Ассемблер для процессоров Intel Pentium

Ю. Магда

Оглавление

Список литературы

Введение

Эта книга посвящена описанию возможностей языка ассемблера процессоров Intel Pentium. Книга не является учебником по языку ассемблера, хотя и может использоваться в этом качестве, — скорее это расширенное руководство по применению ассемблера процессоров Intel Pentium. Материал книги содержит много справочной информации по командам ассемблера и современным технологиям обработки данных. Изучение современного ассемблера — задача далеко не простая, и эта книга позволит читателю успешно ее решить.

Язык ассемблера появился вместе с появлением процессоров и тесно связан с их архитектурой, позволяя напрямую обращаться к аппаратным ресурсам компьютера. Часто у читателей возникает вопрос: а зачем вообще нужно изучать язык ассемблера, когда имеются развитые средства программирования на языках высокого уровня, такие, например, как Visual C++ .NET фирмы Microsoft или Borland Delphi 2005? Тем более что помимо этих средств есть еще целый спектр специализированных программных продуктов для разработки офисных приложений, баз данных, электронных таблиц и т. д. Подобные программы называются средствами быстрой разработки и позволяют в считанные недели создавать самые сложные приложения.

Тем не менее, значение языка ассемблера трудно переоценить. Все без исключения средства разработки программ в той или иной степени используют ассемблер. К примеру, большинство библиотечных функций языка C++ и Pascal, на основе которых построены такие мощные инструменты разработки, как Visual C++ и Delphi, написаны на ассемблере. Мультимедийные приложения, программы обработки сигналов и многие другие используют высокопроизводительные библиотеки функций, разработанные с помощью ассемблерных команд технологии SIMD. Наконец, если требуется, чтобы приложение работало максимально быстро и занимало меньше памяти (а это нужно для встроенных и мобильных систем в различных отраслях промышленности), то применение ассемблера является едва ли не единственным способом достижения цели. По этой причине большинство приложений, работающих в режиме реального времени, либо написаны целиком на ассемблере, либо используют в критических участках кода ассемблерный код.

Даже по этим нескольким примерам видно, что язык ассемблера имеет свои сферы применения, свои ниши, которые никогда и ничем не будут заняты. Кроме того, как уже отмечалось, при разработке приложений на языках высокого уровня критические секции, требующие высокой скорости выполнения, пишутся на ассемблере. Именно поэтому в Visual C++ .NET и Delphi 2005 имеется возможность создавать программный код на встроенном ассемблере. Должен заметить, что фирма Microsoft постоянно совершенствует встроенный ассемблер.

Вряд ли кому-то придет в голову разрабатывать большие и многофункциональные приложения на языке ассемблера, но ускорить производительность работы таких приложений с помощью ассемблера можно. По сравнению с языками высокого уровня ассемблер обладает одним фундаментальным преимуществом, проистекающим из его природы — он позволяет писать самый быстрый и компактный код. Изучение языка ассемблера дает программисту одно очень важное преимущество — он глубже начинает понимать принципы работы приложений, написанных на любых языках, в том числе и на языках высокого уровня. Ассемблер очень помогает при разработке программ на языках высокого уровня, поскольку знание низкоуровневого программирования позволяет выбирать оптимальные решения.

Что же касается инструментальных средств для разработки приложений на «чистом» ассемблере, то в последнее время появились очень мощные приложения такого рода, что вынуждает по-другому взглянуть на проблему. Из таких инструментальных средств проектирования можно выделить в первую очередь макроассемблер MASM32, а также AsmStudio и NASM. Эти и другие инструменты разработки программ имеют самый современный графический интерфейс. Не следует забывать и о том, для ассемблера разработаны многочисленные библиотеки функций, приближающие этот язык по своим функциональным возможностям к высокоуровневым средствам разработки приложений.

Материал книги охватывает полный спектр архитектурно-программных решений для процессоров Intel Pentium, включая как базовую программную архитектуру и набор основных команд ассемблера, так и современные технологии параллельной обработки данных (SIMD). Эта книга задумана как расширенный справочник по применению ассемблера в практических разработках, хотя может быть использована и как практическое пособие для программистов-разработчиков, желающих углубить свои знания о современных технологиях программирования на ассемблере.

Материал книги включает много примеров программного кода, в том числе и для технологий SIMD. Этими практическими примерами подкрепляются большинство теоретических аспектов, рассматриваемых в книге. По мнению автора, такой путь является наиболее эффективным для изучения языка ассемблера.

Все примеры программ являются полностью работоспособными и проверены автором. Они демонстрируют ключевые моменты использования тех или иных команд или технологий и реализованы в виде коротких процедур. Такая методика выбрана сознательно, поскольку длинные и сложные программы обычно запутывают читателя, и при их анализе легко теряются ключевые моменты, ради которых эти программы собственно и были разработаны. Любой пример несложно адаптировать для дальнейшего использования в собственных разработках.

Для разработки примеров используется макроассемблер MASM фирмы Microsoft с компилятором версии 7.10.*xxxx*. Этот компилятор включен в состав Windows XP DDK и Windows Server 2003 DDK. Подойдет и компилятор версии 6.14.*xxxx*, но в этом случае примеры применения технологий SIMD компилировать будет невозможно. В качестве среды разработки можно порекомендовать свободно распространяемый макроассемблер MASM32 версии 8, который включает в себя компилятор ML версии 6.14.*xxxx* и компоновщик LINK версии 5.12.*xxxx* фирмы Microsoft.

Во всех примерах синтаксис языка ассемблера максимально упрощен, используется минимум высокоуровневых конструкций языка. В книге не приводится детальное описание компилятора MASM, а упоминаются лишь те сведения, которые необходимы для работы.

Книга рассчитана на широкий круг читателей — от начинающих программистов до опытных разработчиков.

Структура книги

Структура книги такова, что материал можно изучать выборочно по отдельным главам, или последовательно, начиная с первой главы. Это позволяет различным категориям читателей изучать тот материал, который им более всего интересен.

Книга состоит из 14 глав.

- Глава 1, «Базовая архитектура процессоров Intel x86». В этой главе рассматривается базовая архитектура процессоров x86 фирмы Intel и эволюция к последним моделям процессоров Intel Pentium.
- Глава 2, «Основы создания приложений на языке ассемблера». Материал этой главы посвящен общим принципам создания программ на ассемблере. Здесь также рассмотрены основные этапы компиляции и компоновки приложений с использованием макроассемблера MASM фирмы Microsoft.
- Глава 3, «Синтаксис языка ассемблера». В этой главе проанализирован синтаксис языка ассемблера, включая основные типы данных, модели памяти и типы адресации при работе с процессорами Intel.
- Глава 4, «Структура программ на языке ассемблера». В этой главе проанализирована сегментная структура ассемблерных программ и ее взаимосвязь с используемыми моделями памяти.
- Глава 5, «Организация вычислительных циклов». Материал главы посвящен организации вычислительных алгоритмов с использованием команд условных и безусловных переходов. Здесь также рассматриваются варианты оптимизации ветвлений в программах с применением специальных команд процессоров Intel Pentium.
- Глава 6, «Процедуры на языке ассемблера». В этой главе описан процесс разработки и применения процедур на языке ассемблера, а также вопросы организации и использования стека для передачи параметров. Рассмотрены различные варианты обработки данных в процедурах, обращений к регистрам и памяти.
- Глава 7, «Операции со строками и массивами». Здесь рассматриваются строковые команды процессора Intel Pentium и практические аспекты их применения при обработке символьных строк и числовых массивов. Проанализированы методы оптимизации строковых операций.
- Глава 8, «Арифметические и логические операции». Материал главы посвящен анализу арифметических и логических команд процессора, а также преобразованиям целочисленных данных из одних форматов в другие.

- Глава 9, «Использование математического сопроцессора». Здесь рассматриваются вопросы применения математического сопроцессора в операциях над числами с плавающей точкой и способы создания эффективных алгоритмов обработки данных.
- Глава 10, «Интерфейс с языками высокого уровня». Материал главы посвящен применению отдельно скомпилированных ассемблерных модулей в программах на языках высокого уровня. В главе подробно анализируются методы передачи параметров в процедуры и получения результатов.
- Глава 11, «Процессоры Intel Pentium в современных разработках». В главе рассматриваются общие вопросы применения процессоров последних поколений Intel Pentium 4 в разработке высокоэффективных приложений. Показаны возможности оптимизации приложений для процессоров Pentium 4.
- Глава 12, «MMX-расширение процессоров Intel Pentium». Здесь проанализированы основные аспекты использования технологии MMX для повышения производительности мультимедийных приложений и операций с целыми числами.
- Глава 13, «SSE-расширение процессоров Intel Pentium». В главе рассматриваются основные аспекты применения технологии SSE для повышения производительности операций с плавающей точкой в коротком формате и возможности оптимизации программ.
- Глава 14, «Технология SSE2 в процессорах Intel Pentium 4». Глава посвящена вопросам применения технологии SSE2 для повышения производительности операций с плавающей точкой двойной точности. Материал сопровождается многочисленными примерами практического применения данной технологии.

Материал книги дополнен справочником по системе команд процессоров Intel. Поскольку полная система команд насчитывает несколько сотен наименований, приведены только наиболее часто используемые команды.

Автор благодарит коллектив издательства «Питер» за помощь в подготовке книги к изданию. Особая признательность жене Юлии за поддержку и помощь в написании книги.

Глава 4. Структура программы на языке ассемблера

Материал этой главы посвящен вопросам организации и компоновки программного кода на языке ассемблера. Затронуты вопросы взаимодействия различных частей ассемблерной программы, организации сегментов программного кода, данных и стека в контексте различных моделей памяти. Напомню, что мы рассматриваем эти аспекты применительно к макроассемблеру MASM фирмы Microsoft, хотя многие положения действительны и для других компиляторов. Начнем с анализа сегментов. Мы уже сталкивались с этими вопросами в главе 3, сейчас же рассмотрим их более детально.

4.1. Организация сегментов

Для хорошего понимания, как работает программа на ассемблере, нужно очень четко представлять себе организацию сегментов. Применительно к процессорам Intel Pentium термин “сегмент” имеет два значения:

- Область физической памяти заранее определенного размера. Для 16-разрядных процессоров размер сегмента физической памяти не может превышать 64 Кбайт, в то время как для 32-разрядных может достигать 4 Гбайт.
- Область памяти переменного размера, в которой могут находиться программный код, данные или стек.

Физический сегмент может располагаться только по адресу, кратному 16, или, как иногда говорят, по границе параграфа. Логические сегменты тесно связаны с физическими. Каждый логический сегмент ассемблерной программы определяет именованную область памяти, которая адресуется селектором сегмента, содержащимся в сегментном регистре. Сегментированная архитектура создает определенные трудности в процессе разработки программ. Для небольших программ, меньших 64 Кбайт, программный код и данные могут размещаться в отдельных сегментах, поэтому никаких особых проблем не возникает.

Для больших программ, занимающих несколько сегментов кода или данных, необходимо правильно адресовать данные, находящиеся в разных сегментах данных. Кроме того, если программный код находится в нескольких сегментах, то усложняются реализация переходов и

ветвлений в программе, а также вызовы процедур. Во всех этих случаях требуется задавать адреса в виде *сегмент*:*смещение*.

При использовании 32-разрядного защищенного режима эти проблемы исчезают. Например, в плоской модели памяти (о ней мы поговорим чуть позже) для адресации программного кода и данных достаточно 32-разрядного эффективного адреса внутри непрерывной области памяти.

Логические сегменты могут содержать три основных компонента программы: программный код, данные и стек. Макроассемблер MASM обеспечивает правильное отображение этих компонентов на физические сегменты памяти, при этом сегментные регистры `CS`, `DS` и `SS` содержат адреса физических сегментов памяти.

4.2. Директивы управления сегментами и моделями памяти макроассемблера MASM

В макроассемблер MASM включены директивы, упрощающие определение сегментов программы и, кроме того, предполагающие те же соглашения, которые используются в языках высокого уровня Microsoft. Упрощенные директивы определения сегментов генерируют необходимый код, указывая при этом атрибуты сегментов и порядок их расположения в памяти. Везде в этой книге мы будем использовать именно упрощенные директивы определения сегментов, наиболее важные из которых перечислены далее:

- `.DATA` (`.data`) — определяет начало инициализированного сегмента данных с именем `_DATA` и при наличии предыдущего сегмента завершает его. Этой директиве должна предшествовать директива `.MODEL`. Сегмент, определенный с атрибутом `.DATA`, должен содержать только инициализированные данные, то есть имеющие начальные значения, например:

```
.data
  val1    DW 11
  string1 DB "Text string"
  byte1   DB ?
```

- `.DATA?` (`.data?`) — определяет сегмент данных, в котором располагаются неинициализированные данные. При наличии предыдущего сегмента новый сегмент завершает его. Неинициализированные данные могут объявляться в сегменте `.DATA?` при помощи оператора `?`. Преимуществом директивы `.DATA?` является то, что при ее использовании уменьшается размер исполняемого файла и, кроме того, обеспечивается лучшая совместимость с другими языками. Этой директиве должна предшествовать директива `.MODEL`. Вот пример использования директивы `.DATA?`:

```
.data?
   DB 5 DUP (?)
```

- .CONST (.const) — определяет начало сегмента данных, в котором определены константы. При наличии предыдущего сегмента новый сегмент завершает его. В целях совместимости с другими языками данные должны быть в формате, совместимом с принятыми в языках высокого уровня соглашениями. Сегмент, определенный директивой .CONST, имеет атрибут “только для чтения”. Этой директиве должна предшествовать директива .MODEL.
- .STACK (.stack) [размер] — определяет начало сегмента стека с указанным размером памяти, который должен быть выделен под область стека. Если параметр не указан, размер стека предполагается равным 1 Кбайт. При наличии предыдущего сегмента новый сегмент завершает его. Этой директиве должна предшествовать директива .MODEL.
- .CODE (.code) [имя] — определяет сегмент программного кода и заканчивает предыдущий сегмент, если таковой имеется. Необязательный параметр имя замещает имя _TEXT, заданное по умолчанию. Если имя не определено, ассемблер создает сегмент с именем _TEXT для моделей памяти tiny, small, compact и flat или сегмент с именем имя_модуля_TEXT для моделей памяти medium, large и huge. Этой директиве должна предшествовать директива .MODEL, указывающая модель памяти, используемую программой.
- .MODEL (.model) модель_памяти [,соглашение_о_вызовах] [,тип_ОС] [,параметр_стека] — определяет модель памяти, используемую программой. Директива должна находиться перед любой из директив объявления сегментов. Она связывает определенным образом различные сегменты программы, определяемые ее параметрами tiny, small, compact, medium, large, huge или flat. Параметр модель_памяти является обязательным.

Если разрабатывается процедура для включения в программу, написанную на языке высокого уровня, то должна быть указана та модель памяти, которая используется компилятором языка высокого уровня. Кроме того, модель памяти должна соответствовать режиму работы (типу) процессора. Это имеет значение для плоской модели памяти, которую можно применять только в режимах `.386`, `.486`, `.586`, `.686`. Модель памяти определяет, какой тип адресации данных и команд поддерживает программа (`near` или `far`). Это имеет смысл для команд перехода, вызовов и возврата из процедур. В табл. 4.1 демонстрируются эти особенности.

Модель памяти	Адресация кода	Адресация данных	Операционная система	Чередование кода и данных
TINY	NEAR	NEAR	MS-DOS	Допустимо
SMALL	NEAR	NEAR	MS-DOS, Windows	Нет
MEDIUM	FAR	NEAR	MS-DOS, Windows	Нет
COMPACT	NEAR	FAR	MS-DOS, Windows	Нет
LARGE	FAR	FAR	MS-DOS, Windows	Нет
HUGE	FAR	FAR	MS-DOS, Windows	Нет
FLAT	NEAR	NEAR	Windows NT, Windows 2000, Windows XP, Windows 2003	Допустимо

Таблица 4.1. *Параметры моделей памяти*

Все семь моделей памяти поддерживаются всеми компиляторами MASM, начиная с версии 6.1.

Модель `small` поддерживает один сегмент кода и один сегмент данных. Данные и код при использовании этой модели адресуются как `near` (ближние). Модель `large` поддерживает несколько сегментов кода и несколько сегментов данных. По умолчанию все ссылки на код и данные считаются дальними (`far`).

Модель `medium` поддерживает несколько сегментов программного кода и один сегмент данных, при этом все ссылки в сегментах программного кода по умолчанию считаются дальними (`far`), а ссылки в сегменте данных — ближними (`near`). Модель `compact` поддерживает несколько сегментов данных, в которых используется дальняя адресация данных (`far`), и один сегмент кода с ближней адресацией (`near`). Модель `huge` практически эквивалентна модели памяти `large`.

Должен заметить, что разработчик программ может явно определить тип адресации данных и команд в различных моделях памяти. Например, ссылки на команды внутри одного сегмента кода в модели `large` можно сделать ближними (`near`). Проанализируем, в каких случаях лучше всего подходят те или иные модели памяти.

Модель `tiny` работает только в 16-разрядных приложениях MS-DOS. В этой модели все данные и код располагаются в одном физическом сегменте. Размер программного файла в этом случае не превышает 64 Кбайт. С другой стороны, модель `flat` предполагает несегментированную конфигурацию программы и используется только в 32-разрядных операционных системах. Эта модель подобна модели `tiny` в том смысле, что данные и код размещены в одном сегменте, только 32-разрядном. Хочу напомнить, что многие примеры из этой книги разработаны именно для модели `flat`.

Для разработки программы для модели `flat` перед директивой `.model flat` следует разместить одну из директив: `.386`, `.486`, `.586` или `.686`. Желательно указывать тот тип процессора, который используется в машине, хотя на машинах с Intel Pentium можно указывать директивы `.386` и `.486`. Операционная система автоматически инициализирует сегментные регистры при загрузке программы, поэтому модифицировать их нужно, только если необходимо смешивать в одной программе 16- и 32-разрядный код. Адресация данных и кода является ближней (`near`), при этом все адреса и указатели являются 32-разрядными.

Параметр *соглашение_о_вызовах* используется для определения способа передачи параметров при вызове процедуры из других языков, в том числе и языков высокого уровня (C++, Pascal). Параметр может принимать следующие значения: `C`, `BASIC`, `FORTRAN`, `PASCAL`, `SYSCALL`, `STDCALL`. При разработке модулей на ассемблере, которые будут применяться в программах, написанных на языках высокого уровня, обращайте внимание на то, какие соглашения о вызовах поддерживает тот или иной язык. Более подробно соглашения о вызовах мы будем рассматривать при анализе интерфейса программ на ассемблере с программами на языках высокого уровня.

Параметр *тип_ОС* равен `OS_DOS`, и на данный момент это единственное поддерживаемое значение этого параметра.

Наконец, последний параметр *параметр_стека* устанавливается равным `NEARSTACK` (регистр `SS` равен `DS`, области данных и стека размещаются в одном и том же физическом сегменте) или `FARSTACK` (регистр `SS` не равен `DS`, области данных и стека размещаются в разных физических сегментах). По умолчанию принимается значение `NEARSTACK`. Рассмотрим примеры использования директивы `.MODEL`:

```
.model flat, c
```

Здесь параметр `flat` указывает компилятору на то, что будет использоваться 32-разрядная линейная адресация. Второй параметр `c` указывает, что при вызове ассемблерной процедуры из другой программы (возможно, написанной на другом языке) будет задействован способ передачи параметров, принятый в языке C. Следующий пример:

```
.model large, c, farstack
```

Здесь используются модель памяти `large`, соглашение о передаче параметров языка C и отдельный сегмент стека (регистр `SS` не равен `DS`).

```
.model medium, pascal
```

В этом примере используются модель `medium`, соглашение о передаче параметров для Pascal и область стека, размещенная в одном физическом сегменте с данными.

4.3. Структура программ на ассемблере MASM

Программа, написанная на ассемблере MASM, может состоять из нескольких частей, называемых модулями, в каждом из которых могут быть определены один или несколько сегментов данных, стека и кода. Любая законченная программа на ассемблере должна включать один главный, или основной (main), модуль, с которого начинается ее выполнение. Основной модуль может содержать программные сегменты, сегменты данных и стека, объявленные при помощи упрощенных директив. Кроме того, перед объявлением сегментов нужно указать модель памяти при помощи директивы `.MODEL`. Поскольку подавляющее большинство современных приложений являются 32-разрядными, то основное внимание в этом разделе мы уделим именно таким программам, хотя не обойдем вниманием и 16-разрядные программы, которые все еще используются. Начнем с 16-разрядных программ.

В следующем примере показана 16-разрядная программа на ассемблере, в которой используются упрощенные директивы ассемблера MASM:

```
.model small, c  ; эта директива указывается до объявления
                 ; сегментов
.stack 100h      ; размер стека 256 байт
.data            ; начало сегмента данных
. . .
; данные
. . .
.code            ; здесь начинается сегмент программ
main:
. . .
; команды ассемблера
. . .
end main
end
```

Здесь оператор `end main` указывает на точку входа `main` в главную процедуру. Оператор `end` закрывает последний сегмент и обозначает конец исходного текста программы. В 16-разрядных приложениях MS-DOS можно инициализировать сегментные регистры так, чтобы они указывали на требуемый логический сегмент данных. Листинг 4.1 демонстрирует это.

```
.model large
.data
 s1 DB "TEST STRING$"
.code
 mov AX, @data
 mov DS, AX
 lea DX, s1
 mov AH, 9h
 int 21h
 mov ax, 4c00h
 int 21h
endъ
```

Листинг 4.1. *Пример адресации сегментов в программе MS-DOS*

Здесь на экран дисплея выводится строка `s1`. При помощи следующих команд в сегментный регистр `DS` помещается адрес сегмента данных, указанного директивой `.data`:

```
mov AX, @data
mov DS, AX
```

Затем строка `s1`, адресуемая через регистры `DS:DX`, выводится на экран с использованием прерывания 9h функции 21h MS-DOS. Попробуйте закомментировать проанализированные две строки кода и посмотреть на результат работы программы.

Для 32-разрядных приложений шаблон исходного текста выглядит иначе:

```
.model flat
.stack
.data
; äàííûå
.code
main:
. . .
; команды ассемблера
. . .
end main
end
```

Основное отличие от предыдущего примера — другая модель памяти (`flat`), предполагающая 32-разрядную линейную адресацию с атрибутом `near`.

Как видно из примера, “классический” шаблон 32-разрядного приложения содержит область данных (определяемую директивой `.data`), область стека (директива `.stack`) и область программного кода (директива `.code`). Может случиться так, что 32-разрядному приложению на ассемблере потребуется несколько отдельных сегментов данных и/или кода. В этом случае разработчик может создать их с помощью директивы `SEGMENT`. Директива `SEGMENT` определяет логический сегмент и может быть описана следующим образом:

```
имя SEGMENT список атрибутов
. . .
имя ENDS
```

Замечу, что директива `SEGMENT` может применяться с любой моделью памяти, не только `flat`. При использовании директивы `SEGMENT` потребуется указать компилятору на то, что все сегментные регистры устанавливаются в соответствии с моделью памяти `flat`. Это можно сделать при помощи директивы `ASSUME`:

```
ASSUME CS:FLAT, DS:FLAT, SS:FLAT, ES:FLAT, FS:ERROR, GS:ERROR
```

Регистры `FS` и `GS` программами не используются, поэтому для них указывается атрибут `ERROR`.

Сейчас мы рассмотрим программный код 32-разрядной процедуры на ассемблере (она называется `_seg_ex`), в которой используются два логических сегмента данных. Процедура выполняет копирование строки `src`, находящейся в сегменте данных `data1`, в область памяти `dst` в сегменте данных `data2` и содержит один логический сегмент программного кода (`code segment`).

Успокою читателей, незнакомых с принципами работы процедур (они рассмотрены далее в книге): в данном случае нас будет интересовать код внутри процедуры `_seg_ex` (команды, находящиеся между директивами `_seg_ex proc` и `_seg_ex endp`). Исходный текст программного кода процедуры `_seg_ex` представлен в листинге 4.2.

```
.586
.model flat
 option casemap:none
data1 segment
  src  DB   "Test STRING To Copy"
  len  EQU  $-src
data1  ends
data2  segment public
  dst  DB  len+1 DUP('+')
data2  ends
code segment
_seg_ex proc
 assume CS:FLAT,DS:FLAT, SS:FLAT, ES:FLAT, FS:ERROR, GS:ERROR
 mov    ESI, offset data1
 mov    EDI, offset data2
 cld
 mov    CX, len
 rep    movsb
 mov    EAX, offset data2
 ret
_seg_ex endp
code ends
end
```

Листинг 4.2. *Использование двух логических сегментов данных в 32-разрядной процедуре*

При использовании модели `flat` доступ к данным осуществляется по 32-разрядному смещению, поэтому смысл показанных ниже команд, загружающих адреса логических сегментов (а заодно и адреса строк `src` и `dst`) в регистры `ESI` и `EDI`, думаю, понятен:

```
mov ESI, offset data1
mov EDI, offset data2
```

Группа следующих команд выполняет копирование строки `src` в `dst`, при этом регистр `CX` содержит количество копируемых байтов:

```
cld
mov CX, len
rep movsb
```

В регистре `EAX` возвращается адрес строки-приемника `dst`. Обращаю внимание читателей на то, что никакой инициализации сегментных регистров не требуется, поскольку это делается при помощи директивы `.model flat`. Еще один важный момент: программа, использующая модель `flat`, выполняется в одном физическом сегменте, хотя логических сегментов может быть несколько, как в нашем случае.

Работоспособность процедуры легко проверить, вызвав ее из программы на Visual C++ .NET (нужно только включить объектный файл процедуры в проект приложения). Исходный текст приложения приведен в листинге 4.3.

```
#include <stdio.h>
extern "C" char* seg_ex(void);
int main(void)
{
 printf("EXTERNAL MODULE EXAMPLE: %s\n", seg_ex());
 return 0;
}
```

Листинг 4.3. *Программа, вызывающая процедуру seg_ex*

Здесь процедура `seg_ex` является внешней, поэтому объявлена как `extern`.

Результатом выполнения программы будет строка на экране дисплея

```
EXTERNAL MODULE EXAMPLE: Test STRING To Copy+
```