которые нужно было вручную (как правило) переключить в сигнализирующее состояние. Ждущие таймеры – это объекты, которые сигнализируют себя сами после некоторого периода времени.

Ждущий таймер создается следующим образом:

```
push указатель на имя
push булевое значение – авто/ручной сброс
push указатель на SECURITY_ATTRIBUTES
call CreateWaitableTime(A/W)
```

Авто/ручной сброс – также, как в событиях

Можно открыть существующий ждущий таймер так:

```
push указатель на имя
push булевое значение – наследование
push доступ
call OpenWaitableTimer(A/W)
```

Наследование – если установлено в TRUE(1), handle события может быть унаследован другими созданными процессами.

Доступ:

TIMER_ALL_ACCESS = 01f0002h – полный доступ

SYNCHRONIZE = 0100000h – позволяет использование функций WaitX

TIMER_MODIFY_STATE = 2 – доступ к SetWaitableTime и CancelWaitableTimer

Обе функции возвращают handle ждущего таймера в случае успеха.

Для установки ждущего таймера:

```
push булевое значение – продолжение
push указатель на аргумент завершающей процедуры
push указатель на завершающую процедуру
push период
push указатель на начальное время
push handle ждущего таймера
call SetWaitableTimer
```

Указатель на начальное время – указание, когда таймер сработает в первый раз. Дата должна быть в формате LARGE_INTEGER.

Период – устанавливает период срабатываний в наносекундах.

Указатель на завершающую процедуру – процедура, выполняющаяся при срабатывании таймера.

Можно сбросить настройки любого ждущего таймера:

```
push handle ждущего таймера
call CancelWaitableTimer
```

ЛЕКЦИЯ 16
МАКРОСРЕДСТВА АССЕМБЛЕРА

План лекции

1. Понятие макросредств.
2. Макрокоманды.
3. Аргументы макрокоманд, исключение дублирования меток.

Программирование на ассемблере представляет совой довольно трудоемкий процесс, требующий крайней внимательности и неоднократного использования довольно длинных последовательностей команд для выполнения в общем-то рутинных операций. Стандартным способом облегчения программирования в такой ситуации является создание библиотек общеупотребительных процедур, которые можно впоследствии использовать для разработки новых программ. К сожалению, готовые процедуры не всегда отвечают конкретным требованиям новых программ и приходится их корректировать, порождая все новые и новые варианты процедур, выполняющих в принципе одни и те же операции. Альтернативой такому способу решения вопроса трудоемкости программирования на ассемблере является использование макросредств.

1. Понятие макросредств.

Макросредства ассемблера – это совокупность конструкций языка, заставляющих транслятор генерировать и/или модифицировать исходный текст программы.

Само по себе понятие макросредств не является уникальной особенностью ассемблера. В C и C++ существует понятие препроцессора, а без директив #include и #define не обходится практически ни одна программа. Препроцессорные директивы являются аналогами макросредств ассемблера. В ассемблере набор макросредств получил наибольшее развитие.

Строго говоря, элементы макросредств в ассемблере часто используются неявно:

директивы proc и local вызывают подстановку в текст программы последовательностей команд, образующих коды пролога и эпилога;

директива uses вызывает генерацию последовательностей команд push и pop;

в TASM в командах сдвигов можно указывать в качестве второго аргумента произвольное число:

```
shl     ax,n,
```

хотя процессор позволяет сдвигать либо на один разряд, либо помещать произвольное количество сдвигов в регистр cl. Транслятор в этом случае подставит в текст программы n команд

```
shl     ax,1
shl     ax,1
……..
shl     ax,1
```

} n раз

в TASM присутствуют директивы обращения к полям записей getfield и setfield, вместо которых транслятор подставит соответствующие последовательности команд;

и т. д.

С учетом возможности использования макросредств ассемблерный транслятор можно рассматривать как совокупность двух взаимосвязанных модулей: макропроцессора и собственно транслятора. Макропроцессор – это модуль, который выискивает в тексте программы макросредства и соответствующим образом модифицирует исходный текст программы. Модифицированный макропроцессором текст далее обрабатывается транслятором, и в итоге получается объектный код программы.

Все макросредства ассемблера можно поделить на четыре группы:

макрокоманды;

блоки повторений;

блоки условной трансляции;

вспомогательные директивы.

2. Макрокоманды.

Механизм макрокоманд является, по-видимому, наиболее используемым из макросредств. По своей сути он позволяет при разработке программы использовать заранее написанные фрагменты текстов, имеющих собственные имена.

Макрокоманды ассемблера часто рассматривают как аналогию процедур, но это не совсем корректно: принципиальное отличие заключается в том, что выполнение макрокоманд происходит не в процессе выполнения программы, а во время трансляции, и сводится к подстановке поименованного фрагмента текста в исходный текст программы.

При работе с макрокомандами используются следующие понятия:

определение макрокоманды;

вызов макрокоманды;

расширение макрокоманды;

генерация расширения макрокоманды.

Определение макрокоманды, часто называемое также макроопределением, – это описание фрагмента текста и присваивание ему некоторого имени. Присвоенное фрагменту имя в дальнейшем используется как имя макрокоманды. Для определения макрокоманд используются директивы macro и endm. Синтаксис директивы macro (рис. 16.1).

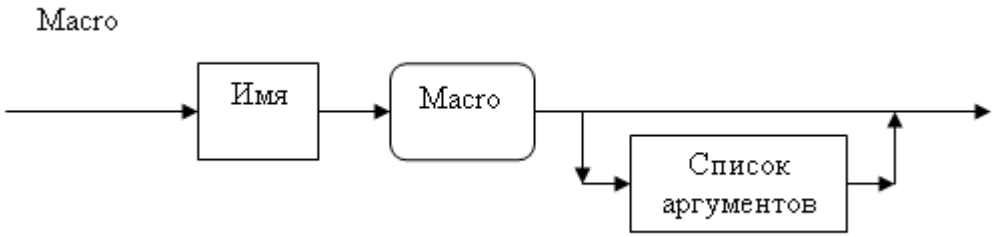


Рис. 16.1. Синтаксис директивы macro

Структура макроопределения:

```
name macro      a, b, c
        ……..
        ……..    тело макроопределения
        ……..
        endm.
```

где name – имя макрокоманды; a, b, c – формальные аргументы.

Тело макроопределения – это любой синтаксически правильный, с точки зрения ассемблера, текст, в котором, как правило, используются формальные аргументы.

Определение макрокоманды в тексте программы должно быть размещено до первого вызова этой макрокоманды.

Типичные примеры определения макрокоманд:

```
;макрокоманда завершения программы
Exit            macro       returncode
                Mov         ah,4ch
                Mov         al,returncode
                Int         21h
                Endm

;макрокоманда вывода  строки символов на экран
Print           macro       string
                Lea         dx,string
                Mov         ah,9
                Int         21h
                Endm
```

Вызов макрокоманды, часто называемый также просто макрокомандой, – это указание имени макрокоманды с последующим перечислением фактических аргументов. Для вышеприведенных определений они выглядят следующим образом

```
                ……
```

```
Msg     db      'данный текст выводится'
        db      'макрокомандой Print$'
        …….
        Print   msg
        Exit    0
```

Расширение макрокоманды, или макрорасширение, – это текст, подставляемый макропроцессором вместо макрокоманды. По сути, макрорасширение – это тело макроопределения, в котором формальные аргументы заменены на фактические, указанные в макрокоманде. Для вышеприведенной последовательности макрокоманд это:

```
Lea     dx,msg   ┐
Mov     ah,9     ├ расширение для print
Int     21h      ┘
Mov     ah,4ch   ┐
Mov     al,0     ├ расширение для exit
Int     21h      ┘
```

Генерация расширения макрокоманды (или просто макрогенерация) – процесс замены макрокоманды на ее расширение.

3. Аргументы макрокоманд, исключение дублирования меток.

В директиве macro может указываться список формальных аргументов, синтаксис которого иллюстрируется следующей диаграммой (рис. 16.2).

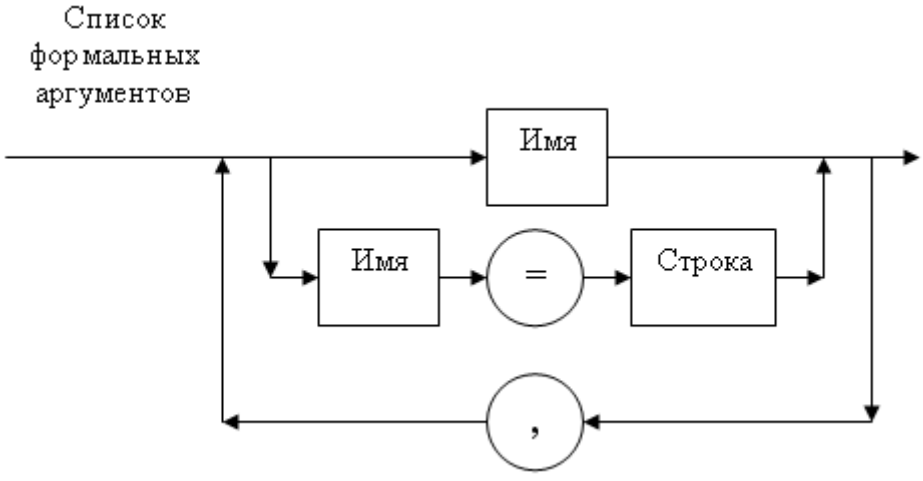


Рис. 16.2. Синтаксис списка формальных аргументов

Синтаксис списка фактических аргументов, указываемых в макрокоманде (рис. 16.3).

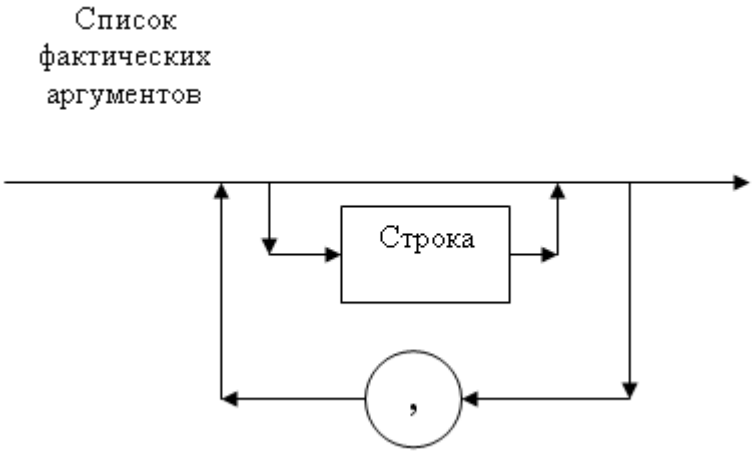


Рис. 16.3. Синтаксис списка фактических аргументов

Как видно из синтаксиса списка формальных аргументов макрокоманды, допускается запись вида *имя=строка*. Эта запись позволяет задать значение формального аргумента по умолчанию, то есть если соответствующий фактический аргумент в вызове макрокоманды опущен (что разрешается синтаксической диаграммой списка фактических аргументов), то он получает значение *строка*, указанное после знака равенства в списке формальных аргументов.

Если в вызове макрокоманды опущен аргумент, не имеющий значения по умолчанию, он получает пустое значение. Например, для макроопределения

```
Sum     macro   a=bx, b, c
        Mov     ax,a
        Add     ax,b
        Mov     c,ax
        endm
```

вызов макрокоманды

```
        Sum     , , bp
```

приведет к генерации следующего расширения:

```
        Mov     ax,bx
        Add     ax,
        Mov     bp,ax
```

Из приведенного примера видно, что пустое значение второго аргумента макрокоманды привело к отсутствию второго операнда у команды Add, что является синтаксической ошибкой.

Если строка, указываемая в качестве фактического аргумента макрокоманды, содержит разделители, то она заключается в угловые скобки. Например, определение

```
A       macro       b,c,d
        b
        c
        d
        endm
```

позволяет использовать макрокоманду

```
        A       <inc   ax>, <xor    bx,bx>, <mov       cx,10>
```

дающую расширение из трех строк

```
        inc     ax
        xor     bx,bx
        mov     cx,10
```

При использовании макрокоманд часто возникает вопрос дублирования меток, имеющихся в теле макроопределения, и соответственно появляющихся в расширениях макрокоманд. Для решения этого вопроса простейшим является вынесение имен меток в список аргументов директивы macro:

```
A       macro       ……, m
        ……
m:      ……
        ……
        endm
```

В этом случае в каждом вызове макрокоманды необходимо явно указывать новое имя метки.

Вместо того чтобы каждый раз заново придумывать имена меток, можно использовать какую-либо систему формирования этих имен, вроде XX1, XX2, XX3 и т. д.

В этом случае помощь может оказать директива склеивания текстовой строки с аргументом макрокоманды &:

```
A           macro       ……, m
            ……
XX&m:       ……
            ……
            endm
```

Здесь в качестве аргумента m указывается только изменяемая часть имени метки 1, 2, 3 и т. д.

Аналогичный механизм формирования меток в макрокомандах уже встроен в ассемблер: транслятор может автоматически генерировать символы (имена) вида ??xxxx, где xxxx – числа 0000, 0001, 0002 и т. д. в расширениях макрокоманд. Для этого сразу после директивы macro необходимо использовать директиву local, имеющую следующий формат (рис. 16.4).

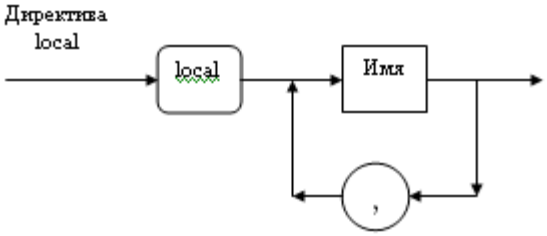


Рис. 16.4. Синтаксис директивы local

Имя – имя метки, используемой в теле макроопределения.

ЛЕКЦИЯ 17
БЛОКИ ПОВТОРЕНИЙ

План лекции

1. Директивы rept и while.
2. Директивы IRP и IRPC.
3. Пример макробиблиотеки.

Блоки повторений заставляют транслятор подряд повторять фрагменты текста несколько раз. Их синтаксис (рис. 17.1).

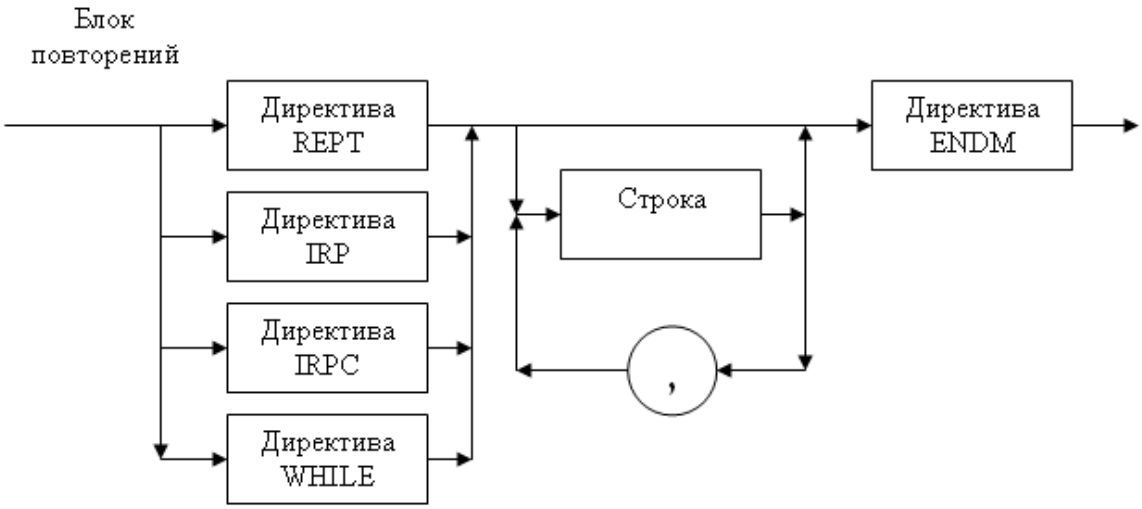


Рис. 17.1. Синтаксис блока повторений

1. Директивы rept и while.

Синтаксис директив rept и while:

Rept выражение

While выражение

При использовании директивы WHILE макрогенератор транслятора будет повторять последовательность строк до тех пор, пока значение *выражение* не станет равно нулю. Это значение вычисляется каждый раз перед очередной итерацией цикла повторения (то есть значение *выражение* должно подвергаться изменению внутри *последовательность строк* в процессе макрогенерации).

Директива REPT, подобно директиве WHILE, повторяет *последовательность строк* столько раз, сколько это определено значением *выражение*. Отличие этой директивы от WHILE состоит в том, что она автоматически уменьшает на единицу значение *выражение* после каждой итерации.

Простейшим блоком повторений является использование директивы rept:

```
Rept  10
Shl   ax,1
```

```
Endm
```

Приведенный блок повторений заставит транслятор сгенерировать линейную последовательность из десяти команд shl ax,1. Еще один пример:

```
T = 0
Rept  5
Dw    t
T = T+1
Endm
```

Данная запись эквивалентна

```
Dw    0, 1, 2, 3, 4, 5
```

2. Директивы IRP и IRPC.

Директива IRP имеет следующий синтаксис:

```
IRPформальный_аргумент,<строка_символов1,...,строка_символовN>
Последовательность_строк
ENDM
```

Действие данной директивы заключается в том, что она повторяет последовательность_строк N раз, то есть столько раз, сколько *строк_символов* заключено в угловые скобки во втором операнде директивы IRP. Повторение *последовательности_строк* сопровождается заменой в ней *формального_аргумента* строкой символов из второго операнда. Так, при первой генерации *последовательности_строк формальный_аргумент* в них заменяется на *строка_символов1*. Если есть *строка_символов2*, то это приводит к генерации второй копии *последовательности_строк*, в которой *формальный_аргумент* заменяется на *строка_символов2*. Эти действия продолжаются до *строка_символовN* включительно.

```
Irp     x, ax, bx, cx
Push x
endm
```

Фактически будет сгенерирована последовательность команд:

```
Push ax
Push bx
Push cx
```

Директива IRPC имеет следующий синтаксис:

```
IRPC формальный_аргумент,строка_символов
Последовательность_строк
ENDM
```

Действие данной директивы подобно IRP, но отличается тем, что она на каждой очередной итерации заменяет *формальный_аргумент* очередным символом из *строка_символов*. Понятно, что количество повторений *последовательность_строк* будет определяться количеством символов в *строка_символов*.

Вышеприведенную последовательность команд можно записать

```
Irpc   r, abc
Push r&x
endm
```

3. Пример макробиблиотеки.

В заключение пример небольшой макробиблиотеки:

```
exit   macro
       mov       ah,4ch
       int       21h
       endm

pushr  macro     registers
       irp       x,<registers>
       push      x
       endm
       endm

popr   macro     registers
       irp       x,<registers>
       pop       x
       endm
       endm

print  macro     string
       pushr     <dx,ax>
       lea       dx,string
       mov       ah,9
       int       21h
       popr      <ax,dx>
       endm
```

```
gtlin   macro       mes,string
        pushr       <ax,dx,di>
        print       mes
        mov         ah,0ah
        lea         dx,string
        int         21h
        lea         di,string+2
        mov         al,-1[di]
        xor         ah,ah
        add         di,ax
        mov         [di],ah
        popr        <di,dx,ax>
        endm

wrword  macro       num
        local       m,m1
        pushr       <ax,bx,dx,cx>
        xor         cx,cx
        mov         bx,10
        mov         ax,num
m:      inc         cx
        xor         dx,dx
        div         bx
        push        dx
        or          ax,ax
        jnz         m
m1:     pop         dx
        add         dx,'0'
        mov         ah,2
        int         21h
        loop        m1
        popr        <cx,dx,bx,ax>
        endm

wrint   macro       num
        local       m,m1
        pushr       <ax,bx,dx,cx>
        xor         cx,cx
        mov         bx,10
        mov         ax,num
        cmp         ax,0
        jge         m
        neg         ax
        push        ax
        mov         ah,2
```

```
        mov     dl,'-'
        int     21h
        pop     ax
m:      inc     cx
        xor     dx,dx
        div     bx
        push    dx
        or      ax,ax
        jnz     m
m1:     pop     dx
        add     dx,'0'
        mov     ah,2
        int     21h
        loop    m1
        popr    <cx,dx,bx,ax>
        endm
```

ЗАКЛЮЧЕНИЕ

Наличие макросредств делает язык ассемблера не таким уж низкоуровневым. Продуманный набор макрокоманд позволяет писать программы на «макроязыке» с использованием достаточно крупных конструкций, сравнимых, а может быть и превосходящих, операторы процедурно-ориентированнных языков высокого уровня. Набор таких конструкций в случае необходимости легко расширяется самим программистом.

Наряду с несомненными достоинствами использования макросредств необходимо отметить один неприятный момент: непродуманное использование директив повторения (особенно вложенных одна в другую) может привести к генерации программ непомерно большого объема, поэтому необходимо соблюдать осторожность и тщательно оценивать предполагаемые размеры генерируемого текста программы.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

Основной

1. Юров, В. Assembler : учеб. / В. Юров. – СПб. : Питер, 2001.
2. Юров, В. Assembler : практикум / В. Юров. – СПб. : Питер, 2001.
3. Юров, В. Assembler : спец. справ. / В. Юров. – СПб. : Питер, 2000.
4. Пильщиков, В. Н. Программирование на языке ассемблера IBM PC / В. Н. Пильщиков. – М. : Диалог-МИФИ, 1997. – 288 с.
5. Финогенов, К. Г. Самоучитель по системным функциям MS-DOS / К. Г. Финогенов. – 2-е изд. – М. : Радио и связь ; Энтроп, 1995. – 382 с.
6. Нортон, П. Язык ассемблера для IBM PC / П. Нортон, Д. Соухэ. – М. : Финансы и статистика, 1992. – 352 с.
7. Нортон, П. Персональный компьютер IBM и операционная система MS-DOS / П. Нортон. – М. : Радио и связь, 1991. – 416 с.

Дополнительный

8. Белецкий, Я. Турбо Ассемблер. Версия 2.0 / Я. Белецкий. – М. : Машиностроение, 1994. – 160 с.
9. Абель, П. Язык ассемблера для IBM PC и программирования / П. Абель. – М. : Высш. шк., 1992. – 487 с.
10. Джордейн, Р. Справочник программиста персональных компьютеров типа IBM PC, XT и AT : пер. с англ. / Р. Джордейн. – М. : Финансы и статистика, 1992. – 544 с.
11. Скэнлон, Л. Персональные ЭВМ IBM PC и XT. Программирование на языке ассемблера : пер. с англ. / Л. Скэнлон. – М. : Радио и связь, 1989. – 336 с.
12. Рудаков, П. И. Программируем на языке ассемблера IBM PC / П. И. Рудаков, К. Г. Финогенов. – 2-е изд. – Обнинск : Принтер, 1997. – 584 с.
13. Сван, Т. Освоение Turbo Assembler / Т. Сван. – Киев : Диалектика, 1996. – 544 с.
14. Юров, В. Assembler : учеб. курс / В. Юров, С. Хорошенко. – СПб. : Питер Ком, 1999. – 672 с.
15. Айден, К. Аппаратные средства PC : пер. с нем. / К. Айден, Х. Фибельман. – 2-е изд. – М. : Крамер; СПб. : BHV – Санкт-Петербург, 1998.