

МИНЕСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ  
федеральное государственное бюджетное образовательное учреждение  
высшего профессионального образования  
«УЛЬЯНОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

**В. Н. НЕГОДА**

# **МАШИННО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ**

**Учебное пособие**

для студентов, обучающихся по направлению 09.03.01  
«Информатика и вычислительная техника»

Ульяновск  
УлГТУ  
2015

УДК 004.43 (075)  
ББК 32.973.26 – 018.1 я7  
Н 41

Рецензенты:

доктор технических наук, профессор, главный научный  
сотрудник ФНПЦ АО «НПО «Марс» Ю. П. Егоров;

кандидат технических наук, начальник НИО-21 ФНПЦ АО  
«НПО «Марс» П. И. Смикун.

Утверждено редакционно-издательским советом университета  
в качестве учебного пособия

**Негода, Виктор Николаевич**

Н 41 Машинно-ориентированное программирование : учебное пособие  
/ В. Н. Негода. – Ульяновск : УлГТУ, 2015. – 160 с.

ISBN 978-5-9795-1522-9

Излагаются основы машинно-ориентированного программирования на языке Ассемблер с активным использованием аналогичных управляющих конструкций и структур данных языка Си.

Пособие предназначено для студентов, изучающих дисциплину «Машинно-ориентированное программирование» в рамках учебных планов подготовки бакалавров направления 09.03.01 «Информатика и вычислительная техника».

**УДК 04.43 (075)**  
**ББК 32.973.26 – 018.1 я7**

ISBN 978-5-9795-1522-9

© Негода, В. Н., 2015  
Оформление. УлГТУ, 2015

## Оглавление

Введение.....	5
1. Основы технологии программирования на ассемблере .....	8
1.1. Введение в машинно-ориентированное программирование.....	8
1.1.1 Базовые определения.....	8
1.1.2 Машинно-ориентированные языки и их применение.....	12
1.2. Базовые процессы создания программ средствами ассемблера .....	15
1.3. Инструментальные средства программирования на ассемблере....	26
1.3.1 Ассемблер-трансляторы и компоновщики.....	26
1.3.2 Средства отладки ассемблер-программ.....	38
1.3.3 Интегрированная среда RadAsm .....	45
1.4. Вопросы и упражнения .....	56
2. Основные элементы языка ассемблера для архитектур x86 .....	58
2.1. Структура ассемблер-программ .....	58
2.2. Синтаксис оператора .....	60
2.3. Константы и выражения .....	61
2.4. Декларация данных .....	63
2.5. Элементы макропрограммирования .....	65
2.5.1 Директивы условной трансляции.....	66
2.5.2 Макроопределения и макрорасширения .....	68
2.5.3 Использование меток и объявлений данных в макросах.....	71
2.5.4 Макросы организации записей.....	74
2.6. Вопросы и упражнения .....	76
3. Функциональная организация процессоров с архитектурой Intel x86....	80
3.1. Программно-доступные компоненты и методы адресации .....	80
3.2. Система команд.....	86
3.2.1 Соглашения по описанию системы команд .....	86
3.2.2 Организация передачи управления .....	87
3.2.3 Пересылка данных .....	90
3.2.4 Команды арифметической обработки .....	92
3.2.5 Команды логических операций .....	93
3.2.6 Команды сдвигов .....	93
3.2.7 Команды обработки бит .....	94
3.2.8 Префиксы команд .....	94
3.2.9 Команды поддержки обработки массивов .....	95
3.3. Вопросы и упражнения .....	96
4. Реализация управления обработкой данных на ассемблере .....	98
4.1. Базовые понятия и соглашения .....	98
4.2. Программирование ветвлений.....	100

4.2.1	Ветвление по условиям .....	100
4.2.2	Реализация оператора switch .....	106
4.2.3	Программирование циклов .....	111
4.3.	Организация подпрограмм .....	118
4.3.1	Базовые механизмы и операторы .....	118
4.3.2	Передача параметров через регистры .....	123
4.3.3	Передача параметров через стек .....	125
4.3.4	Передача параметров через статические данные .....	130
4.3.5	Организация локальных переменных .....	131
4.3.6	Рекурсивные подпрограммы .....	132
4.4.	Вопросы и упражнения .....	136
5.	Организация ввода-вывода .....	138
5.1.	Базовые средства ввода-вывода .....	138
5.2.	Ввод целочисленных данных и преобразование из внешнего представления во внутреннее .....	143
5.3.	Вывод целочисленных данных и преобразование из внутреннего представления во внешнее .....	147
5.4.	Вопросы и упражнения .....	151
6.	Обработка массивов и адресная арифметика .....	153
6.1.	Объявление массивов и последовательный доступ к их элементам через адресные переменные .....	153
6.2.	Произвольный доступ к элементам массивов .....	154
6.3.	Организация обработки текстов .....	157
6.4.	Вопросы и упражнения .....	158
	Библиографический список .....	160

## Введение

За последние три десятилетия использование машинно-ориентированных языков в разработке программного обеспечения существенно сократилось. Языки ассемблеров перестали доминировать в разработке системного программного обеспечения в восьмидесятых годах прошлого столетия, а в разработке программ микропроцессорных систем – в девяностых. Сначала ассемблеры были потеснены языком Си, а в последние двадцать лет наблюдается расширение применения языка C++ для создания таких систем и приложений, где требуется учет особенностей целевой компьютерной архитектуры.

В то же время многообразие издаваемой научно-технической литературы по программированию на ассемблерах за последние годы не уменьшается. Вы можете убедиться в этом, запустив поиск книг на сайте какого-нибудь электронного книжного магазина. Например, поиск в конце сентября 2014 года по фразе «*assembler language*» на сайте *amazon.com* дал 1465 ссылок, а поиск по фразе «язык ассемблера» на сайте *ozon.ru* дал 65 ссылок. Примечательным фактом является также сохранение приверженности к машинно-ориентированному языку автора знаменитого многотомника «Искусство программирования» Дональда Кнута. Первые тома Дональд Кнут издавал в семидесятых годах и использовал в примерах программ мнемокод гипотетической машины MIX. Кстати, эти тома по версии известного журнала *American Scientist* вошли в список 12 самых знаменитых книг 20-го века по разделу физико-математической литературы. Уже в 21-м веке Дональд Кнут издает продолжение многотомника, где использует более совершенную гипотетическую машину MMIX, активно иллюстрируя обсуждаемые комбинаторные алгоритмы реализациями на мнемокоде этой машины.

Основными потребителями литературы по машинно-ориентированным языкам являются две категории людей. Во-первых, это студенты, которые вовлекаются преподавателями в программирование на ассемблере в учебных дисциплинах «Архитектура ЭВМ»,

«Программирование на языке Ассемблера», «Системное программирование», «Микропроцессорные системы» и др. Во-вторых, профессиональные разработчики системного программного обеспечения и систем на базе микроконтроллеров.

Возникает закономерный вопрос: «Почему при существенном сокращении применения ассемблеров в практическом программировании машинно-ориентированные языки занимают заметное место в обучении программированию и в научно-технической литературе?». Достаточно убедительным будет ответ: «Потому, что существует довольно много видов программистской деятельности, в которых требуется понимать, что происходит в среде компьютера во время исполнения программы, а также обеспечивать эффективность реализации программных функций». Перечень основных из этих видов деятельности можно найти в подразделе 1.1 данного пособия.

Дисциплина «Машинно-ориентированное программирование» преподается на кафедре вычислительной техники Ульяновского государственного технического университета в рамках плана подготовки бакалавров и инженеров направления «Информатика и вычислительная техника» с 1996 года. В этой дисциплине изучаются не только ассемблер, но и программно-технические решения для языков высокого уровня (ЯВУ), повышающие эффективность создаваемых программ, а также вопросы программного моделирования средств вычислительной техники.

В данном учебно-методическом пособии в компактной форме излагаются основные идеи, методы и программно-технические решения, связанные с машинно-ориентированным программированием. В качестве ЯВУ, используемого наряду с ассемблером для изложения методов машинно-ориентированного программирования, в пособии используется язык Си/С++. Такой выбор связан с тем, что именно Си и С++ чаще всего используются для программирования аппаратно-зависимых частей операционных систем, разработки программных функций управления оборудованием цифровых систем управления, программирования

высокопроизводительных приложений, где очень важно учитывать особенности среды исполнения, и именно Си и С++ являются основными языками программного моделирования средств вычислительной техники.

Большая часть учебной литературы по программированию на ассемблере относится к ассемблерам для семейства компьютерных архитектур Intel x86. Это объясняется тем, что в распоряжении практически каждого студента, изучающего программирование, имеется компьютер, в котором реализованы две-три архитектуры данного семейства. Такого же типа компьютеры доминируют в компьютерных классах учебных заведений. Доступ к другим компьютерным архитектурам у студентов существенно ниже.

В семейство архитектур Intel x86 входят:

- а) 16-разрядная архитектура IA-16 (Intel Architecture, 16 bit);
- б) 32-разрядная архитектура IA-32;
- в) 64-разрядное расширение архитектуры IA-32, называемое по-разному – AMD64, x86-64, EM64T.

На самом деле каждая из этих архитектур пережила в своем развитии несколько стадий развития, в ходе которых к базовой системе команд добавлялись новые, расширялись возможности адресации данных и управления доступом к памяти, развивались механизмы управления машиной. Это порождает ситуации программной несовместимости, однако для программ, создаваемых в учебном процессе, такое случается очень редко.

При чтении пособия рекомендуется сначала просто сориентироваться в содержании фактологического материала, которым изобилует третья глава. А затем, по мере изучения основ машинно-ориентированного программирования, использовать этот материал для понимания и выполнения упражнений.

В конце глав пособия приводятся вопросы и упражнения, работа над которыми может потребовать сведений, не приводимых в самом пособии. Для получения таких сведений достаточно воспользоваться поиском в Internet.

# 1. Основы технологии программирования на ассемблере

## 1.1. Введение в машинно-ориентированное программирование

### 1.1.1 Базовые определения

*Машинно-ориентированным программированием* является любой процесс разработки программы, в котором учитываются свойства компьютера или отдельных его устройств, блоков и узлов. К основным видам машинно-ориентированного программирования относятся:

1) Программирование на языке ассемблера. Термином *ассемблер* называют транслятор с языка ассемблера, а часто и сам язык. Главной особенностью языка является использование в качестве исполняемых операторов мнемокодов машинных команд. Иначе говоря, программирование на ассемблере является программированием в машинных командах целевой архитектуры, представленных мнемокодами (псевдокодами), а не машинными кодами.

#### Пример 1.1. Реализация на ассемблере простого выражения языка Си

Выражение  $z = 2 * (z + 34)$  для случая, когда  $z$  имеет тип `long`, на ассемблере для IA-32 может быть представлено так:

```
add    dword ptr z, 34    ; прибавить 34 к z
shl     z, 1              ; умножить на 2 через сдвиг влево
```

*Целевая архитектура* – это архитектура семейства компьютеров, на которых могут выполняться машинные команды, полученные в результате трансляции либо ассемблер-программы, либо программы на ЯВУ. Поскольку существует много компьютерных архитектур, существует много языков ассемблера. Более того, для одной и той же архитектуры может иметь место несколько языков ассемблера, отличающихся по синтаксису операторов.



*Машинная программа* – это совокупность машинных команд и наборов данных, представленных в формате машинных слов, которая может быть выполнена либо в реальном компьютере, либо в среде симулятора целевой архитектуры. Различают внутреннее и внешнее представления машинных программ. Во внутреннем представлении каждая команда – это набор битовых полей (в частном случае – байт-код). Во внешнем представлении это обычно шестнадцатеричная или восьмеричная символьная запись машинных кодов, используемая для восприятия человеком. Например, машинный код приведенного в примере 1.1 фрагмента ассемблер-программы при размещении *z* по адресу 100h (это шестнадцатеричное число) имеет вид:

```
83 05 00000100 22
D1 25 00000100
```

Здесь первые байты в каждой строке задают код операции, вторые байты – метод адресации. Байты 4-7 содержат 32-разрядный адрес 100h. Десятичная константа 34 в первой команде в шестнадцатеричном виде представляется как 22.

*Точка программы* – обычно это позиция перед определенной машинной командой. Чаще всего понятие точки программы используется в контексте перехода к какому-то оператору по командам ветвления, вызова подпрограммы или возврата из нее.

*Симулятор* компьютерной архитектуры или конкретного компьютера – это программная модель, которая может выполняться даже в среде иной компьютерной архитектуры. В настоящее время активно используются симуляторы микроконтроллеров с архитектурой ARM, включаемые в состав средств программирования смартфонов, планшетов, промышленных контроллеров и др. Сами средства программирования при этом работают обычно на компьютере с архитектурой из семейства Intel x86, однако в фазе исполнения разработанных программ симулятор интерпретирует машинные коды ARM как реальный микроконтроллер.

2) Такое программирование на ЯВУ, в ходе которого программист задается вопросом «В какие машинные команды и какие машинные типы

данных будет транслироваться программа». Это бывает нужным для выбора программно-технических решений, которые либо обеспечат требуемые быстродействие и затраты памяти, либо защитят программу от взлома.

3) Программирование тех частей операционной системы, которые связаны с управлением памятью компьютера, процессами и оборудованием. Некоторые программные функции ядра операционной системы пишутся либо на ассемблере, либо на смеси ассемблера с ЯВУ.

4) Разработка программ ассемблирования, т. е. компиляторов языков ассемблеров.

5) Разработка интерпретаторов языка ассемблера. Такие интерпретаторы используются обычно в учебном процессе.

6) Программирование фазы кодогенерации компилятора. В этой фазе компилятор порождает машинные команды, что как раз и обуславливает машинную ориентацию указанного вида программирования, несмотря на то, что компиляторы обычно пишутся на ЯВУ.

7) Разработка бинарных трансляторов. Бинарный транслятор переводит машинный код одной целевой архитектуры в машинный код другой. К бинарным трансляторам относятся, например, JIT-компиляторы Java Virtual Machine (JVM). Эти компиляторы транслируют байт-код архитектуры JVM в байт-код архитектуры компьютера, в котором выполняется JVM.

8) Программирование оптимизатора компилятора. Качество оптимизатора во многом определяется тем, насколько глубоко разобрались его разработчики в целевой компьютерной архитектуре. В этом виде программирования разработчик должен находить такие последовательности машинных команд, которые наиболее быстро или экономно по затратам памяти реализуют операторы или группы операторов исходной программы.

9) Программирование компоновщика объектных модулей. Компоновщик должен объединять сегменты машинных программ и данных и модифицировать адресные поля машинных команд объектных

модулей таким образом, чтобы продукты независимой трансляции нескольких исходных модулей могли быть связаны в целостную машинную программу.

10) Программирование дизассемблера. *Дизассемблер* – это программа, которая преобразует код машинной программы в мнемокод ассемблера. Естественно, что разработчик дизассемблера должен очень хорошо знать функциональную организацию целевой архитектуры.

11) Программирование отладчиков машинных программ. Отладчик может трансформировать машинную программу в точках останова и фазах обработки прерываний, может следить за состояниями ячеек памяти, поддерживать покомандную трассировку машинной программы с использованием либо встроенного дизассемблера, либо исходного кода ассемблер-программы. Некоторые отладчики имеют в своем составе встроенные компиляторы ассемблер-команд.

12) Программирование профилировщиков машинных программ. *Профилировщик* – это программа, которая формирует статистические данные (так называемый профиль исполнения программы) о ходе выполнения программы. В профиле фиксируются затраты времени на исполнение программных функций, частота обращения к подпрограммам, частота перехода по тем или иным ветвям в операторах условного перехода и т. п.

13) Программирование симуляторов компьютерных архитектур. Симуляторы используются в качестве средств автоматизации проектирования компьютерных систем и средств эксплуатации программ, созданных для иной компьютерной архитектуры, нежели та, в которой программа запускается на исполнение.

14) Разработка программных моделей устройств вычислительной техники. Программные модели используются либо как средство автоматизации проектирования средств вычислительной техники, либо как средство автоматизации обучения, позволяющее «заглянуть» в такие детали поведения устройств, которые невозможно наблюдать даже при

наличии дорогостоящей измерительной аппаратуры (например, внутренние процессы интегральной схемы).

15) Модификация машинных программ в условиях, когда исходный код недоступен. С помощью такой модификации исправляют ошибки программ, обеспечивают сопряжение старых программ с новым оборудованием, на которое они не были рассчитаны, модифицируют объектные библиотеки с целью оптимизации и т. п.

16) Разработка анализаторов машинного кода или ассемблер-программ для оценки затрат времени, получения статистики частоты использования тех или иных машинных команд и т. п. Процесс такого анализа часто называют статическим профилированием.

17) Разработка программ определения свойств компьютера – состава и параметров его устройств, быстродействия процессоров.

18) Обратное проектирование – восстановление проектных решений в условиях, когда их спецификации, включая исходные тексты, отсутствуют, а функциональность программы либо нужно изменить, либо повторить для другой аппаратной платформы. Иногда приходится разрабатывать на ЯВУ программу, которая буквально воспроизводит поведение машинной программы, подвергаемой обратному проектированию. В этом случае задача может быть сведена к получению такого продукта трансляции создаваемого исходного кода, который полностью совпадает с исходной машинной программой.

Успешное осуществление любого из вышеперечисленных видов программирования требует разнообразных знаний, умений и навыков, значительная часть которых приобретается в ходе программирования на ассемблере.

### **1.1.2 Машинно-ориентированные языки и их применение**

Основными машинно-ориентированными языками являются языки ассемблеров или, как часто их называют, ассемблеры. Транслятор с такого языка также называют ассемблером. Когда говорят «ассемблер x86», то могут подразумевать как язык машинно-ориентированного

программирования компьютеров с архитектурой Intel x86, так и сам компилятор с этого языка. Чтобы разобраться, что имеется в виду, нужно чувствовать контекст. Ассемблер архитектуры x86 совершенно не пригоден для разработки программ с иной целевой архитектурой, например, микроконтроллеров ARM. Ассемблер ARM также не может транслировать программы, предназначенные для x86.

Для компьютеров семейства архитектур Intel x86 существует довольно много ассемблер-трансляторов, основными из которых являются: MASM, TASM, NASM, FASM, YASM, WASM, HLA. Большинство учебной и научно-технической литературы по программированию на ассемблере базируется на применении MASM. В данном учебном пособии использование MASM мотивировано еще и тем, что именно синтаксис и трансляторы MASM используются для смешанного программирования на ЯВУ и ассемблере в широко распространенных средах Visual Studio Express / Community. Эти среды представляют собой линейку бесплатных интегрированных сред разработки, каждая из которых создана компанией Microsoft в качестве облегченной версии Microsoft Visual Studio. Согласно утверждению Microsoft, «Express»-редакции предлагают отлаженную, простую в обучении и использовании среду разработки пользователям, не являющимся профессиональными разработчиками ПО — любителям и студентам.

Естественно считать машинно-ориентированным языком язык машинных кодов. Чаще всего машинные коды пишутся в шестнадцатеричной системе счисления (HEX-код), но для некоторых архитектур используется восьмеричная система. Использование машинных кодов выгодно в случаях, когда необходимо что-то изменить непосредственно в машинной программе, исходные коды для которой отсутствуют.

Машинные коды иногда приходится использовать при программировании на ассемблере. Это бывает, когда ассемблер не поддерживает трансляцию мнемокодов некоторых команд,

поддерживаемых целевой архитектурой. Например, в случае, когда мы имеем самый новый процессор, а используемый ассемблер прекратил свое развитие уже несколько лет назад. Тогда для использования новых машинных команд приходится в последовательность операторов ассемблера вставлять декларации размещения данных, через которые кодируются новые команды.

В ЯВУ часто имеются операторы вставки ассемблерных строк (это называется встроенным ассемблером) и даже машинных кодов. В некоторые ЯВУ встраивают расширения, позволяющие напрямую работать с регистрами и портами ввода-вывода компьютера. Чаще всего это происходит с языком Си или C++, поскольку именно в этих языках хорошо поддерживается обработка адресов (адресная арифметика), задаваемое программистом размещение данных в регистрах, и набор операций присваивания сконструирован буквально на основе наборов двухадресных команд обработки. Проиллюстрируем это примерами.

### **Пример 1.2. Реализация операторов присваивания языка Си**

Пусть в Си-процедуре фигурирует объявление:

```
register long n1, n2, *p1, *p2;
```

Допустим, что Си-компилятор целевой архитектуры IA-32 выделил для указанных переменных такие регистры: n1 – ECX, n2 – EBX, p1 – ESI, p2 – EDI. Благодаря наличию у Си свойств машинной ориентации, мы получим следующие очень простые реализации операторов присваивания:

```
n1 += 5 ;           // add ECX, 5
n2 <<= 3;           // shl EDX, 3
n1 |= n2;           // or  ECX, EDX
*p1 -= n1;          // sub [ESI], ECX
n2 &= p2[3];        // and EDX, [EDI + 12]
```

### **Пример 1.3. Копирование массивов с помощью ассемблерной вставки**

Пусть Си-программист, работающий в Visual Studio, знает о том, что быстрое копирование 32-разрядных машинных слов из одной области памяти в другую в IA-32 можно выполнить строковой командой MOVSD с префиксом повторения REP при условии, что в ESI находится адрес исходной области,

в EDI – адрес результирующей, а в ECX – число копируемых машинных слов. Тогда реализацию быстрого копирования массива m1 данных типа long в массив m2 может быть выполнено такой ассемблерной вставкой:

```
__asm
{
    LEA ESI, m1
    LEA EDI, m2
    MOV ECX, 4
    CLD
    REP MOVSD
}
```

Транслятор ассемблера, который подключается для компиляции ассемблер-вставок, не различает регистры символов, использованных для обозначения имен регистров и ассемблер-операторов. Это означает, что приведенный фрагмент может быть полностью написан строчными буквами. Однако по отношению к именам m1 и m2 такой произвол недопустим, поскольку в Си M1 и m1 – это разные идентификаторы.

## **1.2. Базовые процессы создания программ средствами ассемблера**

Процесс разработки ассемблер-программы представлен на рис. 1.1. Этот процесс имеет циклический характер и подобен процессу программирования на любом ЯВУ, предполагающем отдельную компиляцию многомодульной программы.

Вначале формируется один или несколько исходных модулей. При этом используют либо обычные редакторы текста, либо синтаксически-ориентированные (расцветка ключевых слов, отступы и т. п.).

В программировании на ассемблере используются несколько типов исходных модулей, каждый из которых определяется расширением имени файла.

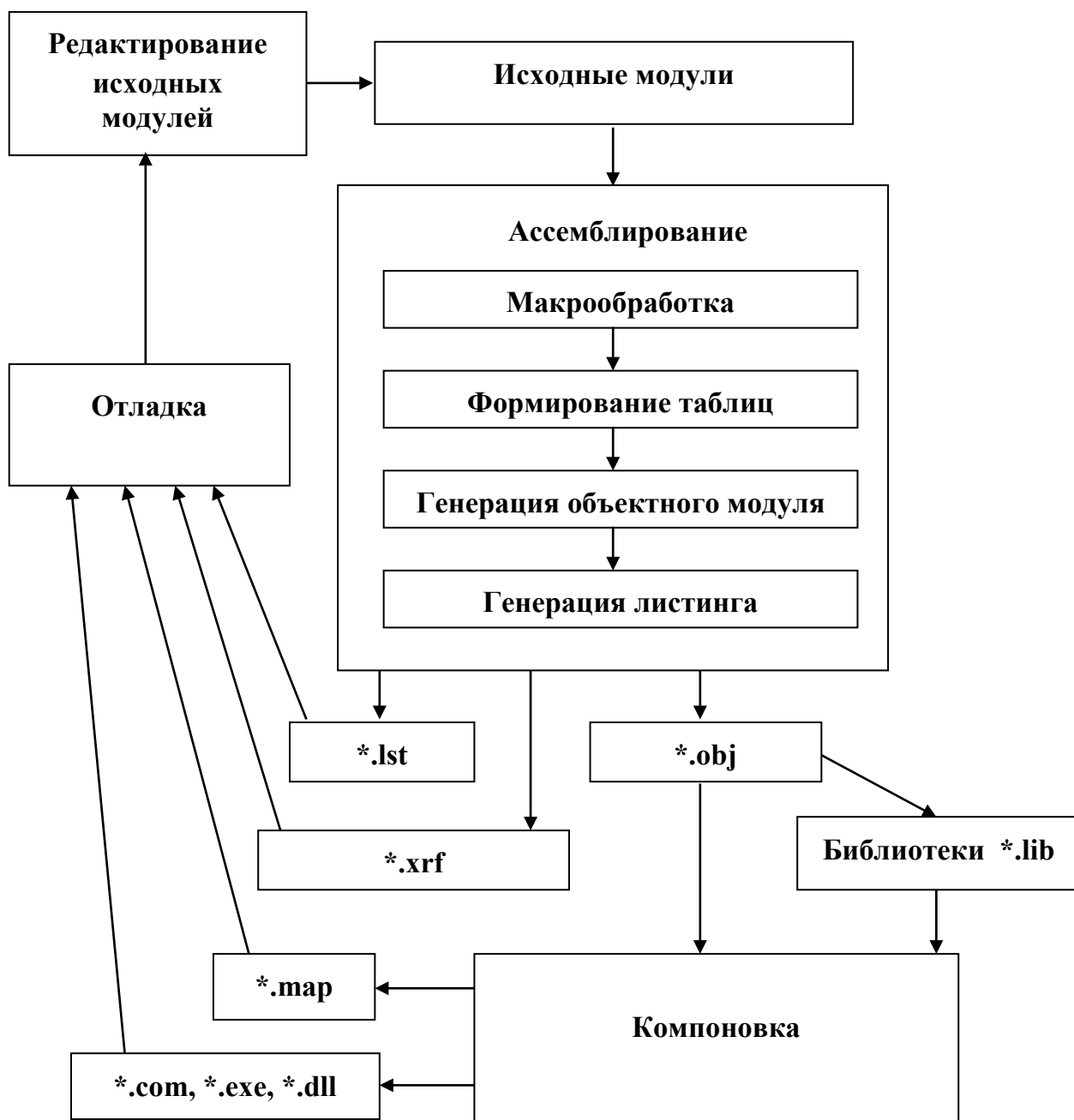


Рис. 1.1. Процесс разработки ассемблер-программы

Основные типы исходных модулей задаются следующими расширениями:

.asm — файл ассемблер-программы;

.inc – файл с операторами определения общих имен, прежде всего именованных констант, а также определениями макрокоманд; в некоторых системах программирования на ассемблере макроопределения концентрируют в файле с расширением .mac;



\*.def и \*.rc – файлы определений и ресурсные файлы, используемые в системах программирования на ЯВУ и поддерживаемые средствами макрообработки ассемблера для обеспечения совместного использования ассемблера и ЯВУ.

Исходный модуль ассемблер программы (файл с расширением .asm) транслируется ассемблером в несколько проходов. Один проход заключается в сканировании текста программы от начала до конца. Многопроходность обусловлена тем, что операторы могут содержать имена, определение которых пока еще не встретились обработчику в ходе сканирования текста программы.

На первом проходе выполняется макрообработка. На этом этапе исходный asm-модуль преобразуется во внутреннее представление в соответствии с содержанием таких операторных строк, которые содержат директивы поддержки макропрограммирования. Такие директивы называются макросами. Самым распространенным макросом является объявление имен констант. Пусть, например, во многих местах программы нужно обрабатывать строки длиной в 80 байт. Очевидно, что целесообразно константу 80 целесообразно заименовать, например, именем LenStr. Это улучшает семантическое восприятие исходного текста и создает условия для простого и безошибочного изменения соглашения о длине строк.

#### **Пример 1.4. Пример макрообработки, использующей директиву invoke**

Наиболее часто вызов функций Windows API в ассемблере базируется на использовании встроенной в ассемблер директивы «invoke». Эта директива позволяет записывать вызов функций, не программируя явным образом передачу параметров через стек. Типичный вариант вывода на консоль строки msg длиной 20 байт при использовании этой директивы имеет вид (символ “\” задает продолжение операторно строки):

```
invoke WriteConsoleA, hStdout, ADDR msg,\
20, ADDR cWritten, 0
```

Этот оператор эквивалентен Си-функции:

```
WriteConsoleA(hstdout, msg, 20,&cWritten,0);
```

Если программировать вызов этой функции без использования директивы `invoke`, то потребуется следующий фрагмент ассемблер-программы:

```
push 0
push offset cWritten
push 20
push offset msg
push hStdout
call WriteConsoleA
```

Именно такой фрагмент и будет порожден в ходе макрообработки строки с директивой `invoke WriteConsoleA`.

Пользователь может определить свою собственную директиву макровывода. Для этого в ассемблере имеется директива `MACRO`, использование которой будет рассмотрено в подразделе 2.5.

Текст ассемблер-программы, полученной в результате макрообработки, подвергается анализу с целью формирования таблиц имен. Как и в программе на ЯВУ в ассемблерах для адресации элементов данных и точек программ принято использовать имена, причем очень часто использование имени в программе фигурирует раньше его определения. В этой связи формирование таблицы имен выполняется в отдельном проходе по тексту программы.

Программа обычно имеет несколько сегментов. Наиболее часто это сегмент данных и сегмент кода, где находятся машинные команды. В ходе сканирования текста ассемблер для каждого сегмента программы ведет так называемый счетчик размещения. В начале любого сегмента соответствующий счетчик обнуляется. По мере сканирования текста ассемблер вычисляет размер области памяти данных или кода, который будет занят в ходе кодогенерации, и продвигает счетчик размещения так, чтобы он начал указывать на ту область памяти, в которой должен будет размещен продукт кодогенерации следующего оператора. Если оператор помечен меткой, то текущее значение счетчика размещения используется как значение метки, получаемое через механизм ведения счетчика размещения, является смещением от начала сегмента в общей памяти до местоположения помеченной меткой области памяти.

Рассмотрим это на примере.

### Пример 1.5. Распечатка трехстрочного текста

Пусть требуется написать такие фрагменты ассемблер-программы распечатки массива строк, которые соответствуют следующим двум фрагментам Си-программы:

```
/* Объявление массива строк, именуемого txt */
char *txt[] = {"Первая строка",
               "Вторая строка", "Третья строка", 0};
```

...

```
/* Цикл распечатки массива строк */
for (char **p = txt; *p != NULL; p++) puts(*p);
```

Нужно иметь в виду, что переменная `txt` в приведенном фрагменте именуется область памяти одномерного массива адресов строк. То есть каждый элемент массива `txt` является адресом первого символа соответствующей строки. Учитывая это обстоятельство, на ассемблере архитектуры IA-32 эти два фрагмента могут быть реализованы следующим образом:

```
; Для размещения данных с помощью директивы .data включаем
; ведение соответствующего счетчика размещения
.data
; Размещаем 4 элемента массива адресов txt, именуя строки.
; Размещаем массив txt адресов строк с помощью директивы dd,
; поскольку адреса 4-байтные
txt      dd s1,s2,s3,0;
; Размещаем 3 строки, используя директиву размещения байтов db,
; не забывая добавлять байт 0, который ограничивает Си-строки
s1       db 'Первая строка',0
s2       db 'Вторая строка',0
s3       db 'Третья строка',0
; Для размещения команд цикла распечатки включаем ведение
; счетчика размещения сегмента кода директивой .code
.code
; регистр esi будет указателем p
mov esi, offset txt ; p = **txt
; Метим именем @For начало тела цикла для перехода с целью повтора
@For:
; Сначала проверяем, не достигнут ли конец массива
```

```

    cmp dword ptr [esi], 0
    je  @AfterFor ; выход при обнаружении NULL
    push    [esi] ; Передаем подпрограмме puts адрес строки через
стек
    call    puts ; Вызываем процедуру
    add esi, 4 ; r++ для движения по массиву
    jmp @For
@AfterFor:

```

В приведенном фрагменте дважды встречаются ссылки вперед: в инициализации элементов массива `txt` адресами `s1`, `s2`, `s3` и в переходе по метке `@AfterFor` для выхода из цикла. В первом случае ассемблер сразу не сможет заполнить машинные слова массива `txt` адресами строк, поскольку он еще не знает этих адресов. Во втором случае ассемблер не сможет сформировать адресную часть машинной команды условного перехода по равенству `je`, поскольку местоположения точки, именуемой меткой `@AfterFor`, ассемблер также еще не знает. А для вычисления адресов, по которым размещаются помеченные метками операторы, у ассемблера все есть.

Пусть, например, в точке появления директивы `.data` соответствующий счетчик размещения, назовем его `DataDot`, был равен 1000. Тогда при встрече метки `txt`, которая помечает место размещения массива адресов, ассемблер занесет в таблицу имен следующую информацию:

- а) само имя `txt`;
- б) тип отмеченного меткой элемента данных `dd`, означающее четырехбайтное слово;
- в) адрес размещения `DataDot`, равный 1000, который будет использоваться в следующем проходе ассемблера при формировании адресной части оператора `"mov esi, offset txt"`.

После формирования элемента таблицы имен для метки `txt` ассемблер посчитает длину данного элемента. Для этого достаточно пересчитать аргументы директивы размещения четырехбайтных слов `dd`.

Поскольку имеется 4 элемента, общий размер памяти, в которой будет размещен транслируемый оператор, равен 16 байт. И это означает, что счетчик размещения `DataDot` должен быть увеличен на 16, что дает новое значение `DataDot = 1016`. Сам размер 16 также размещается в таблице.

Во второй строке фрагмента ассемблер-программы из примера 1.5 ассемблер встречается метку `s1`, формирует информацию о ней и помещает в таблицу имен. Очевидно, что в качестве адреса размещения, т. е. значения имени `s1`, выступает число 1016. Именно это число будет вставлено в память сегмента данных в следующем проходе в качестве первого адреса массива `txt` по адресу 1000. Определяя размер участка памяти, помеченного именем `s1`, ассемблер будет считать символы между кавычками, выделяющими константную строку, а затем прибавлять еще единицу, учитывающую байт 0, используемый как признак конца строки. Результатом подсчета будет число 14, что благодаря приращению `DataDot += 14`, обеспечивает формирование адреса `s2 = 1020`.

Как мы видим, процесс формирования таблиц для сегмента данных является достаточно простым. Также простым кажется процесс использования таблицы имен в следующем проходе. Очевидно, что встретив на этапе кодогенерации имя `s1` в строке с меткой `txt`, ассемблер легко найдет информацию об этом имени в таблице и сможет скопировать значение имени 1016 в машинное слово первого элемента массива `txt`. Простота процесса наверняка наталкивает некоторых любителей программирования на мысль: «А не написать ли мне свой собственный ассемблер?». Именно этим обстоятельством можно объяснить тот факт, что довольно многие ассемблер-компиляторы разработаны не какими-то крупными фирмами, а либо программистами-одиночками, либо небольшой группой программистов любителей.

Встретив директиву `.code`, ассемблер переключится на ведение уже другого счетчика размещения. Назовем этот счетчик именем `CodeDot`. Пусть текущее значение этого счетчика в момент переключения равно 500. В принципе, он может быть также равным 1000, поскольку разбиение

программы на сегменты позволяет с помощью нескольких счетчиков размещения транслировать строки сегментов в соответствующие области памяти независимо друг от друга. В ходе анализа первой строки после директивы `.code` ассемблер выполняет более сложную работу, нежели при анализе параметров директив размещения данных. В ходе анализа оператора `"mov esi, offset txt"` ассемблер найдет в таблице предопределенных имен ассемблера имя `mov`, затем, обработав параметры оператора `mov`, определит формат машинной команды и посчитает ее размер. Определив размер, который будет равным 6 байтов, ассемблер продвинет `CodeDot` к значению 506. Именно это значение будет занесено в спецификацию метки `@For` таблицы имен. Кроме адреса и имени в таблицу будет занесен также тип имени – адрес точки программы. В следующем проходе значение 506 используется для формирования адресной части машинной команды, полученной в результате трансляции оператора `"jmp @For"`. Во время прохода формирования таблицы имен определяется только размер памяти продукта трансляции этого оператора, чтобы получить счетчик размещения для метки `@AfterFor`.

На этапе генерации объектного кода формируются машинные команды. Казалось бы, наличие таблицы имен обеспечивает формирование законченной машинной программы, однако объектный код является некоторым полуфабрикатом, который не может быть загружен операционной системой в память и запущен на выполнение. Это связано с необходимостью поддержки раздельной компиляции многомодульных программ.

Пусть, например, рассмотренный в примере 1.5 фрагмент находится в модуле `m1.asm`, а подпрограмма `puts` находится в модуле `m2.asm`. Это же логично, иметь набор подпрограмм в виде готового объектного модуля или библиотеки объектных модулей и использовать их в различных программах. Именно так используются функции из наборов подпрограмм Си/C++ `stdlib`, `stdio`, `string` и т. п.

Раздельная компиляция означает, что сегменты кода и сегменты данных в разных модулях будут адресоваться от нулевых значений соответствующих счетчиков размещения. Во время трансляции модуля `m1.asm` ассемблер ничего не знает о модуле `m2.asm`. Это приводит к тому, что адресное поле машинной команды `call puts` не может быть заполнено в фазе кодогенерации модуля `m1`. В принципе, модуль `m2.asm` может содержать фрагменты обработки строк массива `txt`. Тогда при трансляции модуля `m2.asm` неопределенным окажется имя `txt`. Если, например, самый первый исполняемый оператор сегмента кода модуля `m1.asm` помечен меткой `BeginM1`, а самый первый исполняемый оператор сегмента кода модуля `m2.asm` – меткой `BeginM2`, то значения обеих меток в таблицах имен будет равным нулю. Иначе говоря, пространства значений меток в таблицах имен обоих модулей пересекаются. И это справедливо также для сегментов данных.

Разнесение машинных команд, данных и местоположений имен различных модулей в пространстве памяти формируемой машинной программы обеспечивает компоновщик. Если компоновщику в перечне объектных модулей задать список "`m1, m2`", то в ходе компоновки все имена таблицы имен модуля `m1` сохранят свои значения, а для таблицы имен значения должны быть увеличены, исходя из предположения, что сначала в памяти размещается код сегмента из первого модуля, а затем код из одноименного сегмента второго модуля. Если компоновщику задать порядок "`m2, m1`", то значения должны быть увеличены у имен из таблицы модуля `m1.obj`. Если размеры сегментов кода и данных обоих модулей равны соответственно `SizeSegCode1`, `SizeSegData1`, `SizeSegCode2` и `SizeSegData2`, то при первом порядке значения меток второго модуля должны быть увеличены на `SizeSegCode1` и `SizeSegData1` для имен, размещаемых в соответствующих модулях. При обратном порядке в качестве приращений будут использованы соответственно `SizeSegCode2` и `SizeSegData2`. Если компоновщику передается, например, 10 объектных модулей в порядке `m1,m1, ..., m10`,

то для последнего из них в качестве приращения значений меток в сегменте кода необходимо использовать сумму

`SizeSegCode11+SizeSegCode2+ ... +SizeSegCode9.`

Естественно, что адресные поля, соответствующие измененным по значению именам, компоновщик тоже должен модифицировать, поскольку машинная команда через свой адрес должна теперь ссылаться на другое местоположение адресуемого объекта, нежели спланировал ассемблер, опираясь на продвигаемые счетчики размещения. Значения адресов в таких адресных полях называются относительными (относительно начала соответствующего сегмента своего модуля). Относительными могут быть не только адресные поля, но и другие адресные величины, например, непосредственные операнды команд загрузки указателей в регистры, используемые в дальнейшем в качестве адресных переменных. По сути дела, компоновщик планирует размещение в памяти фрагментов данных и машинных программ, фигурирующих в различных объектных модулях, и на основе результатов этого планирования настраивает все обращения к данным и командам на их реальное положение в памяти.

Поддержка отдельной компиляции предполагает использование компоновщика объектных модулей и в рассматриваемом примере требует следующего:

1) При трансляции модуля `m1.asm` необходимо как-то сообщить транслятору факт допустимости использования имени `puts` и тип этого имени. В ассемблере MASM для этого используется директива `EXTRN`. В этой директиве через запятую перечисляются пары "имя: тип". Например, имя `puts` может быть объявлено директивой " `EXTRN puts: NEAR` ". Здесь тип `NEAR` означает в 32-разрядном режиме адресации архитектуры IA-32 четырехбайтный адрес, позволяющий адресовать `puts` в том же самом сегменте кода, в котором находится команда вызова подпрограммы. Если бы использовался тип `FAR`, то для адресации `puts` потребовалась бы шестибайтная пара "Сегментный регистр –



Внутрисегментное смещение". В случае типа NEAR длина машинной команды `call puts` будет равна 5, а в случае FAR – 7 байтам. Это значит, что приращение к счетчику размещения на этапе формирования таблицы имен зависит от типа внешнего имени.

2) При трансляции исходного модуля необходимо сформировать и поместить в объектный модуль такую таблицу указателей на адресные поля, которая позволит компоновщику настроить соответствующие команды на обращение по реальным, а не относительным адресам.

3) При трансляции исходного модуля необходимо сформировать и поместить в объектный модуль таблицу тех имен, которые определены по месту в текущем модуле, но используются для ссылки на данные или команды в других модулях. Для этого в MASM используется директива PUBLIC. В этой директиве просто перечисляют через запятую те имена, спецификации которых из внутренней таблицы имен ассемблера должны быть помещены в таблицу публичных (глобальных) имен объектного модуля.

4) При компоновке с использованием большого объема библиотечных процедур целесообразно вставлять в формируемую машинную программу только код используемых процедур. Если мы прокомпилируем исходный модуль `proc.asm`, включающий в себя, скажем, 100 процедур, то при компоновке с вовлечением модуля `proc.obj` коды всех 100 процедур попадут в формируемую программу, даже если мы обращаемся только к нескольким из них. Чтобы такого не происходило, наборы процедур оформляют как библиотеки `.lib`, внутри которых имеются каталоги публичных имен с указанием на то, в каком объектном модуле библиотеки это имя определено. Компоновщик обращается к библиотекам в случае, когда какое-то имя, фигурирующее в директиве EXTRN, отсутствует в таблицах имен, сформированных в `obj`-файлах согласно директивам PUBLIC. Если имя в библиотеке найдено, то в формируемую программу вставляется только тот библиотечный модуль, который содержит определение имени.

Ассемблер в целях поддержки отладки формирует также файл листинга `.lst`, а некоторые ассемблеры еще и файл таблицы перекрестных ссылок `.xrf`. Файл листинга очень удобен в случае, когда используемый отладчик работает на уровне машинного кода, показывая ассемблерные

команды только как результат дизассемблирования. Листинг может использоваться также в случаях, когда отладчика вообще нет, но программа зависает в связи с такими ошибками, которые не искажают алгоритм обработки данных в части последовательности операций, что и создает трудности их локализации. Это обычно ошибки адресации данных (вместо адреса массива пересылается содержимое первого элемента, вместо содержимого ячейки памяти используется ее адрес, вместо  $m$ -го элемента массива адресуется элемент со смещением от начала массива, равным  $m$ , и т. п.), ошибки представления данных машинными операндами не той длины, ошибки, не учитывающие наличие или отсутствие знака у операнда (signed и unsigned).

Листинг часто формируется на том же самом проходе, что и объектный модуль, поскольку в листинге отражается машинный код и таблицы имен, определяемые в режиме раздельной компиляции без учета влияния на окончательный машинный код других модулей и компоновщика.

Для целей отладки также используется карта загрузки \*.map, формируемая компоновщиком.

### **1.3. Инструментальные средства программирования на ассемблере**

#### **1.3.1 Ассемблер-трансляторы и компоновщики**

Как уже было отмечено в подразделе 1.1, существует довольно много ассемблеров для семейства архитектур Intel x86. При использовании MASM фирмы Микрософт транслятор имеет имя ml.com, а компоновщик – link.exe. Если у вас установлена среда Visual Studio, то в подкаталоге bin каталога, где установлен Си-транслятор, легко можно найти эти программы и даже использовать для разработки простых ассемблер-программ. Правда, при этом придется самостоятельно определять значения и имена, используемые для выполнения запросов к

операционной системе. А без этих запросов нельзя организовать ввод-вывод данных.

Фирма Микрософт уже много лет не распространяет систему программирования на ассемблере как независимый и полноценный комплект программ. Однако в Internet можно найти несколько сборок для программирования на MASM, наиболее популярной из которых является так называемая «сборка Хатча» (Steve Hutchesson, Australia) MASM32 SDK, последнюю версию которой можно свободно загрузить с сайта <http://masm32.com>. Пакет поставляется в виде архива, который нужно просто распаковать в какой-то каталог. Лучше всего его распаковать в корневой каталог какого-то носителя. При этом создается директорий `masm32` с довольно большим числом внутренних подкаталогов.

Наиболее важными для изучения машинно-ориентированного программирования являются такие подкаталоги:

`bin` – здесь находится ассемблер-транслятор `ml.exe`, компоновщик `link.exe`, программа работы с библиотеками `lib.exe`, дизассемблер `dumpre.exe`, компилятор ресурсных файлов `rc.exe`, несколько `bat`-файлов для автоматизации использования программ каталога, а также ряд вспомогательных программ и файлов с лаконичными описаниями механизмов их использования;

`include` – каталог с файлами `*.inc`, в которых специфицированы имена констант, переменных и подпрограмм, а также макроопределения; эти спецификации вставляются в ассемблер-программу с помощью директивы `include`;

`lib` – каталог с библиотеками объектных модулей, используемыми компоновщиком для сборки программ, в которых используются библиотечные процедуры;

`example` – примеры ассемблер-программ;

`help` – подборка `chm`-файлов; для изучения МОП наиболее важными являются файлы: `asmintro.chm` – введение в ассемблер `MASM32`; `masm32.chm` – описание языка ассемблера; `masmlib.chm` – описание библиотечных процедур;

`tutorial` – подборка вспомогательных материалов для изучения, в том числе примеры программ для консольного режима;

`m32lib` – исходные тексты библиотечных процедур.

Рассмотрим использование сборки `MASM32 SDK` на примере.

### **Пример 1.6. Апробация сборки `MASM32 SDK`**

Пусть нам требуется апробировать `MASM32 SDK` на самой простой программе, которая ничего не обрабатывает, но результат ее работы все же можно наблюдать. Очевидно, что минимально необходимой функцией программ является возврат управления операционной системе. Это можно сделать двумя способами.

Первый способ заключается в вызове процедуры `ExitProcess`, объектный модуль которой находится в библиотеке `kernel32.lib`. У этой процедуры есть операнд, через который в ОС передается код возврата. Поскольку этот код может быть доступен в консольном режиме через переменную окружения `ErrorLevel`, минимальная программа может содержать только спецификацию кода возврата и выход через процедуру `ExitProcess` с передачей ей этого кода.

Второй способ основан на том факте, что `Windows` после загрузки программы в память использует стартовую метку в качестве адреса подпрограммы. То есть `Windows` просто вызывает подпрограмму с адресом стартовой метки. Это значит, что для возврата в `Windows` достаточно выполнить команду выхода из подпрограммы `ret`. Код возврата при этом должен быть в регистре `EAX`.

С целью увидеть результат компоновки будем использовать первый способ. При втором способе компоновщик, по сути дела, просто строит `exe`-файл на основе одного `obj`-файла, а при первом – компоует наш объектный модуль с объектным модулем процедуры `ExitProcess`.

Откроем любой редактор текста, способный редактировать обычные текстовые файлы. Это может быть Notepad, Notepad++ и т. п. Наберем в нем исходный код задуманной нами самой простой программы:

```
.386 ; Транслируем для IA-32
; Задаем модель памяти и соглашение о вызове процедур
.model flat, stdcall
; Подключаем файлы для использования ExitProcess
include kernel32.inc
includelib kernel32.lib
; В сегменте данных поместим только код возврата
.data
cret dd 5 ; 32-разрядный код возврата
.code ; В сегменте кода только возврат в Windows
main proc ; точка входа в программу
    pushd cret ; Передача кода возврата через стек
    call ExitProcess ; Вызов процедуры возврата
main endp ; Оператор завершения процедуры
end main ; Фиксация конца программы с заданием точки ее старта
```

Сохраним программу под именем `erlevel.asm` в специально созданном для изучения МОП каталоге `MOP`, создав в нем подкаталог `ErLevel`. При этом обеспечиваем, чтобы этот подкаталог был на том логическом диске, где установлен пакет `MASM32 SDK`. Дальнейшее описание процесса приводится ниже существенно более подробно, нежели принято в учебной и технической литературе. Это сделано по трем причинам.

Во-первых, изучение ассемблера связано с активным использованием открытых исходных текстов ассемблер-программ и зачастую полезные тексты созданы в другой системе программирования на ассемблере, например, `NASN`, `YASM` и др. Разные ассемблеры для одной и той же компьютерной архитектуры в общем случае по лексике и грамматике несовместимы. Это значит, что для экспериментирования с исходными текстами, написанными на разных диалектах языка ассемблера `x86`, зачастую придется повторять процесс освоения для новых пакетов. Во-вторых, время, отводимое на дисциплину МОП в рамках учебного семестра, не столь велико, чтобы позволить себе либо чтение многостраничных руководств, либо выполнение

большого числа проб и ошибок. Рационализировать работу в таких условиях крайне необходимо. В-третьих, документацию по ассемблерам пишут в расчете на довольно квалифицированных программистов, а многие студенты пока таковыми не являются.

Заглянем в bat-файлы подкаталога bin пакета MASM32 SDK для нахождения файла сборки консольной программы. Находим там файл buildc.bat, который копируем в подкаталог MOP\ErLevel. В данном bat-файле обнаруживаем, что через первый параметр командной строки запуска этого файла на исполнения (параметр %1 внутри bat-файла) в процесс исполнения команды buildc.bat передается имя ассемблер-программы без указания расширения. Обнаруживаем также, что ассемблер-транслятор и компоновщик запускаются в предположении, что пакет MASM32 SDK находится в подкаталоге masm32 корневого каталога того же самого логического диска, на котором находится транслируемая программа. Это удобно, поскольку командная строка запуска ассемблера имеет такое начало: «\masm32\bin\ml ...», что позволяет установить этот пакет даже на мобильный носитель и легко продолжать программирование на ассемблере хоть на домашнем компьютере, хоть на ноутбуке, хоть на компьютере приятеля, которого просим помочь преодолеть трудности изучения.

Запускаем bat-файл с параметром erlevel и получаем следующее сообщение об ошибке:

```
erlevel.asm(4) : fatal error A1000: cannot open file : kernel32.inc
```

Из этого сообщения видно, что ассемблеру неизвестен путь к включаемому файлу kernel32.inc. У нас есть три варианта решения этой проблемы: а) скопировать kernel32.inc в каталог MOP\ErLevel; б) указать в исходном модуле полное имя файла; в) найти среди опций компилятора такую, которая позволяет указать путь к подкаталогу, где находится включаемый файл. Третий вариант явно более технологичнее двух первых.

Чтобы осуществить третий вариант, нам нужно либо заглянуть в документацию на ассемблер, либо просто воспользоваться встроенной справкой транслятора, набрав в консоли команду

```
"\masm32\bin\ml /?".
```

Использование этой команды проще и оно дает следующий текст:

ML [ /options ] filelist [ /link linkoptions ]

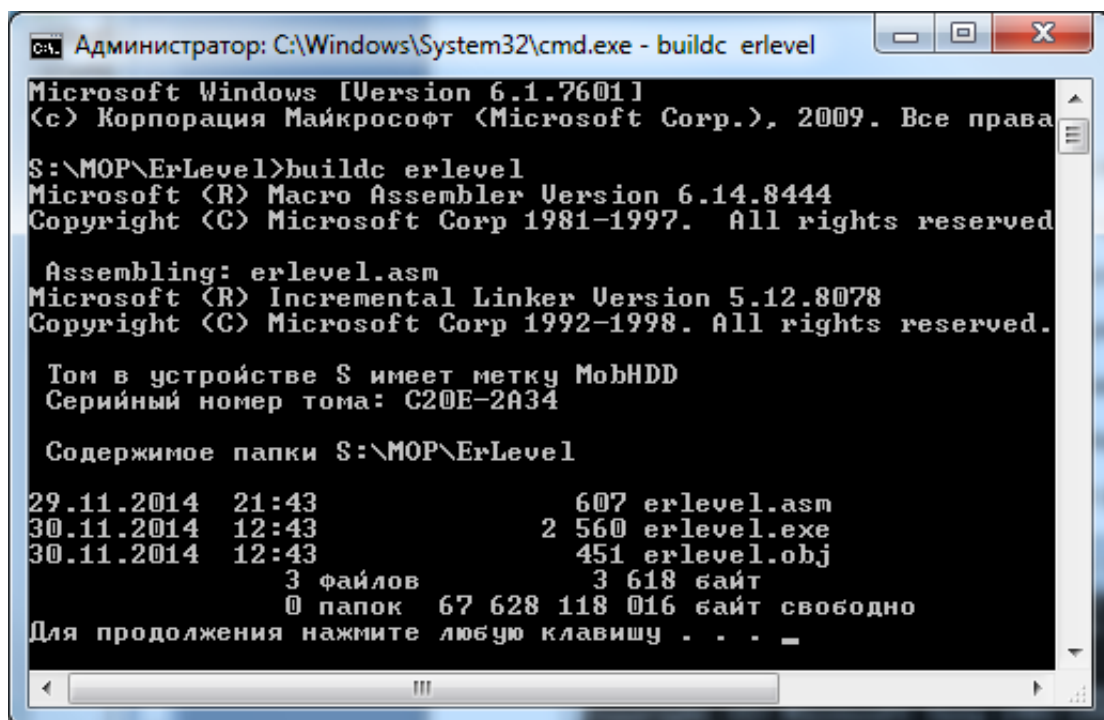
/Bl<linker> Use alternate linker	/safeseh Assert all exception handlers are declared
/c Assemble without linking	
/Cp Preserve case of user identifiers	/Sf Generate first pass listing
/Cu Map all identifiers to upper case	/Sl<width> Set line width
/Cx Preserve case in publics, externs	/Sn Suppress symbol-table listing
/coff generate COFF format object file	/Sp<length> Set page length
/D<name>[=text] Define text macro	/Ss<string> Set subtitle
/EP Output preprocessed listing to stdout	/St<string> Set title
/F <hex> Set stack size (bytes)	/Sx List false conditionals
/Fe<file> Name executable	/Ta<file> Assemble non-.ASM file
/Fl[file] Generate listing	/w Same as /W0 /WX
/Fm[file] Generate map	/WX Treat warnings as errors
/Fo<file> Name object file	/W<number> Set warning level
/Fr[file] Generate limited browser info	/X Ignore INCLUDE environment path
/FR[file] Generate full browser info	/Zd Add line number debug info
/G<c d z> Use Pascal, C, or Stdcall calls	/Zf Make all symbols public
/I<name> Add include path	/Zi Add symbolic debug info
/link <linker options and libraries>	/Zm Enable MASM 5.10 compatibility
/nologo Suppress copyright message	/Zp[n] Set structure alignment
/omf generate OMF format object file	/Zs Perform syntax check only
/Sa Maximize source listing	
/errorReport:<option> Report internal assembler errors to Microsoft	
none - do not send report	
prompt - prompt to immediately send report	
queue - at next admin logon, prompt to send report	
send - send report automatically	

Находим в этом тексте опцию “/I<name> Add include path” и получаем новую версию строки запуска транслятора в файле buildc.bat:

```
\masm32\bin\ml /c /coff /I\masm32\include %1.asm
```

Вновь запускаем bat-файл и получаем другое сообщение об ошибке: LINK : fatal error LNK1104: cannot open file "kernel32.lib"

Понятно, что теперь придется добавить пути к файлам библиотек еще и в строку запуска компоновщика. Для знакомства со списком опций набираем команду «\masm32\bin\link >link.log» и сразу получаем в файле link.log текст с полным описанием всех опций, в котором присутствует опция /LIBPATH:dir. Добавляем в строку вызова компоновщика bat-файла конструкцию «/LIBPATH:\masm32\lib». Здесь нас может подстерегать еще одно разочарование – повторение сообщения об ошибке компоновки, что естественно, когда мы хотим сделать что-то очень быстро. Обнаруживается, что в файле buildc.bat две строки запуска компоновщика. Добавляем опцию пути поиска библиотек во вторую строку запуска компоновщика bat-файла и, запустив его, наконец-то получаем результат, представленный на рис. 1.2 в виде экрана консольного режима.



```
Администратор: C:\Windows\System32\cmd.exe - buildc erlevel
Microsoft Windows [Version 6.1.7601]
(c) Корпорация Майкрософт (Microsoft Corp.), 2009. Все права
S:\MOP\ErLevel>buildc erlevel
Microsoft (R) Macro Assembler Version 6.14.8444
Copyright (C) Microsoft Corp 1981-1997. All rights reserved

Assembling: erlevel.asm
Microsoft (R) Incremental Linker Version 5.12.8078
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

Том в устройстве S имеет метку MobHDD
Серийный номер тома: C20E-2A34

Содержимое папки S:\MOP\ErLevel
29.11.2014  21:43                607 erlevel.asm
30.11.2014  12:43             2 560 erlevel.exe
30.11.2014  12:43             451 erlevel.obj
              3 файлов              3 618 байт
              0 папок 67 628 118 016 байт свободно
Для продолжения нажмите любую клавишу . . .
```

Рис. 1.2. Экран успешного выполнения трансляции и компоновки

Теперь можно запустить программу и посмотреть результат ее работы. Если при работе с bat-файлом мы могли все делать, запуская buildc.bat из среды какого-нибудь менеджера файлов, то сейчас нам придется либо сделать bat-файл с запуском программы erlevel.exe и оператором pause с целью увидеть результат работы программы, либо войти в режим командной строки с



помощью консольной команды `cmd`. Второй вариант проще, если не предполагать многократное исполнение разработанной программы.

Входим в режим командной строки, запускаем нашу программу и, увы, программа ничего не выдает. Что и следовало ожидать, поскольку в ней нет обращения к процедурам вывода данных. Понимая, что код возврата программы операционная система Windows помещает в переменную окружения `errorlevel`, и зная, что любую переменную окружения мы можем посмотреть с помощью команды `echo` и конструкции `%ИмяПеременной%`, набираем в командной строке `«echo %errorlevel%»` и получаем желаемое – число 5, равное коду возврата.

Файл `buildc.bat` составлен таким образом, чтобы на консоли был виден результат работы транслятора в формате распечатки информации обо всех файлах с именем, равным первому аргументу `«%1»`. На рисунке 1.2 можно видеть результат команды `dir`. Получается всего три файла: `erlevel.asm`, `erlevel.obj` и `erlevel.exe`.

На рисунке 1.1 в качестве продуктов трансляции и компоновки фигурируют файлы `*.lst` и `*.map`. Чтобы посмотреть листинг и карту загрузки, нам необходимо в списке опций ассемблера `ml` найти строку `«/Fl[file] Generate listing»` и вставить опцию `«/Fl»` в командную строку трансляции, затем из списка опций компоновщика перенести в строки его запуска опцию `«/MAP»`.

После такого изменения `bat`-файла его запуск порождает файлы `erlevel.lst` и `erlevel.map`. список из пяти файлов, причем файл листинга имеет размер более 172 тысяч байт, а при просмотре текстовым редактором мы обнаруживаем в этом файле более 3200 строк. Очевидно, что большинство строк взято директивой `«include kernel32.inc»`.

Поскольку нам хотелось бы посмотреть листинг той части, что написано нами, а не разработчиками библиотеки `kernel32.lib`, придется поискать директивы ассемблера, которые способны включать-выключать процесс генерации листинга по ходу трансляции исходного кода. Открываем файла `masm32.chm` подкаталога `help` пакета `MASM32` и через несколько секунд видим описание директивы `.NOLIST`, которое представлено на рисунке 1.3.

На основе полученного описания вставляем в исходный текст ассемблер-программы (файл `erlevel.asm`) директиву `.NOLIST` перед директивой «include `kernel32.inc`» и директиву `.LIST` после директивы `include`. После этого размер листинга уменьшился раза в два.

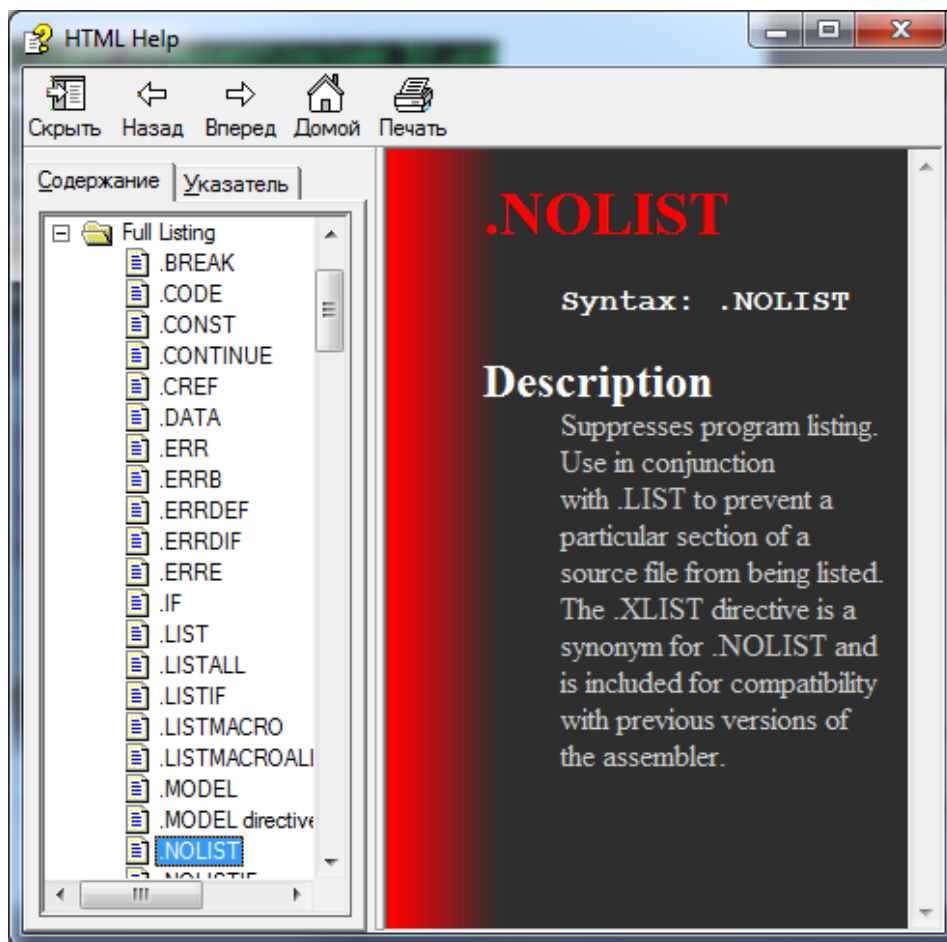


Рис. 1.3. Использование справки MASM32.CHM для получения информации о директиве отключения генерации листинга

Текст нашей программы мы сразу видим в листинге, и он выглядит так:

```
Microsoft (R) Macro Assembler Version 6.14.8444      12/02/14
20:27:54
erlevel.asm                                         Page 1 - 1

        .386 ; Транслируем для IA-32
        ; Задаем модель памяти
        ; и соглашение о вызове процедур
        .model flat, stdcall
        .NOLIST
        .LIST
        ; В сегменте данных поместим только код возврата
00000000    .data
        ; 32-разрядный код возврата
00000000 00000005    cret    dd 5
```

```

                                ; В сегменте кода только возврат в Windows
00000000      .code
                                ; точка входа в программу
00000000      main proc
                                ; Передача кода возврата через стек
00000000  FF 35 00000000 R      pushd cret
                                ; Вызов процедуры возврата
00000006  E8 00000000 E      call ExitProcess
                                ; Оператор завершения процедуры
0000000B      main endp
                                ; Фиксация конца программы
                                ; с заданием точки ее старта
                                end main

```

В приведенном листинге легко обнаруживаются значения счетчиков размещения для обоих сегментов – это первый столбец, представленный в 16-ричном виде. Можно видеть коды команд `pushd` и `call`, легко предположить, что символ `R` после адресной части команды `pushd` означает «относительный адрес», а символ `E` – «внешний».

В то же время файл листинга содержит слишком много информации, мало относящейся к тексту исходного модуля. Это таблица символов, определенных в том числе и в `kernel32.inc`. Заглянув еще раз в список опций командной строки ассемблера, находим там строку «/Sn Suppress symbol-table listing», обещающую подавление таблицы символов. Вставив опцию `/Sn` в командную строку запуска `ml` в файле `builddc.bat`, получаем листинг размером всего 868 символов, в котором фигурирует только часть, представленная выше.

Несмотря на некоторую «нудность» описания, рассмотренный выше процесс осуществляется довольно быстро. Хотя никто не застрахован от сообщения «`erlevel` не является внутренней или внешней командой, исполняемой программой или пакетным файлом». Это сообщение означает, что файла `erlevel.exe` в текущем подкаталоге нет. Причиной такого сообщения может быть работа антивируса. Только был создан `erlevel.exe`, как тотчас же антивирус переместил его в карантин. Автору этих строк пришлось наблюдать такое в двух окнах: в одном отражался процесс выполнения `builddc.bat`, в другом – экран `Total Commander` в реальном времени отображал сначала появление в списке файлов элемента `erlevel.exe`, а затем пропадание примерно через секунду. Вряд ли

в базе штаммов вирусов нашлось что-то похожее на ничего не делающую программу. Поводом для беспокойства антивируса скорее всего является слишком малый размер программы. В рассматриваемом случае нужно либо отключать антивирус, либо сконфигурировать его так, чтобы полученные с помощью ассемблера программы не становись жертвой антивирусных атак.

После внесения нескольких изменений в bat-файл buildc.bat можно его переименовать, дав более короткое имя (назовем его bu.bat), удалив строки, относящиеся к компоновке ресурсного файла rsrc.obj, и добавив явное задание путей к местонахождению файлов пакета MASM32 DSK. В результате получаем следующий текст:

```
@echo off
set pm=\masm32\
path %pm%bin;%path%
if exist %1.obj del %1.obj
if exist %1.exe del %1.exe
ml /c /Fl/Sn /coff /I%pm%include %1.asm
if errorlevel 1 goto errasm
link /SUBSYSTEM:CONSOLE /LIBPATH:%pm%lib /OPT:NOREF
%1.obj
if errorlevel 1 goto errlink
dir %1.*
goto TheEnd
:errlink
echo _
echo Link error
goto TheEnd
:errasm
echo _
echo Assembly Error
goto TheEnd
:TheEnd
pause
```

Теперь попробуем изменить программу так, чтобы в ней не было вызова процедуры ExitProcess. При этом проверим, входит ли имя стартовой процедуры в список соглашений об именах MASM32. Для этого

меняем это имя на `erlevel` и даже объявляем это имя как метку оператора ассемблера, а не как имя процедуры. Помещаем такую модификацию программы в файл с новым именем `erlevelr.asm` со следующим содержанием:

```
.386
.model flat, stdcall
.code
erlevel: ; Точка входа в программу
    mov eax, 12345678
    ret ; Выход в Windows
end erlevel ; конец исходного модуля с заданием стартовой метки
```

После трансляции убеждаемся, что ассемблер никаких предупреждений нам не делает, то есть предоставляет программисту большую степень свободы. Сравниваем размеры продуктов трансляции:

`erlevel.exe` – 2560 байтов, `erlevelr.exe` – 1024 байта,  
`erlevel.obj` – 451 байт, `erlevelr.obj` – 346 байтов.

Как видим, отказ от использования процедуры `ExitProcess` приводит к заметному уменьшению размера программы.

Вставая на путь экспериментирования с различными вариантами реализации требуемой функциональности, целесообразно автоматизировать с помощью `bat`-файлов не только сборку программ, но и исполнение. В рассматриваемом случае это может быть `bat`-файл с таким содержанием:

```
erlevel.exe
@echo off
echo Return code = %errorlevel%
@echo on
erlevelr.exe
@echo off
echo Return code = %errorlevel%
```

Результатом его работы будет такой вывод в окно консоли:

```
S:\MOP\ErLevel>erlevel.exe
Return code = 5
S:\MOP\ErLevel>erlevelr.exe
```

Return code = 12345678

Представленный выше процесс предполагает, что программист редко совершает логические ошибки. У начинающих ассемблер-программистов это не так даже для простых программ, что заставляет использовать специальные средства отладки.

### **1.3.2 Средства отладки ассемблер-программ**

#### **Функции отладчиков**

Основным инструментом отладки являются программы отладчики. Основные возможности отладчиков таковы:

- покомандная трассировка программ с возможностью не заходить в подпрограммы;
- слежение за состоянием регистров и памяти;
- назначение точек и условий останова;
- слежение за состоянием стека, в том числе за адресами возврата;
- дизассемблирование машинной программы.

#### **Отладчик OllyDbg**

Для учебного процесса естественно использовать такие отладчики, которые распространяются свободно. Среди ассемблер-программистов популярен отладчик OllyDbg разработки украинского программиста Олега Ющука. Отладчик доступен для свободного скачивания на сайте автора <http://www.ollydbg.de>. Некоторые функции отладчика требуют прав администратора, однако выполнение отладочных действий в рамках заданий дисциплины МОП возможно и без таких прав.

Для отладки конкретной программы через меню File необходимо ее выбрать в файловой системе. В заголовке окна отладчика отображается имя загруженного файла. Если загрузка выбранной программы не происходит из-за антивирусной защиты, то нужно либо включить отлаживаемую программу в «белый список» антивируса, либо отключить его на время отладочных действий.

### Пример 1.7. Программа erlevel.exe в среде отладчика OllyDbg

Результат выбора файла erlevel.exe в меню File отладчика OllyDbg показан на рисунке 1.4.

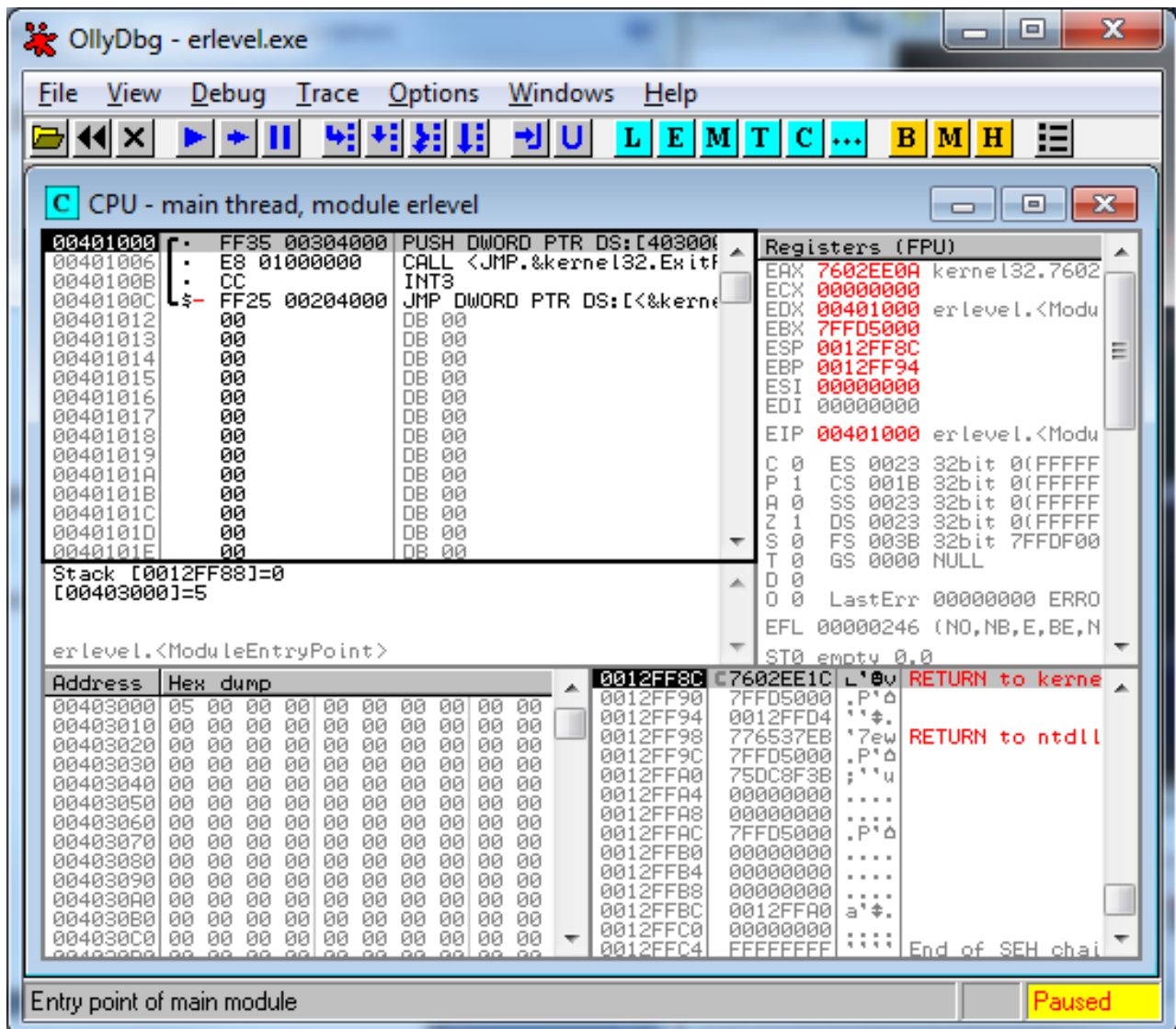


Рис. 1.4. Вид отладчика OllyDbg после загрузки программы erlevel

В левом верхнем окне представляется машинный и ассемблерный коды отлаживаемой программы. В левом нижнем окне – сегмент данных, содержащий число 5, которым инициализирована переменная cret. В правом верхнем окне отображаются значения регистров и флагов. В правом нижнем окне представлено состояние стека. Поскольку программа создана для консольного режима, при ее запуске в среде Windows создается окно консоли.

Отладка становится более комфортной, если обеспечить в ходе трансляции и компоновки создание отладочной информации. Для этого

необходимо в командную строку трансляции добавить опцию `/Zi`, а в командную строку компоновки – опцию `/DEBUG`. Для возможности получать машинные программы с отладочной информацией и без нее целесообразно скопировать файл сценарий сборки `bu.bat` в новый файл, который логично назвать `bud.bat`, затем добавить указанные ключи и командные строки запуска `ml` и `link`.

После запуска выполнения команды сборки `bud erlevel` получаем целевые файлы `erlevel.obj` и `erlevel.obj` существенно большего размера. Кроме того, в каталоге проекта появляется специальный файл `erlevel.pdb`, содержащий отладочную информацию.

После запуска отладчика мы можем видеть в тексте трассируемой программы имена данных и меток перехода. Можем даже отслеживать значения имен, активизировав окно `Watches` через соответствующую позицию меню `View`.

#### **Пример 1.8. Имена данных и метки перехода в среде OllyDbg**

На рисунке 1.5 показан экран отладчика после загрузки программы `erlevel.exe`, полученной с помощью пакетного файла `bud.bat`.

В верхней строке окна сегмента кода операнд команды `push` в отличие от рисунка 1.4 представлен именем `cret`, а не абсолютным адресом.

В окне `Watches` представлено значение имени `cret = 404000h`. Окно отлаживаемой программы показано в состоянии после выполнения машинной команды `push cret` в режиме трассировки командой отладчика `Step over (F8)`. Поэтому в вершине стека мы видим число 5.

Трассировка машинного кода в среде отладчика может быть весьма затратным процессом. Особенно если в программе имеются циклы с большим числом повторов. Существенную экономию времени можно получить, если активно варьировать трассировочные действия – делать шаги без захода в подпрограмму, пользоваться различными вариантами точек останова. Такие команды представлены в меню `Debug` и `Trace`. Состав этих меню меняется в зависимости от состояния процесса



трассировки. Команды отладчика, обслуживающие процесс собственно отладки, доступны как через меню, так и функциональные клавиши.

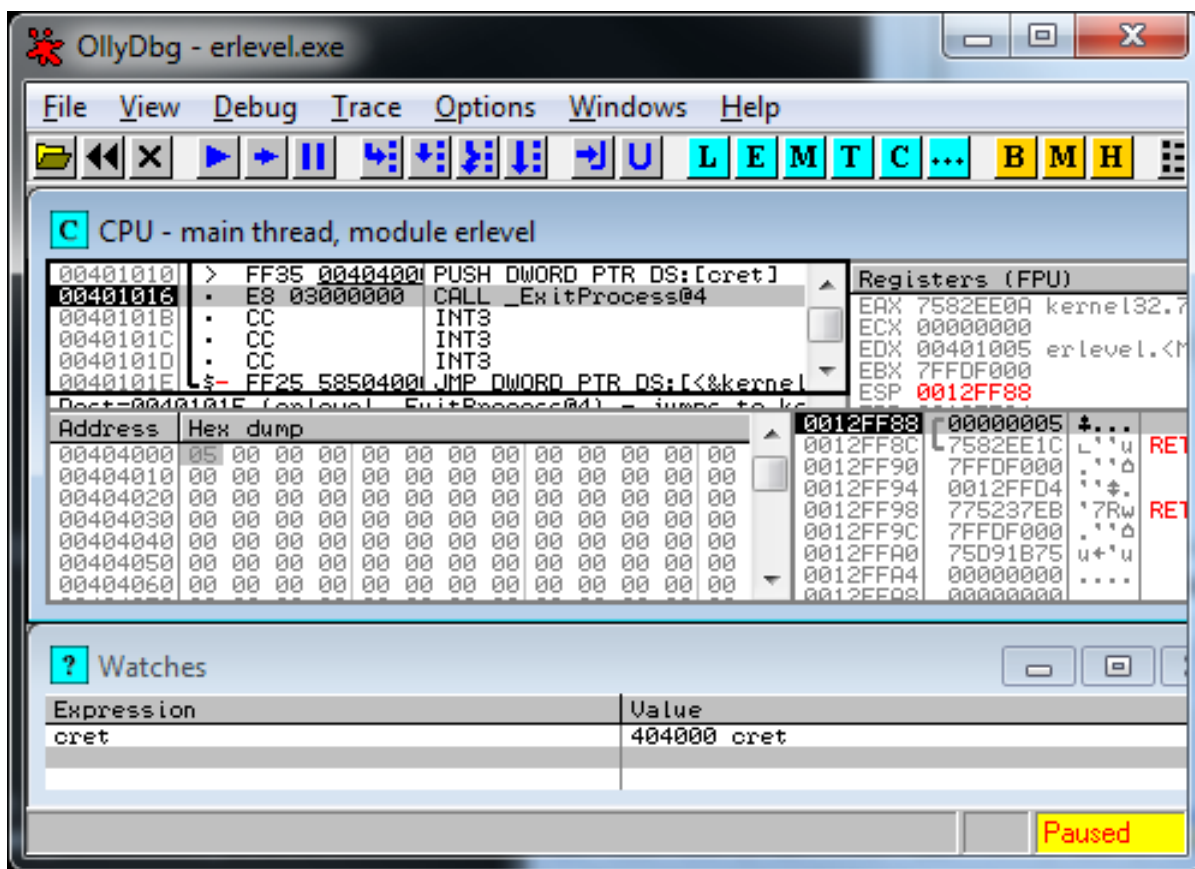


Рис. 1.5. Загрузка отладочной версии erlevel в отладчик

В начале сеанса отладки создается udd-файл, в котором записывается информация для поддержки многосеансовой работы, чтобы можно было продолжить трассировочный процесс в другом сеансе работы на компьютере.

Для версии OllyDbg v2.01 [alpha 1] список основных команд управления трассировкой таков:

- Run (F9) – запуск приложения;
- Run thread (F11) – запуск нити;
- Restart (Ctrl+F2) – перезапуск отлаживаемой программы;
- Close (Alt+F2) – прекращение отладки с закрытием машинной программы;
- Step into (F7) – шаг с заходом в подпрограмму;
- Step over (F8) – шаг без входа трассировщика в подпрограмму;
- Animate into (Ctrl+F7) – автоповтор команды Step into;

Animate over (Ctrl+F8) – автоповтор команды Step over;  
 Execute till (Alt+F7) – выполнить до выхода из подпрограммы;  
 Trace (Ctrl+F11) – автотрассировка с заходом в подпрограммы;  
 Trace (Ctrl+F12) – автотрассировка без захода в подпрограммы;  
 Step over (F8) – шаг без входа трассировщика в подпрограмму;  
 Set condition (Ctrl+T) – назначить условия останова автотрассировки;  
 Breakpoint Toggle (F2) – установить/сбросить точку останова на машинной команде, находящейся в фокусе курсора;

Set condition (Ctrl+T) – назначить условия останова автотрассировки.

Для машинной команды, находящейся в фокусе курсора, поддерживается меню, всплывающее по нажатию на правую кнопку мыши. В этом меню находится широкий спектр команд управления трассировкой.

Отладчик OllyDbg поддерживает изменение программы по ходу отладки, причем в режиме ассемблирования операторов. Ясно, что такая оперативная модификация машинного кода не должна приводить к такому изменению размеров машинных команд, которое меняет адреса точек перехода или портит машинные команды, не подпадающие под модификацию.

На рисунке 1.6 показана часть экрана OllyDbg после загрузки, а на рисунке 1.7 – диалог встроенного отладчика, инициированный либо через выбор меню, всплывающего по правой кнопке мыши, либо по нажатию на пробел. На рисунке 1.8 показана программа с измененной константой команды MOV.

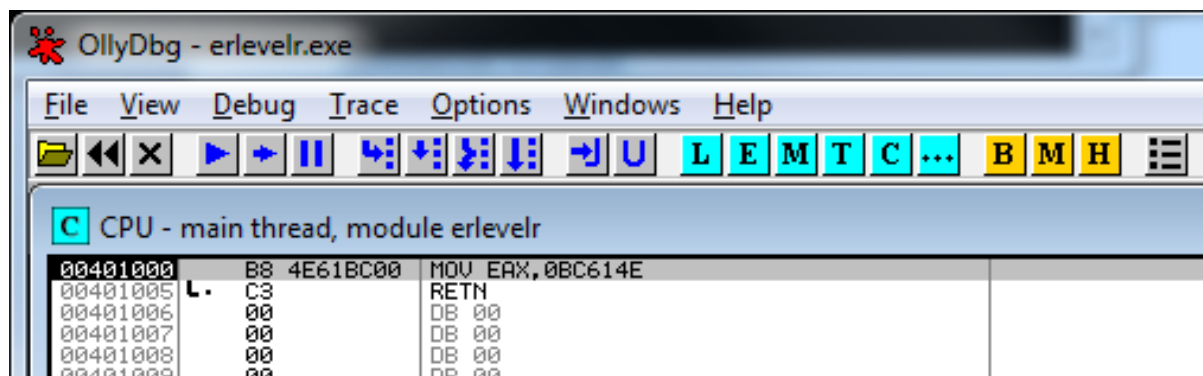


Рис. 1.6. Программа erlevelr.exe до изменения

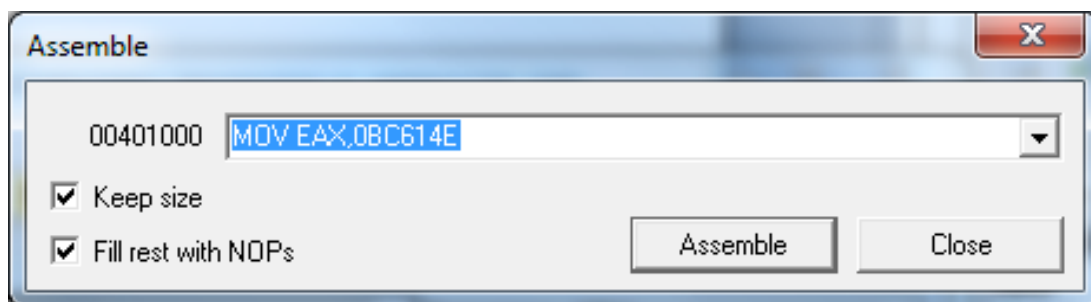


Рис. 1.7. Диалог Assemble, в котором можно изменить команду

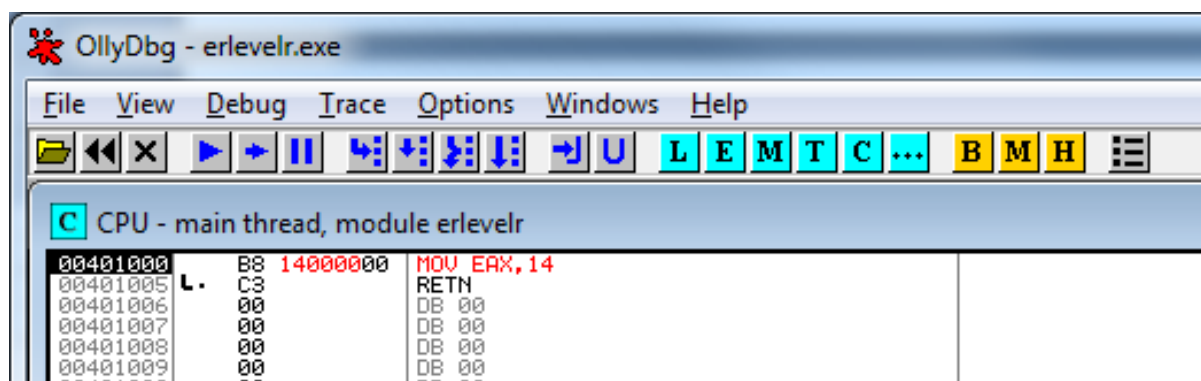


Рис. 1.8. Программа erlevelr.exe после изменения

## Организация отладочного вывода

В некоторых фазах разработки программ и исследования процессов обработки данных, запрограммированных на ассемблере, более эффективным может оказаться отладочный вывод. По сравнению с трассировкой под управлением отладчика у отладочного вывода четыре основных преимущества:

1) В окне консоли мы можем видеть последовательность значений отслеживаемых переменных, а не только текущие значения, как при трассировке под отладчиком. Это позволяет анализировать ход изменения данных на основе сопоставления «было – стало».

2) В отладочном выводе мы можем представлять только интересующие нас значения, а не множество ячеек памяти и все регистры. Это позволяет избежать ошибок, вызванных избыточностью данных на экране во время отладки.

3) Отладочный вывод можно сохранять для последующего анализа и представления в документации (в нашем случае в отчетах по лабораторным работам).

4) Отладочный вывод строится в ходе разработки программы и обладает обычно большей продуманностью, нежели трассировка в среде отладчика, которая чаще всего управляется возникающими в ходе отладки ситуациями. Продуманность отладочного вывода в сочетании со средствами генерации входных данных может играть роль контрольного примера с высокой степенью покрытия. Благодаря директивам условной трансляции средства поддержки отладки легко изолируются от отлаживаемого кода в финишной сборке.

В статье «Отладка программ» Википедии по состоянию на конец 2014 года была приведена такая цитата известных специалистов по программированию Брайана Кернигана и Роба Пайка (<https://ru.wikipedia.org/wiki/>):

«Наш личный выбор — стараться не использовать отладчики, кроме как для просмотра стека вызовов или же значений пары переменных. Одна из причин этого заключается в том, что очень легко потеряться в деталях сложных структур данных и путей исполнения программы. Мы считаем пошаговый проход по программе менее продуктивным, чем усиленные размышления и код, проверяющий сам себя в критических точках.

Щелканье по операторам занимает больше времени, чем просмотр сообщений операторов выдачи отладочной информации, расставленных в критических точках. Быстрее решить, куда поместить оператор отладочной выдачи, чем проходить шаг за шагом критические участки кода, даже предполагая, что мы знаем, где находятся такие участки. Более важно то, что отладочные операторы сохраняются в программе, а сессии отладчика преходящи.

Слепое блуждание в отладчике, скорее всего, непродуктивно. Полезнее использовать отладчик, чтобы выяснить состояние программы, в котором она совершает ошибку, затем подумать о том, как такое состояние могло возникнуть. Отладчики могут быть сложными и запутанными программами, особенно для новичков, у которых они вызовут скорее недоумение, чем принесут какую-либо пользу».

В пакете MASM32 SDK имеется библиотека процедур и макросов для поддержки отладки в режиме отладочного вывода, которая находится в

подкаталоге vkdebug. В файле VKDebug.chm находится описание всех макросов.

### 1.3.3 Интегрированная среда RadAsm

Одной из самых популярных интегрированных сред является RadASM. В RadASM встроен редактор текста, поддерживающий настраиваемую подсветку синтаксиса. RadASM объединяет несколько файлов в единый проект, при создании которого формируется каталог с файлами и набор опций проекта. Кроме того, RadAsm поддерживает многосессионную работу с проектом. В Википедии по адресам [ru.wikipedia.org](http://ru.wikipedia.org) и [en.wikipedia.org/wiki/RadASM](http://en.wikipedia.org/wiki/RadASM) имеются краткие описания и ссылки на сайт, где можно скачать редактор. Информация о RadAsm, приводимая ниже, относится к версии, данные о которой показаны на рисунке 1.9.

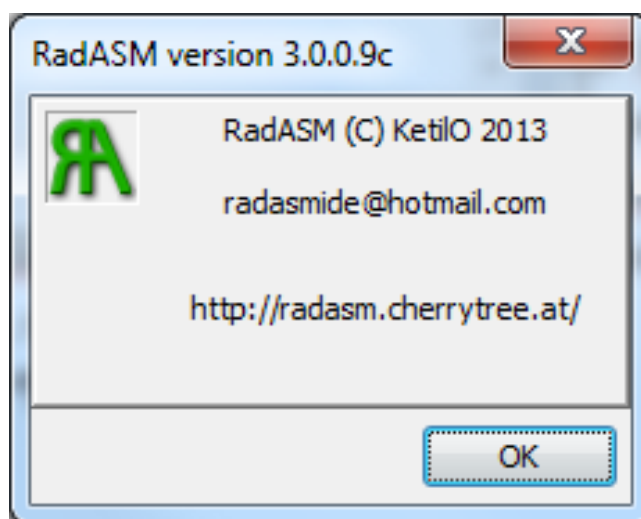


Рис. 1.9. Окно About IDE RadAsm

Установку RadAsm целесообразно выполнить на тот же логический диск, где установлен MASM32 IDE и OllyDbg. Это дает возможность настраивать пути к инструментам программирования на ассемблере без указания имени самого диска и облегчает миграцию всех средств на другой носитель, вплоть до флэшки, что может быть полезным при необходимости разрабатывать учебные программы на разных компьютерах.

После установки целесообразно «прогуляться» по содержимому каталогов пакета RadAsm и всему меню, чтобы сориентироваться. Особенно полезно заглянуть в help-файлы и файлы примеров. Затем нужно настроить пути к используемым файлам и опции командных строк, а также сформировать пространство файлов помощи интегрированной среды.

Многие настройки путей и опций командных строк IDE RadAsm используют акронимы (сокращенные имена), основными из которых являются:

\$A – путь, где находится RadAsm.exe;

\$C – имя основного исходного файла (Main File);

\$M – вовлечение в компоновку списка модулей из файла Mod.txt; этот файл создается и пополняется автоматически при отметке текущего asm-файла как модуля через позицию Toggle Current As Module меню Make;

\$O – результирующий файл сборки;

\$R – имя ресурсного файла со сценарием диалога.

Для настройки путей к файлам пакета MASM32 SDK необходимо войти в меню Option и вызвать диалог Environment, который представлен на рисунке 1.10. Именам путей path, include и lib нужно поставить в соответствие пути \masm32\bin, \masm32\include и \masm32\lib.

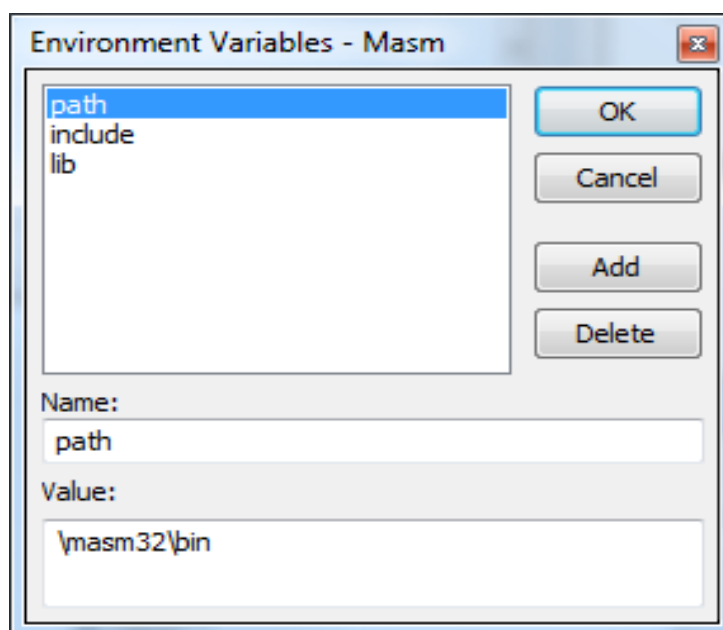


Рис. 1.10. Настройка путей к MASM32 SDK

Опции командных строк компиляции и компоновки устанавливаются через диалог Make Options, вызываемый в меню Option. Этот диалог представлен на рисунке 1.11. В RadAsm поддерживается установка опций командных строк для различного вида целевых сборок: оконные программы – Windows Release/Debug; консольные программы – Console Release/Debug; библиотеки динамической компоновки – Dll Release/Debug; библиотеки процедур – Library.

На рисунке 1.11 показаны поля настройки опций для сборки Console Debug, которая используется как основная в проектах дисциплины МОП. Эта сборка позволяет создавать отладочную информацию для отладки в среде OllyDbg, для привязки которой к среде RadAsm достаточно прописать в строке External debugger полное имя файла отладчика. Если созданная программа имеет не только учебную, но и практическую ценность, то целесообразно пересобрать ее в формате сборки Console Release, опции командной строки которой не порождают отладочную информацию.

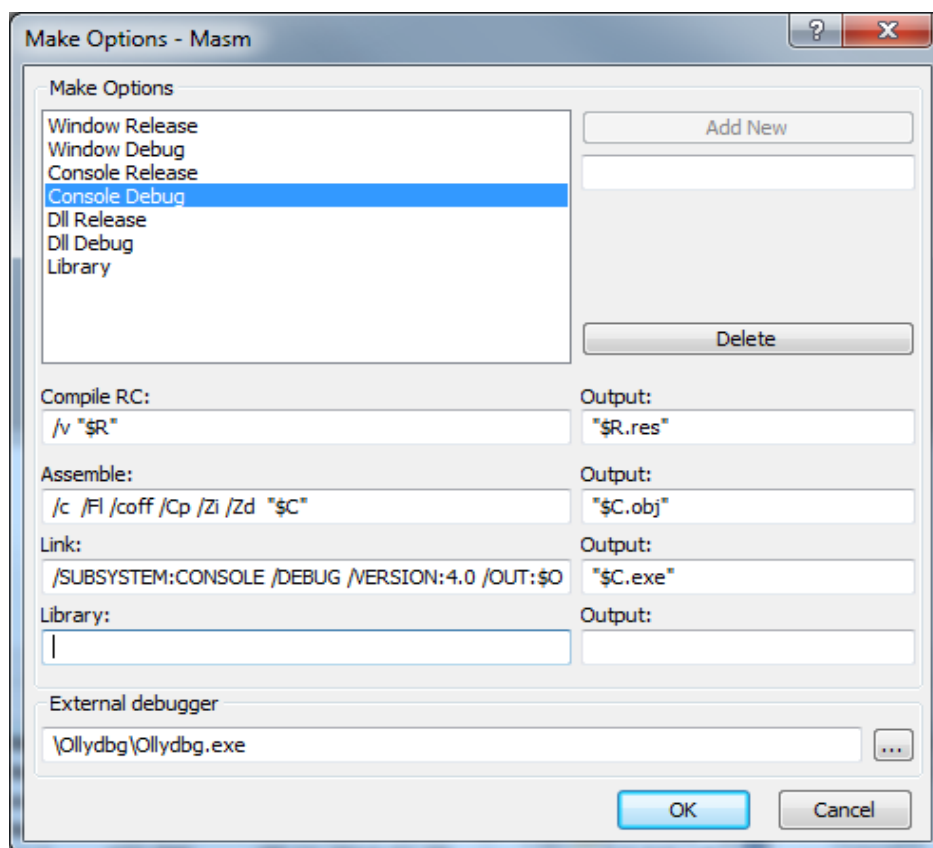


Рис. 1.11. Настройка опций компиляции и компоновки

Для формирования пространства помощи необходимо через меню Option вызвать диалог Help menu и сформировать список пар <Имя позиции help-меню – Команда или полное имя файла>.

На рисунке 1.12 показано добавление пары <MOP-Tutorial – \$A\Help\MOP-Tutorial.pdf>. Первый компонент этой пары попадает в список строк меню Help IDE RadAsm, а второй означает, что сам файл учебного пособия MOP-Tutorial.pdf должен быть размещен в подкаталоге Help каталога, в котором находится файл RadAsm.exe. Поискав в Internet по запросу «WIN API по-русски», мы можем добавить русскоязычные ресурсы Internet, например, пару <WinAPI-Ru – <http://develab.narod.ru/winapi/>>. Кому-то покажется полезным поместить в список Help-файлов электронную версию солидного учебника по ассемблеру.

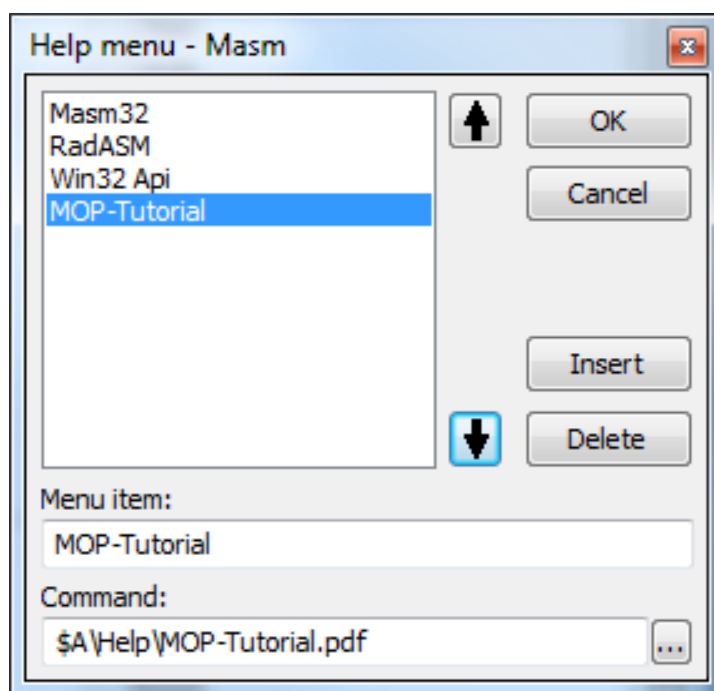


Рис. 1.12. Настройка Help-меню

Для создания проекта используется меню Project и диалог New Project. В этом диалоге задается имя проекта Project Name, описание проекта, вставляемое в заголовочную строку окна RadAsm, и тип сборки. Необходимо учитывать, что во время создания проекта опции, установленные в диалоге Make Options (рисунок 1.11), копируются в



опции нового проекта. Если возникает потребность выполнить создание исполняемого файла уже существующего проекта с другими опциями, то менять их нужно уже в опциях самого проекта через позицию Project Options меню Project. Там же мы можем расширить список возможных типов сборки проекта, например, к элементу Console Debug, которую логично выбрать при создании нового проекта, добавить элемент Console Release. После такого добавления вид сборки можно будет выбирать во всплывающем списке выбора, расположенном в самом конце строки пиктограмм команд среды RadAsm. Если предполагается всегда работать с одним видом сборки, то всплывающий список выбора видов сборки можно убрать через позицию Build Type набора флажков Toolbar меню View.

Для создания проекта и изменения его состава используется меню Project, позиции которого интуитивно понятны и не требуют описания.

Рассмотрим создание и сборку-отладку трех различных проектов, базирующихся на материале примера 1.5, в котором приведен фрагмент программы распечатки трех строк текста, заданного массивом указателей `txt`. Первый проект предполагает размещение всех исходных кодов в одном модуле `PutsTxt1.asm` (пример 1.7), второй – в двух модулях: `PutsTxt2.asm` и `io.asm` (пример 1.8), третий – в `PutsTxtL.asm` и библиотеке `io.lib` (пример 1.9).

#### **Пример 1.9. Одномодульный проект вывода текста**

Диалог создания проекта приведен на рисунке 1.13. В этом диалоге в качестве местоположения проекта указывается содержимое каталога `RadAsm\Masm\Project`, где уже имеется несколько проектов, анализ содержимого которых полезен для изучения чужого опыта. В качестве вида сборки через меню Build выбирается Console Debug.

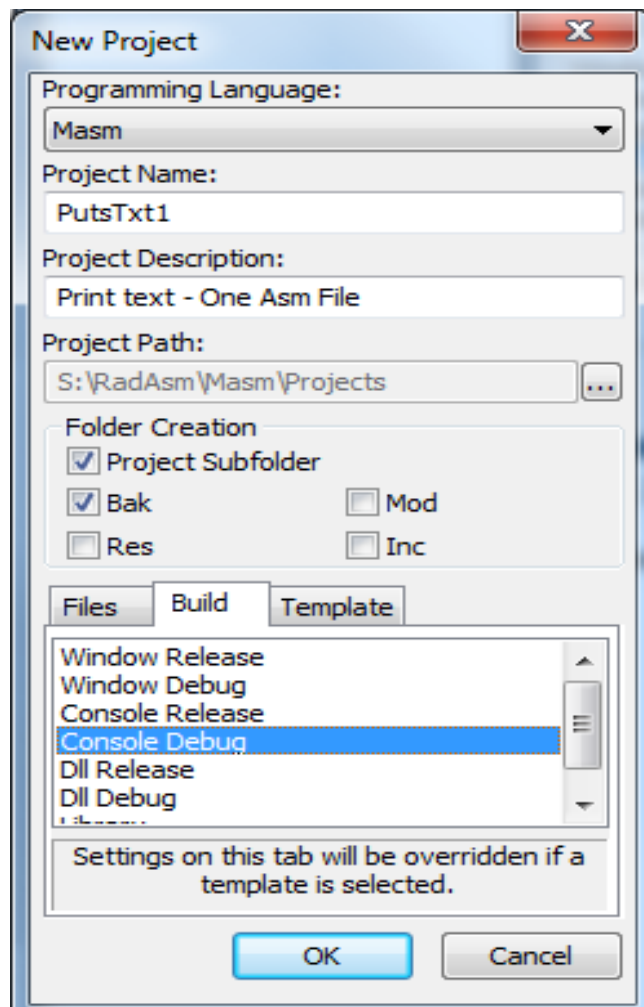


Рис. 1.13. Создание проекта PutsTxt1

После создания проекта в каталоге RadAsm\Masm\Project появляется подкаталог PutsTxt1 с пустым файлом PutsTxt1.asm и файлом описания проекта PutsTxt1.prra.

Для редактирования файла исходного модуля PutsTxt1.asm нужно кликнуть мышью на соответствующую ветвь дерева исходных файлов Assembly в окне Project (рисунок 1.14).

Для повторения описываемых действий примера 1.7 ниже приводится полный текст программы PutsTxt1.asm, который может быть либо введен вручную, либо перенесен в окно редактирования через буфер обмена из электронной версии пособия.

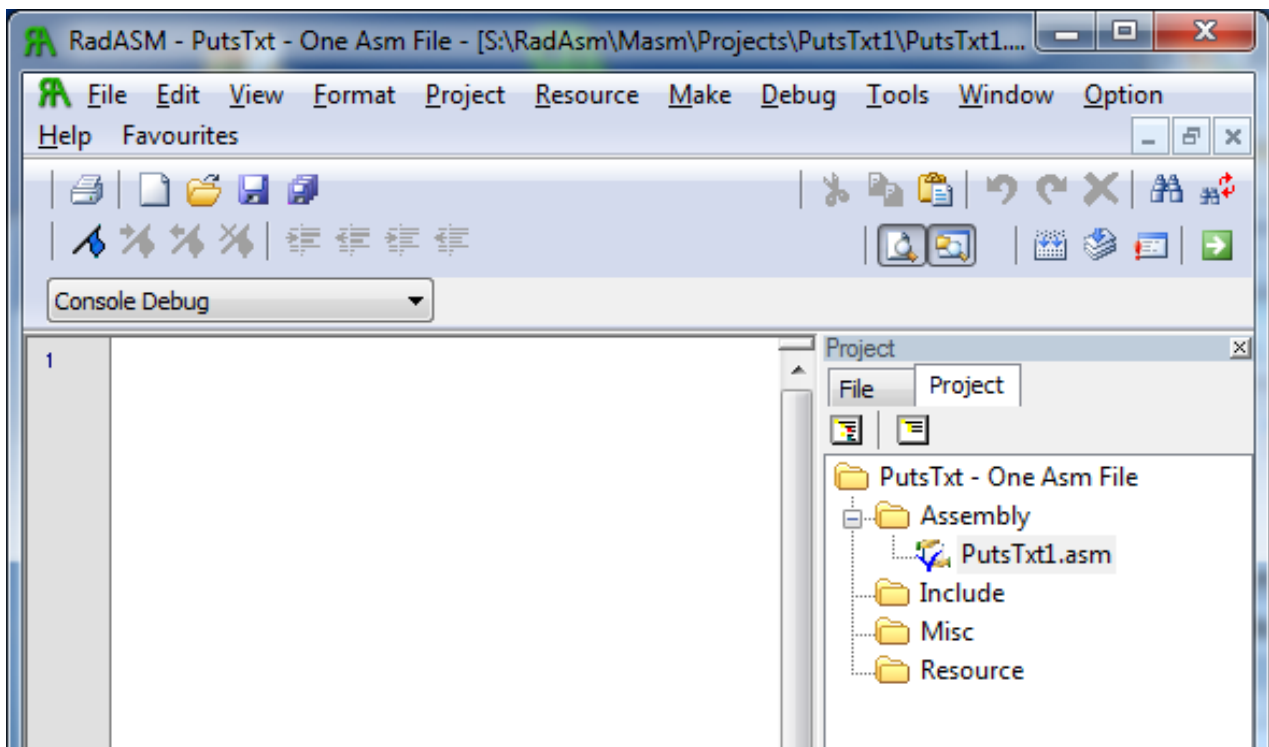


Рис. 1.14. Окно проекта PutsTxt1

Текст исходного модуля PutsTxt1.asm имеет вид:

```
.386
.model flat,stdcall
; Спецификации для функций вывода
Include kernel32.inc
Include user32.inc
Include masm32.inc
Includelib kernel32.lib
Includelib user32.lib
Includelib masm32.lib

.data
; Массив адресов строк
txt      dd s1,s2,s3,0
; Сами строки
s1       db 'Первая строка',0
s2       db 'Вторая строка',0
s3       db 'Третья строка',0
; Коды "Перевод строки, Возврат каретки"
; для перехода на следующую строку консоли
nextLine db 10, 13, 0
.code
```

```

puts PROC, pstr: DWORD
    ; Перекодировка в ASCII
    Invoke CharToOemA, pstr, pstr
    ; Вывод перекодированной строки
    Invoke StdOut, pstr
    ; Переход на следующую строку
    Invoke StdOut, ADDR nextLine
    ret
puts endp
; Основная программа содержит цикл вывода
Main PROC
    mov esi, offset txt ; p = txt
@For:
    cmp dword ptr [esi], 0 ; *p != NULL?
    je @AFor ; Выход при обнаружении 0
    ; puts(*p)
    push [esi]
    call puts
    add esi, 4 ; p++
    jmp @For
@AFor:
    mov eax, 0
    ret
Main ENDP
END Main

```

После ввода приведенного выше текста в окне редактора текста выполняем трансляцию и компоновку, запустив команду Build в меню Make RadAsm. Сообщение Make done в нижнем окне с именем Output говорит о нормальном построении загрузочного файла. В случае сообщений об ошибках необходимо проверить точность ввода исходного текста в тех строках, номера которых фигурируют в сообщениях об ошибках, и, возможно, доступность используемых в директивах include файлов. Необходимо учитывать, что команда Build не сможет выполниться, если целевой файл сборки \*.exe загружен в отладчике. Отладчик блокирует доступ к этому файлу, и команда Build не может удалить старый файл и скомпоновать новый. Если возникла такая ситуация, то необходимо закрыть отлаживаемый файл в OllyDbg .

Запуск программы можно осуществить командой Run меню Make, однако результаты работы программы PutsTxt1 мы не увидим, поскольку время существования окна с выведенными строками слишком мало. Другим способом запуска является вход в режим командной строки через запуск команды Command line меню Tools. RadAsm по этой команде запускает режим командной строки в каталоге текущего проекта. В этом режиме имеет смысл сначала посмотреть состав каталога с помощью команды dir, а затем запустить полученную программу командой PutsTxt1.

Если в программе имеются логические ошибки, то мы их увидим по неправильному выводу. В этом случае необходимо выполнить отладочные действия в среде отладчика, запустив в меню Make команду Run With Debug.

### **Пример 1.10. Двухмодульный проект вывода текста**

Для создания этого проекта выполним действия, аналогичные тем, что рассмотрены в примере 1.7. При этом в качестве имени проекта будем использовать PutsTxt2.

После создания проекта через буфер обмена копируем исходный текст файла PutsTxt1.asm в пустое окно редактирования файла PutsTxt2.asm. Далее в окне проекта правой кнопкой мыши на ветви Assembly вызываем меню, в котором выбираем позицию Add Item/New File и создаем файл io.asm. Теперь наша задача состоит в том, чтобы перенести процедуру puts в модуль io.asm и создать условия для его использования в основном модуле PutsTxt2. Именно это приходится делать в случае, когда программа становится слишком большой и возникает потребность разделить ее на модули. Для навигации по модулям нужно включить флажок Tab Select в меню View.

Вместе с текстом собственно процедуры перенесем все директивы Include и Includelib, поскольку они фигурируют в программе исключительно для поддержки вывода. Имеет смысл перенести и объявление строки символов перехода на следующую строку nextLine. После этого копируем первые два оператора основного модуля PutsTxt2 в начало модуля io, поскольку соглашения, зафиксированные в этих операторах, должны быть общими для обоих модулей. В конец модуля io добавляем оператор END, чтобы ассемблер ml смог обнаружить конец исходного модуля.

В основном модуле PutsTxt2.asm потребуется объявить функцию puts, для чего используется директива PROTO:

```
puts PROTO :DWORD
```

В результате этих действий получаем два модуля:

### **Модуль PutsTxt2.asm**

```
.386
.model flat,stdcall
puts PROTO:DWORD
.data
; Массив адресов строк
txt      dd s1,s2,s3,0
; Сами строки
s1       db 'Первая строка',0
s2       db 'Вторая строка',0
s3       db 'Третья строка',0

.code
Main proc
    mov esi, offset txt ; p = txt
@For:
    cmp dword ptr [esi], 0 ; *p != NULL?
    je  @AFor ; Выход при обнаружении 0
    ; puts(*p++)
    push [esi] ;
    call puts
    add esi, 4 ; p++
    jmp @For
@AFor:
    mov eax, 0
    ret
Main endp
end Main ; задание стартовой метки программы
```

### **Модуль io.asm**

```
.386
.model flat,stdcall
; Спецификации для функций вывода
.nolist
Include kernel32.inc
```

```

Include user32.inc
Include masm32.inc
Includelib kernel32.lib
Includelib user32.lib
Includelib masm32.lib
.list
.data
; Коды "Перевод строки, Возврат каретки"
nextLine db 10, 13, 0
.code
puts proc, pstr: DWORD
    ; Перекодировка в ASCII
    Invoke CharToOemA, pstr, pstr
    ; Вывод перекодированной строки
    Invoke StdOut, pstr
    ; Переход на следующую строку
    Invoke StdOut, ADDR nextLine
    ret
puts endp
end

```

Чтобы получить возможность отдельно ассемблировать модуль в RadAsm, необходимо, находясь в окне редактирования io.asm в меню Make, выполнить команду Toggle Current As Module. После этого в меню Make станет исполняемой команда Assemble Modules, что мы увидим по изменению света строки команды с бледного на нормальный. Выполнив команду Assemble Modules, мы получим объектный файл io.obj.

Командой Build в меню Make собираем исполняемый файл, который в режиме командной строки запускаем командой putstxt2. Необходимо учитывать, что команда Build запускает компоновку объектных файлов, но не запускает ассемблирование тех модулей, исходный код которых имеет более позднее время модификации, нежели объектный файл. Если вносятся изменения в исходные тексты модулей, то перед командой Build важно не забыть выполнить команду Assemble Modules.

### **Пример 1.11. Проект «Основной модуль + Своя библиотека»**

Для этого проекта нам необходимо выполнить следующее:

- а) создать проект PutsTxtL; поскольку отлаживать ничего не придется, целесообразно выбрать в качестве типа сборки Console Release;
- б) скопировать содержимое файла PutsTxt2.asm в окно редактирования файла PutsTxtL;
- в) вставить рядом со строкой описания процедуры puts строку подключения библиотеки: `Includelib io.lib`; именно это обеспечивает вовлечение библиотеки io.lib в процесс компоновки;
- г) создать саму библиотеку io.lib, выполнив в каталоге, где находится файл io.asm команду  

```
\masm32\bin\lib io.obj /out:io.lib;
```
- д) поместить io.lib либо в каталог проекта PutsTxtL, либо в каталог, путь к которому прописан в диалоге Environment.Lib меню Option;
- е) выполнить сборку проекта PutsTxtL командой Build меню Make.

В случаях, когда разработка библиотеки является целью проекта, целесообразно использовать отдельный проект RadAsm с типом сборки Library.

## **1.4. Вопросы и упражнения**

- 1) Можно ли программу, написанную на языке ассемблера для микроконтроллера ATMega128, оттранслировать и выполнить на персональном компьютере с процессором архитектуры x86?
- 2) Какие особенности процесса ассемблирования позволяют считать компилятор языка ассемблера машинно-ориентированной программой?
- 3) Какие особенности процесса статического профилирования позволяют считать соответствующий профилировщик машинно-ориентированной программой?
- 4) Можно ли при программировании на ассемблере использовать только один тип исходных модулей? Если да, то какой это тип и



возникают ли неудобства от неиспользования других типов исходных модулей?

5) Почему требуется более одного прохода для трансляции ассемблер-программы?

6) Какие особенности процесса отладки заставляют пользоваться как отладочным выводом, так и отладчиком типа OllyDbg?

7) Установите пакет MASM.SDK и выполните действия, описанные в примере 1.6. Создайте сценарий трансляции-компоновки в формате bat-файла, включив в него поддержку просмотра листинга трансляции и параметров файлов, порожденных сценарием.

8) Установите отладчик OllyDbg и повторите примеры 1.6-1.8.

9) Установите пакет RadAsm и настройте его. Повторите примеры 1.9-1.11. Выполните в среде отладчика OllyDbg трассировку любой из сборок программы распечатки слов в двух режимах: покомандная трассировка и трассировка с остановками только перед вызовом процедуры puts.

10) Создайте новый проект MyGroup на основе примера 1.9 таким образом, чтобы первый элемент массива строк содержал наименование вашей студенческой группы и еще 5 элементов содержали имя и фамилию некоторых ее студентов. Выполните отладку программы.

11) Выполните модификацию проекта MyGroup таким образом, чтобы он собирался с библиотекой io.lib аналогично примеру 1.11. Выполните отладку программы.

## **2. Основные элементы языка ассемблера для архитектур x86**

### **2.1. Структура ассемблер-программ**

Ассемблер-программа представляет собой совокупность операторов и комментариев, которые обычно объединяются в смысловые разделы, не обязательно выделяемые специальными операторами. Внутри одного модуля основными разделами являются:

- а) заголовок программы с общим комментарием;
- б) раздел базовых соглашений, описывающих семейство процессоров, правила именования внешних процедур и ограничения набора команд;
- в) раздел внешних описаний, используемых для вовлечения в проект библиотек процедур и макросов;
- г) раздел внутренних описаний (локальных для данного модуля), содержащий обычно определения именованных констант и макроопределения;
- д) раздел декларации данных;
- е) раздел кода программы, т. е. исходный текст исполняемой части;
- ж) директива конца исходного текста программ.

Заголовок программы задается директивой «TITLE текст» и служит для размещения на каждом листе листинга текста заголовка. Традиционно в начале каждого модуля размещают комментарий, содержание которого чаще всего делает ненужным директиву TITLE.

В раздел базовых соглашений обычно включают:

- Директиву ограничения набора команд. При отсутствии этой директивы по умолчанию ограничение соответствует директиве `.8086`, т. е. допускается использовать только команды базового 16-разрядного МП Intel 8086. Директивой `.386` разрешается использовать непривилегированные команды базового МП IA-32 Intel 80386, директивой `.686` разрешается использование непривилегированных команд МП Pentium Pro.

- Директиву задания модели памяти. Для программирования в 32-разрядном режиме обычно используется директива `.model flat`. В этой директиве через запятую могут задаваться: а) соглашение о языке (`STDCALL`, `C`, `BASIC`, `FORTTRAN`, `PASCAL`); б) параметры стека (`NEARSTACK`, `FARSSTACK`).

- Директиву задания размера стека `.STACK` число. По умолчанию стек имеет длину 1024 байта, что может быть недостаточным для процедур с большой глубиной вызова и объемными локальными данными.

Раздел внешних описаний обычно содержит директивы `Include` и `Includelib` с указанием имен файлов определений и библиотек. Сами файлы определений, имена которых обычно имеют расширение `.inc`, являются общими внешними описаниями для нескольких модулей. В шаблоне консольного приложения `ConsoleApp.tpl` интегрированной среды `RadAsm` (`\RadAsm\Masm\Templates\`) содержатся такие директивы внешних описаний:

```
include windows.inc
include kernel32.inc
include masm32.inc
include user32.inc
include C:\MASM32\MACROS\strings.mac
includelib kernel32.lib
includelib masm32.lib
includelib user32.lib
```

Раздел внутренних описаний содержит макроопределения, используемые в текущем модуле. По мере разработки текущий модуль может стать слишком громоздким, что потребует его разделения на несколько модулей. При этом общие для разных модулей спецификации из раздела внутренних описаний целесообразно перенести в `inc`-файл, т. е. сделать их внешними.

Разделы декларации данных начинаются с директив объявления сегментов. Кроме использованной выше директивы `.DATA` широко используется также директива `.CONST` (это защищенный от записи сегмент данных). В данном пособии используются так называемые

«упрощенные директивы сегментации», которых вполне хватает для программирования рассматриваемых задач.

Раздел кода программы начинается с директивы `.CODE`, за которой следуют операторные строки, содержащие операторы машинных команд и макровыводы. Операторы машинных команд группируются в подпрограммы, которые обрамляются директивами заголовка «имя `PROC` . . .» и завершения «имя `ENDP`». Если процедура имеет статические локальные данные, то обычно перед процедурой размещается директива `.DATA` с описанием этих данных, а перед заголовком процедуры размещается директива `.CODE`, обеспечивающая продолжение формирования сегмента кода.

Завершение кода программы задается директивой `END`, за которой может быть стартовая метка программы. В многомодульной программе только в основном модуле имеется директива `END` со стартовой меткой. В остальных модулях директива не должна иметь операнда.

## 2.2. Синтаксис оператора

В общем случае операторная строка имеет такой формат:

[<метка>[:]] [<cmd>[<p1>][,p2]..[,pn]] [; комментарий],

где `cmd` – оператор или директива, `pi` – параметр оператора или директивы, квадратные скобки означают необязательность элемента.

Метка – это последовательность символов из множества {0..9, a..z, A..Z, \_, \$, @, ?}, причем первый символ – "не цифра". Двоеточие ":" отделяет метку только от исполняемой команды.

Метка может быть объявлена явно с помощью директивы `LABEL` в операторной строке следующего формата:

метка `LABEL` тип.

С точки зрения языка ассемблера любое имя является меткой, однако ничто не мешает нам использовать традиционные понятия «имя переменной», «имя константы», «имя процедуры» «точка перехода» и т. п.

## 2.3. Константы и выражения

Целые числа в ассемблере x86 задаются в формате:

[знак] цифры [символ\_системы\_счисления].

В качестве символов систем счисления используются следующие латинские буквы (допустимы как верхний, так и нижний регистры):

**B** – двоичная, **Q** или **O** – восьмеричная, **D** – десятичная, **H** – шестнадцатеричная (если число начинается с букв A..F, то впереди должна быть вставлена цифра 0). Если символ системы счисления отсутствует, то по умолчанию принимается десятичная система.

Например: 42q, 00100010b, 22h, 34d, 34. Все приведенные в этом списке константы задают одно и то же числовое значение. Однобайтное число -1 может быть задано любым из следующих констант: -1, 11111111b, 0FFh, 377q.

Вещественные константы задаются десятичными цифрами в следующем формате:

[знак] цифры. [цифры] [степень].

В качестве степени используется конструкция

E [ + | - ] цифры.

Символьные константы представляются одним символом, заключенным в одинарные или двойные кавычки. Например: 'A', "B".

Строковые константы также заключаются в одинарные или двойные кавычки, при этом внешние кавычки могут обрамлять строку, в которой имеется фрагмент в кавычках иного рода. Например:

"ABC", 'DEF',

'Символ "b" завершает цифры двоичной константы'.

Для строковых констант допустима агрегация групп данных с помощью угловых скобок. Например, строка с встроенными в нее символами перевода строки и возврата каретки может быть определена так:

<"Конец текста", 0Ah, 0Dh, 0>

Константы могут именоваться с помощью директив EQU и “=”. Директива присваивания “=” может именовать только целочисленную константу, причем в программе одно и то же имя может с помощью этой директивы несколько раз переопределяться. Директива EQU может определять имя для любой, даже нецелочисленной константы, однако однажды определенное с ее помощью имя нельзя переопределить.

Например:

```
MaxLen = 35
SizeArray EQU <60.32>
Req1 EQU "Введите число строк: "
Byte EQU <"До свидания", 0Ah, 0Dh, 0>
```

В шестой версии MASM появилась директива присваивания специально для тестовых строк – TEXTEQU. Эта директива допускает переопределение имени.

В ассемблере имеется предопределенное имя “\$”, которое задает константу, равную текущему значению счетчика размещения. Если оператор размещается по адресу 1000, то фигурирование в нем имени “\$” означает использование числа 1000 в качестве значения, заданного этим именем.

Выражение – это связанная операторами совокупность констант. Нужно иметь в виду, что выражение вычисляется ассемблером и не является механизмом вычисления выражений, включающих переменные, изменяющие свои значения во время выполнения программы. Иначе говоря, это есть выражения, вычисляемые на стадии трансляции.

В качестве операторов используются:

- +, -, \*, / – знаки арифметических операций;
- MOD – взятие числа по модулю;
- AND, NOT, OR, XOR – логические поразрядные И, ИНВЕРСИЯ, ИЛИ, ИСКЛЮЧАЮЩЕЕ ИЛИ;
- SHL, SHR – сдвиги влево и вправо;

- Операторы сравнения: EQ – равно, NE – не равно, GT – больше, GE – больше или равно, LT – меньше, LE – меньше или равно;
- ( ) – выделение подвыражения.

Например:

Num = 10

mov eax, 1 SHL Num; занесение в eax числа  $2^{\text{Num}}$

mov edx, Num MOD 4; занесение в edx остатка от деления Num на 4

mov ecx, M1 OR M2; занесение в ecx объединения бит-масок

mov ebx, M1 AND M2; занесение в ebx пересечения бит-масок

mov edi, M1 XOR M2; занесение в edi маски M1, часть разрядов которой  
; инвертирована единичными битами маски M2

lea esi, arr[3\*4]; занесение в esi адреса &arr[3]

В ассемблере имеется особый вид выражений, называемых адресными. Эти выражения служат для вычисления адресов данных. Примером адресного выражения служит правый операнд команды lea в приведенном выше примере. Более детально адресные выражения будут рассмотрены в следующей главе при описании функциональной организации архитектуры x86, а также в главе 6, где рассматриваются массивы.

## 2.4. Декларация данных

Данные объявляются с помощью директив со следующей структурой:

*[имя] директива инициализатор[, инициализатор]...*

Директива задает тип данных, а список инициализаторов определяет число элементов и, возможно, их стартовые значения. Типы могут быть базовыми и сконструированными пользователем. На практике распространены два формата задания базового типа: полное имя типа и имя в формате Dx, где x – символ длины элемента данных. Перечень базовых типов приведен в таблице 2.1.

В качестве инициализаторов могут выступать константы, имена меток, символ “?”, соответствующий неинициализированному элементу

данных, а также конструкция *размер DUP (инициализатор)*, которая обеспечивает задание массива с указанным размером.

Для задания строк текста используется директива DB/ BYTE, инициализатором которой могут быть коды символа, символьные и строковые константы.

Таблица 2.1. Базовые типы MASM32

Описание типа / аналог в языке Си	Имя типа	Формат Dx
Однобайтное беззнаковое целое / unsigned char	BYTE	DB
Однобайтное целое со знаком / signed char	SBYTE	DB
2-байтное беззнаковое целое / unsigned short	WORD	DW
2- байтное целое со знаком / short	SWORD	DW
4-байтное беззнаковое целое / unsigned long   far * в IA-16	DWORD	DD
4- байтное целое со знаком / long	SDWORD	DD
6-байтное целое / far * в IA-32	FWORD	DF
8-байтное целое / long long	QWORD	DQ
10-байтное целое	TBYTE	DT
4-байтное вещественное / float	REAL4	DD
8-байтное вещественное / double	REAL8	DQ
10-байтное вещественное / long double	REAL10	DT

### Пример 2.1. Реализация объявлений данных на ассемблере

Пусть имеются такие декларации данных на языке Си:

```
char ch1='X', ch2, ach1[10], ach2[]="hello!\n",
    ach3[]= {'a', 'b', 'c'}; *pch1="abc",
    *pch2=ach2;
short sh1, sh2=-12, ash[]={1,2,4,5}, *psh = ash;
long lo1=3, lo2, alo[]={1,-2,4,5}, *plo = alo;
```

На ассемблере IA-32 аналогичные объявления будут выглядеть так:

```
.data
ch1      db 'X'
ch2      db ?
ach1     db 10 dup(?)
ach2     db "hello!", 0Ah, 0
ach3     db "abc"
```



```

LenAch3 = $ - ach3
pch1    dd s1
labcb   db "abc", 0
pch2    dd ach2
sh1     dw ?
sh2     dw -12
ash     dw 1, 2, 4, 5
LenAsh = ($ - ash) / 2
psh     dd ash
lo1     dd 3
lo2     dd ?
alo     dd 1, -2, 4, 5
plo     dd alo

```

После объявления массива `ach3` и массива `ash` показано, как можно сформировать именованную константу, равную длине массива в количестве элементов. Естественно, что конструкция со счетчиком размещения '\$' должна следовать сразу за объявлением массива, длина которого определяется. Если, например, поместить сразу за строкой с меткой `ach3` какое-то определение данных, то константа `LenAch3` будет определена не в соответствии с длиной строки `ach3`.

## 2.5. Элементы макропрограммирования

Хорошо продуманная система макроопределений и подпрограмм превращает язык ассемблера из низкоуровневого в высокоуровневый. Точнее, в разноуровневый, в котором возможности уровня машинных команд сочетаются с мощностью библиотек процедур и макросов, благодаря чему обеспечивается высокое быстродействие программ при относительно низких затратах времени на программирование.

Наиболее активно используются следующие механизмы генерации:

- а) включение или не включение фрагментов текста на основе директив условной трансляции;
- б) макрорасширение на основе макроопределений;
- г) использование встроенных в MASM32 макросов.
- в) цикловая генерация фрагментов исходного кода.

### 2.5.1 Директивы условной трансляции

Директивы условной трансляции обеспечивают управление включением или не включением фрагмента ассемблер-программы в процесс трансляции. Существуют три формата директив:

1) Включение группы операторов *OpGroup* в трансляцию при выполнении условия задается такой конструкцией:

```
IFxxx ExprPar
    OpGroup
ENDIF
```

В качестве *ExprPar* выступает либо логическое выражение, либо один или два параметра. Встретив такую конструкцию, ассемблер включает в процесс трансляции фрагмент *OpGroup* только при условии истинности логического выражения или соответствия значений параметров условию xxx. В основе логических выражений находятся операторы сравнения EQ, NE, LT, LE, GT, GE, рассмотренные в подразделе 2.3. Слева и справа могут быть любые допустимые выражения, про которые необходимо помнить, что они работают над константами и значениями меток, но не над содержимым ячеек памяти.

2) Включение одной из двух ветвей

```
IFxxx ExprPar
    OpGroupThen
ELSE
    OpGroupElse
ENDIF
```

3) Включение одной из нескольких ветвей

```
IFxxx ExprPar1
    OpGroup1
ELSEIFxxx ExprPar2
    OpGroup2
...
ELSE
    OpGroupElse
ENDIF
```

Ассемблер включает в процесс трансляции одну из двух ветвей в зависимости от истинности условия.

Перечень директив условной трансляции приведен в таблице 2.2.

Таблица 2.2. Директивы условной трансляции

Формат директивы	Описание условия
IF <i>expr</i>	ЕСЛИ логическое выражение <i>expr</i> истинно
IFE <i>expr</i>	ЕСЛИ логическое выражение <i>expr</i> ложно
IFDEF <i>id</i>	ЕСЛИ идентификатор <i>id</i> определен
IFNDEF <i>id</i>	ЕСЛИ идентификатор <i>id</i> не определен
IFB < <i>id</i> >	ЕСЛИ параметр макроса <i>id</i> пуст
IFNB < <i>id</i> >	ЕСЛИ параметр макроса <i>id</i> не пуст
IFIDN < <i>s1</i> >, < <i>s2</i> >	ЕСЛИ строки <i>s1</i> и <i>s2</i> полностью идентичны
IFIDNI < <i>s1</i> >, < <i>s2</i> >	ЕСЛИ строки <i>s1</i> и <i>s2</i> идентичны без учета разницы регистров символов (строчные/прописные)
IFDIF < <i>s1</i> >, < <i>s2</i> >	ЕСЛИ строки <i>s1</i> и <i>s2</i> различны
IFDIFI < <i>s1</i> >, < <i>s2</i> >	ЕСЛИ строки <i>s1</i> и <i>s2</i> различны без учета разницы регистров символов (строчные/прописные)
EXITM	Завершение трансляции макроопределения. Оставшаяся часть до ENDM транслироваться не будет

### Пример 2.2. Директива условной трансляции отладочного вывода

Пусть требуется организовать такую вставку отладочного вывода строки sLog в исходный код, которая имеет место только при определенном идентификаторе FlDebug. Это достигается такой директивой условной трансляции:

```
IFDEF FlDebug
    EXITM
    invoke StdOut, sLog
ENDIF
```

Чтобы задействовать эту директиву, нужно до ее текста вставить в программу определение идентификатора FlDebug, например:

```
FlDebug = 1
```

Чтобы исключить строку вывода sLog из трансляции, достаточно закомментировать объявление FIDebug:

```
; FIDebug = 1
```

### Пример 2.3. Генерация обращения к операндам различной длины

Пусть требуется организовать обнуление переменной, адресуемой указателем Point, при условии, что размер переменной в байтах представлен константой VSize и может иметь значения 1, 2, 4, 8. Это достигается такой директивой условной трансляции:

```
mov eax, Point
IF VSize EQ 1
    mov byte ptr [eax], 0
ELSEIF VSize EQ 2
    mov word ptr [eax], 0
ELSEIF VSize EQ 4
    mov dword ptr [eax], 0
ELSE VSize EQ 8
    mov dword ptr [eax], 0
    mov dword ptr [eax+4], 0
ENDIF
```

В последней ветви приведенной выше директивы условной трансляции IF для обнуления потребовались два оператора пересылки, поскольку в режиме IA-32 одна команда обрабатывает только 4 байта, а Point в этой ветви указывает на 8-байтный операнд.

### 2.5.2 Макроопределения и макрорасширения

Декларация макросов имеет следующий вид:

```
имя      MACRO [ список формальных параметров ] ...
          OpGroup
ENDM
```

Для формирования макрорасширения необходимо вставить в программу так называемый макровывод, который имеет формат:

*имя\_макроса список\_фактических\_параметров*

Элементы списка фактических параметров разделяются запятыми. Формальный параметр может иметь, кроме базовой части, еще и постфиксную в формате *имя[:постфикс]*. Если *постфикс* равен REQ, то

фактический аргумент обязателен в строке вызова макрокоманды. Если *постфикс* представляет собой конструкцию «=*строка\_по\_умолчанию*>», то при отсутствии фактического аргумента в строке вызова макрокоманды соответствующий формальный аргумент в макрорасширении будет заменяться строкой после знака равенства “=”.

#### **Пример 2.4. Макросы сохранения данных в стеке**

В программах часто требуется сохранять несколько операндов в стеке. Можно создать несколько макросов с разным числом параметров:

```
push2 macro x1, x2
        pushd    x1
        pushd    x2
endm

push3 macro x1, x2, x3
        pushd    x1
        pushd    x2
        pushd    x3
endm
```

Примеры использования этих макросов:

```
push2 ecx, 5
push3 abc, esi
```

Пользуясь директивой условной трансляции IFNB, проверяющей свой параметр на пустоту, можно создать один макрос, способный иметь разное число параметров, например:

```
push2_4 macro x1:REQ, x2:REQ, x3, x4
        pushd    x1
        pushd    x2
        ifnb <x3>
            pushd    x3
        endif
        ifnb <x4>
            pushd    x4
        endif
endm
```

Пример вызова:

```
push2_4 i, [ebx+edi+4]
push2_4 5, edx, tbl[4*edi], 0
```

Элементом списка фактических параметров может быть константа, константное выражение, имя переменной, адресное выражение. Если внутри текста одного фактического параметра нужно использовать разделители, которые могут быть интерпретированы макропроцессором как разделитель фактических параметров, то этот параметр либо обрамляется угловыми скобками <..>, либо разделители фильтруются символом “!”.

Если требуется сгенерировать подстроку фактического параметра с текстом числового выражения, то это числовое (константное) выражение вставляется в текст фактического параметра с префиксом “%”.

Если имя формального параметра в тексте макроопределения непосредственно примыкает к какой-то строке, то используется префикс “&”, чтобы макропроцессор мог выделить формальный параметр и заменить его фактическим параметром.

#### **Пример 2.5. Генерация имени переменной на основе параметра**

Пусть требуется разработать макрос сохранения регистров в переменных, в имени которых сначала идет “buf”, а затем имя регистра. Имя регистра должно фигурировать в качестве фактического параметра макровывода. Макрос имеет вид:

```
RGStore macro reg
    mov  buf&reg, reg
endm
```

Если в приведенном макроопределении убрать символ ‘&’, то при макровыводе RGStore EDX макропроцессор вместо правильного оператора `mov bufEDX, EDX` сгенерирует оператор `mov bufreg, EDX`.

Внутри макроопределения могут использоваться макровыводы.

#### **Пример 2.6. Макрос с внутренними макровыводами**

Пусть требуется разработать макрос RG4Store сохранения четырех регистров в переменных аналогично макросу RGStore примера 2.5. Использование вложенного макроса буферной памяти RGStore дает следующее макроопределение:

```

RG4Store macro p1, p2, p3, p4
    RGStore p1
    RGStore p2
    RGStore p3
    RGStore p4
endm

```

### 2.5.3 Использование меток и объявлений данных в макросах

Если в макросе возникает необходимость организовать ветвление, то команда перехода должна содержать метку. Если это будет обычная метка, то при втором макровывозе возникнет ситуация дублирования метки, что отразится в сообщениях об ошибках при ассемблировании. Ассемблер MASM может генерировать уникальную метку. Для этого необходимо сразу после заголовка описать все используемые метки с помощью директивы LOCAL.

#### Пример 2.7. Локальная метка в макроопределении

Пусть требуется в нескольких местах программы формировать в регистре EAX минимум из двух целых чисел со знаком. В языке Си это можно было бы записать таким макроопределением:

```
#define min(x,y) (x<y)? x : y
```

На ассемблере возможность уже нахождения одного из параметров в регистре EAX должна явно быть учтена в макросе:

```

; Формирование минимума двух целых чисел в EAX
; Само значение EAX может быть только в первом параметре
IMIN32 MACRO p1, p2
    LOCAL LDONE
    IFDIFI <p1>, <eax>
        mov    eax, x
    ENDIF
    cmp    eax, y
    jl     LDONE
    mov    eax, y
LDONE:
ENDM

```

Пусть в разрабатываемой программе возникает потребность формировать в EAX минимум из четырех 32-разрядных целых чисел. Использование

возможности включать макровыводы в макроопределение дает такое макроопределение:

```
; Формирование минимума четырех целых чисел в EAX
; Само значение EAX может быть только в первом параметре
IMIN32_4 MACRO p1, p2, p3, p4
    IMIN32 p1, p2
    IMIN32 eax, p3
    IMIN32 eax, p4
ENDM
```

Пусть в программе фигурируют следующие три макровывода:

```
IMIN32 15, EDX
IMIN32 EAX, [ebp+4]
IMIN32_4 EDX, i, abc, tbl[8]
```

В листинге трансляции мы можем увидеть следующие продукты макрогенерации:

```
IMIN32 15, EDX
00000000 B8 0000000F      1      mov eax, 15
00000005 3B C2           1      cmp eax, EDX
00000007 7C 02           1      jl  ??0000
00000009 8B C2           1      mov eax, EDX
0000000B                1      ??0000:
IMIN32 EAX, [ebp+4]
0000000B 3B 45 04          1      cmp eax, [ebp+4]
0000000E 7C 03           1      jl  ??0001
00000010 8B 45 04          1      mov eax, [ebp+4]
00000013                1      ??0001:
IMIN32_4 EDX, i, abc, tbl[8]
00000013 8B C2             2      mov eax, EDX
00000015 3B 05 00000005 R 2      cmp eax, i
0000001B 7C 05             2      jl  ??0002
0000001D A1 00000005 R 2      mov eax, i
00000022                2      ??0002:
00000022 3B 05 00000001 R 2      cmp eax, abc
00000028 7C 05             2      jl  ??0003
0000002A A1 00000001 R 2      mov eax, abc
0000002F                2      ??0003:
0000002F 3B 05 00000011 R 2      cmp eax, tbl[8]
```



```

00000035 7C 05          2          jl  ??0004
00000037 A1 00000011 R    2          mov eax, tbl[8]
0000003C          2      ??0004:

```

При обработке первого макровывоза макропроцессор породил 4 машинных команды и уникальную метку ??0000. При обработке второго макровывоза порождены метка ??0001 и всего три команды, поскольку первый параметр в макровывозе равен EAX, что обнаруживает директивой условной трансляции IFDIFI <p1>,<eax>. При этом директива IFDIFI игнорирует разницу в строчном и прописном написании EAX и eax. Третий макровывоз породил еще три уникальные метки в соответствии с тем, что внутри макроопределения IMIN32\_4 находятся три макровывоза IMIN32.

Локальные метки удобны также для генерации текстовых данных, инициализированных константами. В примере 1.5 для реализации объявления трехстрочной строки пришлось задействовать имена S1, S2, S3 для меток декларации данных самих строк и для инициализации адресов в массиве TXT. Ясно, что это неудобно, поскольку добавляет к пространству имен метки, не используемые за пределами объявления текста. В результате повышается вероятность дублирования имен. Использование локальных меток в макросах позволяет избавиться от такого неудобства.

### **Пример 2.8. Макрос специфицирования трехстрочного текста.**

Текст макроопределения, которое порождает результат, аналогичный объявлению на языке Си массива из трех строк, имеет вид:

```

MakeText3 macro tName, str1,str2,str3
    local l1,l2,l3
    .data
    tName dd  l1,l2,l3,0
    l1     db str1, 0
    l2     db str2, 0
    l3     db str3, 0
endm

```

Макровывоз должен содержать только константные строки:

```

MakeText3 txt "строка1",\
<"строка",9,"с табуляцией">,\

```

<"строка",10,13,"продолжение на следующей строке">

В приведенном макровыводе с помощью угловых скобок в строковые константы вставляются спецсимволы горизонтальной табуляции (код 9), перевода строки (код 10) и возврата каретки (код 13).

#### 2.5.4 Макросы организации записей

Объединение совокупности элементов данных под одним именем в языках высокого уровня называется записью, или структурой. В ассемблере MASM такая возможность обеспечивается специальным встроенным макросом STRUCT. Формат объявления структуры имеет вид:

```
ИмяСтруктуры STRUCT
                        ОписаниеПолей
ИмяСтруктуры ENDS
```

При описании полей используются директивы объявления данных в формате «Имя Тип Инициализатор». Декларация структуры в отличие от декларации данных не порождает данные в сегменте data, а только специфицирует шаблон, используемый ассемблером при обработке объявлений собственно данных, которые будем называть записями. При этом имеются возможности как повторять в создаваемых записях данные инициализации из шаблона, так и устанавливать иные значения.

Обращение к полям записи осуществляется аналогично языкам высокого уровня через конструкцию ИмяЗаписи.ИмяПоля. Возможно использование регистра в качестве указателя на запись. В этом случае используется конструкция (ИмяСтруктуры ptr [регистр]).ИмяПоля.

#### Пример 2.9. Объявление структуры и записей

Пусть, например, необходимо обрабатывать серии данных, вычисляя минимальное, максимальное, среднее значения в ходе поступления

; Объявление структуры для статистической обработки серии данных

```
Stat    STRUCT
        npar db 8 dup(0)    ; имя параметра
        min  dd 7FFFFFFFh   ; минимум
        max  dd 0           ; максимум
        aver dd 0           ; среднее
```

```

        sum    dq 0                ; сумма
        len    dd 0                ; длина серии
Stat ENDS
; Объявление двух записей со структурой Stat
st1     Stat <"st1">
st2     Stat <"st2",3,5,40,4,10>

```

При трансляции эти два объявления порождают в сегменте данных область памяти, представленную следующим листингом:

```

00000000 73 74 31                st1  Stat <"st1">
        00000005 [
        00
        ] 7FFFFFFF
        00000000
        000000000000000000
        00000000
        00000000
00000020 73 74 32                st2  Stat <"st2",3,5,40,4,10>
        00000005 [
        00
        ] 00000003
        00000005
        000000000000000028
        00000004
        0000000A

```

Приведенный листинг показывает, что из объявления записи st1 в запись попали только три байта имени “st1” – остальные данные взяты из шаблона структуры. В объявлении записи st2 все поля проинициализированы из ее декларации, а не из шаблона структуры.

Примеры обращения к полям записей представлены ниже:

```

mov eax, st1.min
mov edx, st2.max
mov edi, offset st2
mov ecx, (Stat ptr [esi]).aver

```

## 2.6. Вопросы и упражнения

1) Пусть следующие две строки сегмента данных протранслированы по адресу 500h:

```
V1 dw 3, 8
```

```
V2 dw 5
```

По какому адресу в памяти будет размещаться переменная V2?

Какое значение окажется в регистре EAX после выполнения в сегменте кода оператора MOV EAX, V1+V2?

2) Создать в среде RadAsm проект sample2 и вставить в него с помощью позиции меню Project => Add Item => New File файл с именем sample2.asm. Вставить в этот файл строки:

```
; Эксперименты с объявлениями данных и макросами
.386
.model flat,stdcall
.data
a db 0
.code
mov al, a
end
```

Выполнить ассемблирование, открыть через меню File => Open File файл листинга sample2.lst. Включить через меню View => Tab Select отображение закладок файлов sample2.asm и sample2.lst для поддержки экспериментирования в режиме «подглядывания» в файл листинга после каждого ассемблирования.

3) Выполнить модификацию исходного текста из предыдущего упражнения таким образом, чтобы в сегменте данных появились данные различной длины – байты (a8, b8), слова (a16, b16), двойные слова (a32, b32), строки (s20, s30). Добавить в сегмент данных несколько указателей, инициализированных адресами переменных, например: pa8, pb16, pa32, ps30. Выполнить трансляцию и проанализировать листинг в части адресов размещения данных и инициализированных значений указателей.

4) Разработать блок спецификации данных на ассемблере, который аналогичен следующему объявлению на языке Си:

```

char leftParent = '(', rightParent = ')',
    expr[] = "(a+b)*(c-d)", *pExpr = expr;
char tic_tac_toe[3][3]
    = { { 'X', '0', '0' },
        { '0', 'X', '0' },
        { '0', '0', 'X' } };
char *ttt[3] = { "X00", "0X0", "00X" };

```

Выполнить ассемблирование. Добиться отсутствия сообщений об ошибках и проанализировать в листинге размещение объявленных данных.

5) Разработать блок спецификации данных на ассемблере, который аналогичен следующему объявлению на языке Си:

```

short v16, av16[5], *pv16 = av16 + 2;
long v32, av32[4] = {-1, 3, 10, -3}, *pv32=av32+1,
    mv32[2][3] = {{20,30,40},{50,60,70}};

```

Выполнить ассемблирование и анализ размещения объявленных данных в листинге.

6) Разработать макроопределение ClearVar, которое реализует очистку переменной на основе кода примера 2.3. В качестве параметров макроопределения должны фигурировать адрес переменной Point и размер VSize. Разместить в сегменте кода набор макровыводов, охватывающий все варианты размеров данных. Выполнить ассемблирование. Добиться отсутствия сообщений об ошибках и проанализировать в листинге результаты макрогенерации.

7) По аналогии с макросов push2\_4 примера 2.4 разработать макрос сохранения в стеке от 3 до 6 параметров. Разработать несколько примеров макровыводов, которые содержат не только макровыводы с числом параметров от 3 до 6, но и макровыводы с иным числом параметров. Проанализировать реакцию ассемблера на нехватку или избыток параметров в макровыводе. Проанализировать результат обработки макропроцессором правильных макровыводов.

8) Вставить в модуль экспериментирования sample2.asm макроопределения RGStore и RG4Store из примеров 2.5 и 2.6. Создать в сегменте данных переменные bufEAX, bufECX, bufEDX, bufEBX, bufESI,

bufEDI, bufEBP. Вставить в сегмент кода несколько макровыводов макросов RGStore и RG4Store. Проанализировать результаты макрогенерации.

9) По аналогии с макросами из примера 2.7 разработать макросы IMAX8 и IMAX8\_3, первый из которых формирует в регистре al максимум из двух 8-разрядных операндов, а второй – максимум из трех 8-разрядных операндов. Вставить в сегмент несколько макровыводов этих макросов, при необходимости создавая байтовые переменные в сегменте данных для использования соответствующих имен в качестве фактических параметров макровыводов. Проанализировать результаты макрогенерации.

10) Пользуясь директивами условной трансляции, создать на базе примера 2.8 макроопределение MakeTex2\_5, которое способно генерировать тексты длиной от 2 до 5 строк в зависимости от числа фактических параметров макровыводов. Вставить в сегмент данных несколько макровыводов и проанализировать результаты макрогенерации.

11) Поместить в файл sample2.asm макроопределение из примера 2.9. Поместить в сегмент данных несколько переменных со структурой Stat и проанализировать продукты макрогенерации. Поместить в сегмент данных группу операторов пересылки mov, в которой фигурируют все поля записей. Написать код копирования из одной записи со структурой Stat в другую через обращения ко всем полям. Проанализировать листинг трансляции.

12) Разработать спецификацию структуры PERSON, в полях которой представляются числовой табельный номер, фамилии, имена, отчество, день, месяц и год рождения персоны. Разместить в сегменте данных объявления переменных pers, pers1, pers2, pers3, инициализируя конкретными данными все, кроме первой. Предполагая, что подпрограмма persprint1 распечатывает сведения, хранимые в записи pers, организовать на основе ее вызова распечатку сведений из записей pers1, pers2 и pers3. Для предотвращения сообщения об ошибке в связи с неопределенностью имени persprint1 использовать решение для функции puts из примера 1.10, в котором описана сборка двухмодульной программы распечатки текста.

13) Модифицировать процесс распечатки сведений pers1, pers2 и pers3 из предыдущего упражнения в предположении, что имеется подпрограмма persprint2, в которую через регистр esi передается адрес области памяти с распечатываемой записью.

14) Разместить в сегменте данных программы, полученной при выполнении упражнения 11, массив указателей на записи pers1, pers2, pers3. Модифицируйте процесс распечатки, запрограммированный при выполнении предыдущего упражнения таким образом, чтобы новый процесс содержал цикл сканирования массива указателей на записи по аналогии с примером 1.9.

15) Добавьте в программу, полученную при выполнении предыдущего упражнения, фрагмент кода предварительной сортировки набора записей по инвентарному номеру перед распечаткой. В ходе сортировки переставляться местами должны элементы массива указателей, а не содержимое сортируемых записей.

### 3. Функциональная организация процессоров с архитектурой Intel x86

#### 3.1. Программно-доступные компоненты и методы адресации

Основными программно-доступными компонентами I80x86, задаваемыми в аргументах командных строк ассемблер-программы, являются регистры общего назначения (РОН), адресные регистры (базовые и индексные), сегментные регистры, счетчик (указатель) команд, регистр флагов, ячейки памяти.

Регистры адресуются зачастую неявно, что закрепляет за ними определенную специализацию.

Некоторые РОН являются частями других РОН. Например,  $AX=AH\_AL=EAX[15:0]$ ,

где  $r_1\_r_2$  – регистровая пара, в которой  $r_1$  – старшая часть,

$r[m:n]$  – диапазон разрядов, в котором  $m$  – номер старшего разряда.

Ниже приводится список регистров общего назначения и адресных регистров. Первыми в списке указаны имена 32-разрядных регистров, которые могут использоваться только в 32-разрядном режиме работы процессора либо в 16-разрядном, но в командах с префиксом изменения длины операнда SIZ. Этот префикс в большинстве случаев вставляется ассемблером автоматически на основе факта фигурирования 32-разрядного регистра среди операндов.

EAX/AX/AH/AL РОН, часто неявно адресуемые как аккумулятор.  
 $AX=AH\_AL=EAX[15:0]$ .

ECX/CX/CH/CL РОН, используемые также как неявно адресуемые счетчики числа сдвигов в сдвиговых командах и как счетчики длины цепочки данных в строковых командах.  $CX=CH\_CL=ECX[15:0]$ .

EDX/DX/DH/DL РОН, используемые также для представления неявно адресуемой старшей части операнда повышенной точности и адреса порта ввода/вывода.  
 $DX=DH\_DL=EDX[15:0]$ .



EBX/BX/BH/BL	РОН, а также базовый регистр, используемый через коды методов адресации. $BX=BH\_BL=EBX[15:0]$ .
ESP/SP	Указатель стека. В большинстве случаев адресуется неявно в командах организации подпрограмм и прерываний. $SP=ESP[15:0]$ .
EBP/BP	Базовый регистр, используемый обычно для адресации локальных данных в стеке. $BP=EBP[15:0]$ . Имеет важную особенность – фигурирование этого регистра в адресном выражении заставляет процессор использовать по умолчанию сегментный регистр SS.
ESI/SI	Индексный регистр/РОН. В строковых командах используется неявно как указатель источника данных. $SI=ESI[15:0]$ .
EDI/DI	Индексный регистр/РОН. В строковых командах используется неявно как указатель приемника данных. $DI=EDI[15:0]$ .
CS	Сегментный регистр кода. Используется при получении линейного адреса машинной команды в процессе ее выборки из памяти.
DS	Сегментный регистр данных, который используется по умолчанию для адресации данных в памяти.
SS	Сегментный регистр стека, который используется для адресации данных в стеке.
ES, FS, GS	Дополнительные сегментные регистры, которые используются для адресации данных, расположенных в нескольких сегментах памяти.

Счетчик команд IP/EIP не фигурирует в аргументах машинных команд и его программная доступность ограничивается командами управления порядком выполнения программы, организации подпрограмм и прерываний.

Регистр флагов RF хранит признаки результата выполненной операции и флаги управления процессом. При программировании прикладных программ используются следующие основные флаги:

CF – признак переноса/заема (Carry Flag); в командах арифметики имеет смысл беззнакового переполнения; в командах сдвига – значение выдвигаемого бита;

PF – признак четности числа единиц (Parity Flag);

AF – признак переноса/заема из младшего полубайта для десятичной арифметики (Auxiliary Flag);

ZF – признак нулевого результата (Zero Flag);

SF – признак отрицательного результата – копия знакового разряда результата (Sign Flag);

OF – признак переполнения результата (Overflow Flag); в большинстве случаев единица в этом признаке указывает на то, что знак результата арифметической операции ложен – например, сложение двух положительных байтов 40h и 50h (десятичные 64 и 80) дает байт 90h, единичный старший разряд которого фиксируется в признаке SF как признак отрицательного результата;

TF – флаг разрешения трассировки (Trap Flag, при TF=1 после выполнения команды происходит прерывание INT 1);

IF – флаг разрешения аппаратных прерываний (Interrupt-enable Flag);

DF – флаг управления направлением автоиндексации в строковых командах (Direction Flag, при DF=0 – автоинкремент индексного регистра, при DF=1 – автодекремент).

Способы адресации операндов задаются в операторных строках ассемблера программы на основе следующих основных правил:

- непосредственный операнд задается константой, идентификатором константы или константным выражением;
- регистровая адресация задается либо неявно, либо представляется своим именем, либо именем, значение которого сопоставлено с регистром через директивы прямого присваивания;

- память адресуется через адресное выражение, перед которым может фигурировать описатель типа указателя (byte ptr, word ptr, dword ptr и т. п.); описатель может быть опущен, если контекст машинной команды однозначно определяет тип указателя.

Для представления основных адресных выражений будем использовать обозначения: *const* – константа или константное выражение (выражение, значение которого известно до выполнения программы, очень часто это метка декларации данных), *base* – базовый регистр, *ind* – индексный регистр, *scale* – масштаб индекса из множества {1,2,4,8}; *ind<sub>s</sub>* – конструкция, представляющая либо индексный регистр *ind*, либо произведение *scale\*ind*; *abs* – метка декларации данных, представляющая абсолютный адрес операнда; символ “/” – разделитель различных вариантов выражений. Основные варианты адресных выражений имеют вид:

*abs* / [*abs*] / *abs+const* / [*abs+const*] / *const[abs]* / [*const*] – выражения абсолютного адреса, где *abs* – имя метки декларации данных, *const* – константное выражение;

[*base*] / [*ind<sub>s</sub>*] – адрес операнда находится в регистре; это косвенно-регистровая адресация, которая соответствует конструкции \*point языка Си в случае, когда указатель point реализован через регистр;

[*base+const*] / [*ind<sub>s</sub>+const*]: / *const[base]* / *const[ind<sub>s</sub>*] – адрес вычисляется как сумма содержимого возможно масштабированного регистра и константного смещения; позволяет реализовать: а) такое обращение к полю записи через указатель согласно выражению point->fld, при котором регистр содержит указатель, а const является смещением от начала записи к полю fld; б) такое обращение к конкретному элементу массива point[ind], при котором массив задан указателем в регистре, а константное смещение равно произведению индекса на размер элемента; в) такое обращение к элементу массива arr[ind], при котором смещение задает адрес статического массива, а в регистре содержится произведение индекса на размер элемента; в случае использования размера элемента из множества {1,2,4,8} эффективно использовать конструкцию [*scale\*ind+const*], в которой индексный регистр содержит само значение индекса;

$[base+ind_s] / [base][ind_s]$  – адрес вычисляется как сумма содержимого двух регистров, один из которых может масштабироваться; один регистр может адресовать динамический массив, а второй – смещение к элементу массива, равное произведению индекса на размер элемента (размер может быть задан величиной *scale*); первый регистр может содержать базовый адрес строки матрицы, а второй регистр – либо смещение к *j*-му элементу строки, либо номер клетки матрицы при задании размера числа в клетке через *scale*;

$[base+ind_s+const] / const[base][ind_s] / [base][ind_s+const]$  – адрес вычисляется как сумма константного смещения и содержимого двух регистров, один из которых может быть масштабирован; в случае обращения  $arr[i][j]$  смещение может задавать адрес статического массива *arr*, а регистры – смещения к строкам и столбцам; в случае  $point \rightarrow arr[i]$  смещение может адресовать поле *arr* записи, а регистры задавать соответственно адрес записи и смещение к *i*-му элементу внутреннего массива записи; в случае  $point[i].fld$  смещение адресует поле *i*-го элемента массива записей, заданного адресом в *base*, причем регистр *ind* может содержать смещение к *i*-му элементу или собственно значение *i*, если размер записи находится в множестве {2,4,8}; в случае обращения к элементу двумерного статического массива записей  $arrrec[i][j].fld$  *const* представляет сумму базового адреса *arrrec* и смещения к полю *fld* от начала записи, а регистры задают смещения к *i*-й строке и *j*-му матрицы записей *arrrec*.

В 16-разрядном режиме работы процессора  $base \in \{BX, BP\}$ ,  $ind \in \{SI, DI\}$ , причем недопустимо масштабирование индексного регистра и адресное выражение  $[BP]$ . В 32-разрядном режиме базовыми могут быть любые 32-разрядные регистры, однако недопустима косвенно-регистровая адресация  $[EBP]$  и  $[ESP]$ , индексными могут быть все регистры, кроме *ESP*.

Адресное выражение задает механизм вычисления так называемого эффективного адреса *EA*. В 16-разрядном режиме эффективный адрес задает адресное пространство всего в 64 килобайта. Адресное пространство эффективного адреса *EA* в 32-разрядном режиме имеет размер в 4 гигабайта.

Эффективный адрес преобразуется после своего формирования согласно адресному выражению в линейный адрес, который, в свою

очередь, преобразуется в физический адрес через дополнительный механизм страничного преобразования. Для выполнения лабораторных работ по машинно-ориентированному программированию достаточно знать правила формирования линейного адреса. Этот адрес формируется как сумма ЕА и базового адреса ВА сегмента памяти. Это позволяет обращаться к памяти с адресным пространством, превосходящим по размеру адресное пространство эффективного адреса ЕА.

В реальном режиме базовый адрес формируется по выражению:

$$BA = 16 * SR,$$

где SR – сегментный регистр. Любой сегментный регистр является 16-разрядным, поэтому линейный адрес  $LA = BA + EA$  в реальном режиме является 20-разрядным, что дает адресное пространство линейного адреса размеров в 1 мегабайт.

В защищенном режиме базовый адрес сегмента находится в дескрипторе сегмента, также связанного с сегментным регистром. В обоих режимах сегментный регистр задается либо соглашениями по умолчанию, либо явно через конструкцию вида «SR:» перед адресным выражением, например, ES:[BX], CS:[ABC]. Такая конструкция порождает в машинной программе так называемый префикс замены. При обработке данных наиболее важными являются следующие соглашения об использовании сегментных адресов по умолчанию:

а) если в адресном выражении фигурирует регистр BP либо EBP, либо ESP, то используется сегментный регистр SS;

б) для строковых команд операнд-источник адресуется через DS:SI либо DS:ESI, а операнд-приемник – через ES:DI либо ES:EDI, причем только DS может быть заменен на другой сегментный регистр через префикс замены;

в) в других случаях используется регистр DS.

При программировании на ассемблере в 32-разрядном режиме доминирует так называемая плоская модель памяти FLAT, которая избавляет программиста от необходимости манипулировать сегментными регистрами.

При трансляции ассемблер-программы адресные выражения чаще всего трансформируются в содержимое адресных полей машинной команды. Иногда адресация в самой машинной команде задается неявно – через ее код операции. Например, при трансляции оператора `mov eax, 5` порожденная машинная команда не будет содержать адресного поля с номером регистра `eax`. Особый случай неявной адресации – стековая адресация. Команды загрузки данных в стек `push` и извлечения из стека `pop` неявно используют указатель стека `ESP/SP`, который всегда модифицируется в ходе выполнения этих команд. Аналогичная модификация происходит и при выполнении команд вызова подпрограмм и возврата из них. Указанное обстоятельство важно учитывать при отладке программы.

## **3.2. Система команд**

### **3.2.1 Соглашения по описанию системы команд**

При описании системы команд приняты следующие обозначения:

а) *v* – константа или константное выражение (значение метки – самый частый случай константного выражения);

б) *v8/v16/v32/v48/v64* – константы или константные выражения, для которых определена соответствующая разрядность; аналогично определяются разрядности для операндов других типов;

в) *s* – операнд-источник, который может находиться в регистре, памяти или команде (непосредственный операнд);

г) *d* – операнд-приемник, который может находиться в регистре или памяти;

д) *r* – регистровый операнд;

е) *m* – операнд в памяти;

ж) информация о признаках результата представляется в последнем предложении спецификации команды списком флагов, в котором равенства вида `CF=0` и `CF=1` означают фиксацию соответствующих констант во флаге, конструкция вида `CF?` означает неопределенность

флага, конструкция вида CF означает фиксацию признака переноса в соответствии с результатом операции, а отсутствие флага в списке означает, что операция не влияет на его значение.

з) конструкции типа eAX, eCX и т. п. представляют возможность использования как 16-разрядного, так и 32-разрядного регистра, т. е. eAX эквивалентно “AX либо EAX”.

В приводимом ниже перечне команд содержатся сведения только об основных командах, которые с большой вероятностью могут фигурировать в программах, разрабатываемых студентами по дисциплине «Машинно-ориентированное программирование».

### 3.2.2 Организация передачи управления

Средства организации передачи управления в машинных программах включают в себя команды условного и безусловного перехода, организации циклов, вызова подпрограмм и возврата из них, вызова прерываний и возврата из них, а также команды манипулирования флагами.

#### Безусловные переходы

**JMP SHORT  $v8$**  Переход, задаваемый знаковым смещением  $v8$  относительно eIP. Реализует операцию  $eIP += v8$ .

**JMP NEAR  $v16/v32$**  Переход, задаваемый знаковым смещением  $v16/v32$  относительно eIP. Реализует операцию  $eIP += v16/v32$

**JMP FAR  $v32/v48$**  Переход, задаваемый FAR-адресом, загружаемым из машинной команды в CS:eIP. В IA-16 реализует присвоение  $CS:IP = v32$ , в IA-32 –  $CS:EIP = v48$ .

**JMP WORD PTR  $d$**  Внутрисегментный переход в режиме IA-16 по адресу, равному операнду  $d$ . Операнд загружается в IP.

**JMP DWORD PTR  $d$**  В режиме IA-32 внутрисегментный переход, а в режиме IA-16 межсегментный переход  $CS:IP = m[d]$ .

**JMP FWORD PTR  $d$**  Межсегментный переход в режиме IA-32 по FAR-адресу, равному операнду  $d$ .

## Условные переходы

Команды условного перехода задаются мнемокодом вида *Jcnd v*, где *cnd* – суффикс, определяющий условие ветвления, *v* – константное выражение размером в байт, слово или двойное слово (в 16-разрядном режиме только байт). Обычно *v* представляется меткой оператора, куда нужно переходить, но в машинную команду ассемблер транслирует расстояние «прыжка» до точки перехода. Расстояние «прыжка» может быть от  $-128$  до  $+127$  при однобайтном *v*, от  $-2^{15}$  до  $+2^{15} - 1$  при двубайтном *v* и от  $-2^{31}$  до  $+2^{31} - 1$  при четырехбайтном *v*.

Суффиксы *cnd* команд условного перехода по флагам используются также в командах условной передачи данных *CMOV cnd* и условной установки байта *SETcnd*.

### *Условные переходы по флагам и значению CX*

<b>JE/JZ</b> <i>v</i>	Переход, если равно (ZF=1).
<b>JNE/JNZ</b> <i>v</i>	Переход, если не равно (ZF=0).
<b>JC</b> <i>v</i>	Переход, если перенос (CF=1).
<b>JNC</b> <i>v</i>	Переход, если нет переноса (CF=0).
<b>JS</b> <i>v</i>	Переход, если отрицательно (SF=1).
<b>JNS</b> <i>v</i>	Переход, если неотрицательно (SF=0).
<b>JO</b> <i>v</i>	Переход, если переполнение (OF=1).
<b>JNO</b> <i>v</i>	Переход, если нет переполнения (OF=0).
<b>JP/JPE</b> <i>v</i>	Переход по четности числа единиц результата (PF=1).
<b>JNP/JPO</b> <i>v</i>	Переход по нечетности числа единиц результата (PF=0).
<b>JCXZ</b> <i>v</i>	Переход, если CX=0.
<b>JECXZ</b> <i>v</i>	Переход, если ECX=0.

### *Переходы по результату операции над числами со знаком*

<b>JG/JNGE</b> <i>v</i>	Переход, если больше (SF=OF И ZF=0).
<b>JGE/JNL</b> <i>v</i>	Переход, если больше или равно (SF=OF ИЛИ ZF).
<b>JL/JNGE</b> <i>v</i>	Переход, если меньше (SF ≠ OF).
<b>JLE/JNG</b> <i>v</i>	Переход, если меньше или равно (SF ≠ OF ИЛИ ZF).



### ***Переходы по результату операции над беззнаковыми числами***

- JA/JNBE** *v*    Переход, если больше/выше (CF=0 И ZF=0).  
**JB/JNAE** *v*    Переход, если меньше/ниже (CF=1).  
**JAЕ/JNB** *v*    Переход, если не меньше/не ниже (CF=0).  
**JBE/JNA** *v*    Переход, если не больше/ не выше (CF ИЛИ ZF).

### **Организация циклов со счетчиком**

- LOOP** *v8*                      Переход, если --сХ ≠ 0  
**LOOPE/LOOPZ** *v8*          Переход, если --сХ ≠ 0 и ZF=1  
**LOOPNE/LOOPNZ** *v8*    Переход, если --сХ\$ ≠ 0 и ZF=0

### **Операции с флагами**

- CLC**                      Сброс флага переноса (CF=0).  
**STC**                      Установка флага переноса (CF=1).  
**CMC**                      Инверсия флага переноса CF.  
**LAHF**                    Загрузка младшего байта регистра флагов в регистр АН.  
**SAHF**                    Загрузка флагов SF, ZF, AF, PF, CF из бит 7, 6, 4, 2, 0 регистра АН.  
**POPF**                    Извлечение данных из стека в регистр флагов (EFLAGS[15:0]).  
**PUSHF**                   Помещение в стек младшего слова регистра флагов (EFLAGS[15:0]).  
**POPFD**                   Извлечение данных из стека в расширенный регистр флагов EFLAGS.  
**PUSHFD**                  Помещение в стек расширенного регистра флагов EFLAGS.

### **Организация процедур и прерываний**

- CALL** *s*                  Вызов процедуры. Аналогично JMP тип операнда *s* может быть NEAR, FAR, WORD PTR, DWORD PTR, FWORD PTR.  
**RET**                      Возврат из процедуры внутрисегментный.  
**RETF**                    Возврат из процедуры межсегментный.

<b>RET</b> <i>v16</i>	Возврат из процедуры внутрисегментный с освобождением в стеке блока параметров размером <i>v16</i> байт при 16-разрядной адресации, и размером <i>v16</i> слов – при 32-разрядной.
<b>RETF</b> <i>v16</i>	Возврат из процедуры межсегментный с освобождением в стеке блока параметров.
<b>ENTER</b> <i>v16, v8</i>	Выделение блока параметров размером <i>v16</i> байт в стеке для процедуры с логической вложенностью <i>v8</i> .
<b>LEAVE</b>	Освобождение блока параметров в стеке.
<b>INT</b> <i>v8</i>	Программное прерывание с номером <i>v8</i> .
<b>INT</b> 3	Однobaйтное прерывание для перехода к отладчику.
<b>INTO</b>	Выполнение программного прерывания 4, если OF=1.
<b>IRET</b>	Возврат из прерывания с восстановлением из стека IP, CS, FLAGS.
<b>IRETD</b>	Возврат из прерывания с восстановлением из стека EIP, CS, EFLAGS.
<b>CLI</b>	Запрет маскируемых аппаратных прерываний (IF=0).
<b>STI</b>	Разрешение маскируемых аппаратных прерываний (IF=1).

### 3.2.3 Пересылка данных

#### Копирование и обмен данными

<b>MOV</b> <i>d, s</i>	Копирование <i>s</i> в <i>d</i> .
<b>MOVSX</b> <i>d, s</i>	Копирование байта/слова <i>s</i> в слово/двойное слово <i>d</i> со знаковым расширением.
<b>MOVZX</b> <i>d, s</i>	Копирование байта/слова <i>s</i> в слово/двойное слово <i>d</i> с нулевым расширением.
<b>CMOVCnd</b> <i>d, s</i>	Копирование <i>s</i> в <i>d</i> , если выполняется условие <i>cnd</i> .
<b>XCHG</b> <i>d, s</i>	Взаимообмен данными между регистрами или регистром и памятью.
<b>CMPXCHG</b> <i>d, s</i>	Если AL/eAX= <i>d</i> , то <i>d</i> = <i>s</i> , ZF=1, иначе AL/eAX= <i>d</i> , ZF=0.

**CMPXCHG8B** *m64*    Условная перестановка учетверенного слова. Если  $EDX\_EAX = m64$ , то ( $m64 = ECX\_EBX$ ,  $ZF=1$ ), иначе  $EDX\_EAX = m64$ .

**BSWAP** *s*    Перестановка байт из порядка младший-старший (L-H) в порядок старший-младший (H-L).

**XLAT/XLATB**    Загрузка AL байтом с адресом  $[EBX+AL]$ .

### Стековые команды пересылки

**POP** *d*    Извлечение слова/двойного слова из стека в регистр или память *d* с последующим автоувеличением SP/ESP на 2/4.

**POPA**    Извлечение 8 слов слов из стека в регистры DI, SI, BP, SP (извлечение без модификации регистра), BX, DX, CX, AX с последующим автоувеличением SP/ESP на 16.

**POPAD**    Извлечение 8 двойных слов из стека в регистры EDI, ESI, EBP, ESP (извлечение без модификации регистра), EBX, EDX, ECX, EAX с последующим автоувеличением ESP на 32.

**PUSH** *s*    Помещение слова/двойного слова в стек после автоуменьшения SP/ESP на 2/4.

**PUSHA/PUSHAD**    Помещение в стек регистров eAX, eCX, eDX, eBX, eSP (исходное значение), eBP, eSI, eDI с автоуменьшением SP/ESP на 2/4 перед записью в стек содержимого каждого регистра.

### Загрузка указателей и ввод-вывод

**LEA** *r, m*    Загрузка эффективного адреса операнда *m* в регистр *r*.

**LDS/LES/LFS/LGS/LSS** *r, m*    Загрузка регистра  $\$r\$$  и сегментного регистра DS/ES/FS/GS/SS far-адресом, равным значению операнда  $\$m\$$ .

**IN** AL/eAX, *p*    Ввод из порта *p* ввода/вывода в AL/eAX; *p* – либо константа  $v8$ , либо DX.

**OUT** *p, AL/eAX*    Вывод в порт из AL/eAX.

### 3.2.4 Команды арифметической обработки

#### Традиционные операции двоичной арифметики

**ADD** *d, s*          Сложение  $d:=d+s$ . Все флаги.

**ADC** *d, s*          Сложение двух операндов с учетом переноса от предыдущей операции  $d:=d+s+CF$ . Все флаги.

**SUB** *d, s*          Вычитание  $d:=d-s$ . Все флаги.

**SBB** *d, s*          Вычитание с заемом  $d:=d-s-CF$ . Все флаги.

**CMP** *d, s*          Сравнение *d-s* без сохранения разности. Все флаги.

**INC** *d*            Инкремент  $d:=d+1$ . Все флаги, кроме CF.

**DEC** *d*            Декремент  $d:=d-1$ . Все флаги, кроме CF.

**NEG** *d*            Изменение знака операнда  $d:=0-d$ . ZF. CF=1.

**MUL** *s*            Умножение беззнаковое

AX:=AL\*s/ DX\_AX:=AX\*s/ EDX\_EAX:=EAX\*s. OF=CF=0, если результат помещается в AL/AX/EAX, иначе OF=CF=1; SF?,ZF?,AF?,PF?.

**IMUL** *s*          Умножение знаковое (аналогично MUL).

**DIV** *s*            Деление беззнаковое

(AL:=AX DIV *s*; AH:=AX % *s*) / (AX:=DX\_AX DIV *s*; DX:=DX\_AX % *s*) / (EAX:=EDX\_EAX DIV *s*; EDX:=EDX\_EAX % *s*); вызов INT 0, если результат не помещается в AL/AX/EAX. Все флаги неопределены.

**IDIV** *d*          Деление знаковое (аналогично DIV).

**XADD** *d, s*        Обмен содержимым и сложение  
( $\langle d, s \rangle := \langle s+d, d \rangle$ ).

**CBW/CWDE**      Преобразование байта AL в слово AX (расширение знака AL в AH – AH заполняется битом 7 AL) или слова AX в двойное слово EAX.

**CWD/CDQ**        Преобразование через расширение знака слова AX в двойное слово DX:AX / двойного слова EAX в учетверенное EDX:EAX.

#### Команды десятичной арифметики

**DAA/DAS**        Десятичная коррекция AL после сложения/вычитания двух упакованных чисел.

**AAA/AAS** Десятичная коррекция AL после сложения/вычитания двух неупакованных чисел.

**AAD** Десятичная коррекция AX перед делением неупакованного двузначного числа.

**AAM** Десятичная коррекция AX после умножения двух неупакованных чисел.

### 3.2.5 Команды логических операций

**NOT  $d$**  Инвертирование всех разрядов  $d := \text{NOT } d$ .

**AND  $d, s$**  Логическое И:  $d := d \text{ AND } s$ . F=OF=0, SF, ZF, AF?, PF.

**OR  $d, s$**  Логическое ИЛИ:  $d := d \text{ OR } s$ . CF=OF=0, SF, ZF, AF?, PF.

**XOR  $d, s$**  Исключающее ИЛИ:  $d := d \text{ XOR } s$ . CF=OF=0, SF, ZF, AF?, PF.

**TEST  $d, s$**  Проверка бит (логическое И  $d \text{ AND } s$  без записи результата). CF=OF=0, SF, ZF, AF?, PF.

### 3.2.6 Команды сдвигов

**SHL  $d, s$**  Логический сдвиг влево  $d$  на  $s \in \{1, \text{CL}, \text{v}8\}$  разрядов. Справа вдвигается 0, а старший бит выдвигается в CF. CF, SF, ZF, AF?, PF, OF при  $s=1$ , OF? при  $s>1$ ). В Си соответствует операции  $d \ll= s$  для случая, когда  $d$  беззнаковый, т. е. unsigned.

**SHR  $d, s$**  Логический сдвиг вправо. Аналогично SHL, но 0 вдвигается слева, а в CF выдвигается младший бит. В Си соответствует операции  $d \gg= s$  для случая, когда  $d$  беззнаковый, т. е. unsigned.

**SAL  $d, s$**  Арифметический сдвиг влево. Совпадает с SHL.

**SAR  $d, s$**  Арифметический сдвиг вправо. Отличается от SHR тем, что знаковый разряд  $d$  сохраняет свое значение. В Си соответствует операции  $d \gg= s$  для случая, когда  $d$  операнд со знаком, т. е. signed.

**ROL/ROR  $d, s$**  Циклический сдвиг влево/вправо. CF, OF при  $s=1$ , OF? при  $s>1$ .

**RCL/RCR**  $d, s$  Циклический сдвиг влево/вправо с включением в кольцо сдвига флага CF. CF, OF при  $s=1$ , OF? при  $s>1$ .

**SHLD**  $d, r, s$  Логический сдвиг влево операнда  $d$ , таким образом, будто  $r$  образует с ним пару  $d\_r$ , однако разряды  $r$  выдвигаются в  $d$ , но в конечном итоге  $r$  сохраняет исходное значение. OF?, SF, ZF, AF?, PF, CF.

**SHRD**  $d, r, s$  Логический сдвиг влево/вправо операнда  $d$ , таким образом, будто  $r$  образует с ним пару  $r\_d$ , однако разряды  $r$  выдвигаются в  $d$ , но в конечном итоге  $r$  сохраняет исходное значение. OF?, SF, ZF, AF?, PF, CF.

### 3.2.7 Команды обработки бит

**BSF/BSR**  $r, s$  Сканирование бит вперед (от младшего к старшему) / назад(от старшего к младшему) до встречи бита со значением 1 и запись номера этого бита в  $r$ . ZF.

**BT**  $d, s$  Тестирование бита –  $CF:=d[s]$ .

**BTR/BTS**  $d, s$  Тестирование и сброс/установка бита –  $CF:=d[s]$ ,  $d[s]=0/1$ .

**BTC**  $d, s$  Тестирование и инвертирование бита –  $CF:=d[s]$ ,  $d[s]:=NOT\ d[s]$ .

**SET** $cnd\ d8$  Если выполняется условие  $cnd$ , то  $d8:=01h$ , иначе  $d8:=0h$ .

### 3.2.8 Префиксы команд

**SIZ** Изменение установленной по умолчанию разрядности данных 16/32 на 32/16 для следующей команды.

**ADDRSIZ** Изменение умалчиваемой разрядности адреса 32/16 на 16/32 для следующей команды.

**CS:/SS:/DS:/ES:/FS:/GS:** Явное назначение сегментного регистра для адресации памяти в следующей команде.

**LOCK** Захват локальной шины на время выполнения инструкции.

**REP/REPE/REPZ/REPNE/REPNZ** Префикс повтора строковых операций (см. ниже).

### 3.2.9 Команды поддержки обработки массивов

Основу поддержки обработки массивов в I80x86 образуют так называемые строковые команды. Эти команды используют неявную адресацию через DS:eSI и ES:eDI с постмодификацией индексного регистра либо в сторону увеличения (DF=0), либо в сторону уменьшения (DF=1) на размер адресуемого данного, т. е. на 1, 2 или 4. Для far-адреса DS:eSI допускается замена сегментного регистра через префикс.

**MOVSБ/MOVSW/MOVSД** Копирование байта/слова/двойного слова из памяти по адресу DS:eSI в память по адресу ES:eDI; модификация eSI, eDI.

**LODSБ/LODSW/LODSД** Копирование байта/слова/двойного слова из памяти с адресом DS:eSI в AL/AX/EAX; модификация eSI.

**STOSБ/STOSW/STOSД** Запись байта/слова/двойного слова из AL/AX/EAX в память по адресу ES:eDI; модификация eDI.

**CMPSБ/CMPSW/CMPSД** Сравнение массивов байт/слов/двойных слов через фиксацию в регистре флагов признаков результата операции вычитания ((far \*) DS:eSI - (far \*) ES:eSI); модификация eSI, eDI. Все флаги.

**SCASБ/SCASW/SCASД** Сравнение с фиксацией в регистре флагов признаков результата операции вычитания AL/AX/EAX из байта/слова/двойного слова по адресу ES:eDI; модификация eDI. Все флаги.

**INSБ/INSW/INSД** Запись байта/слова/двойного слова, введенного из порта с адресом DX, в память по адресу ES:eDI; модификация eDI.

**OUTSB/OUTSW/OUTSD** Вывод байта/слова/двойного слова из памяти с адресом DS:eSI в порт с адресом DX; модификация eSI.

**REP** Префикс повтора строковых операций до обнуления счетчика eCX, счетчик декрементируется на каждом повторе.

**REPE/REPZ** Повторение, пока ZF=1 и CX > 0.

**REPNE/REPNZ** Повторение, пока ZF=0 и CX > 0.

## CLD/STD

Сброс/установка флага направления.

При DF=0 – автоинкремент eSI, eDI, при DF=1 – автодекремент.

## BOUND *r, m*

Проверка нахождения содержимого

регистра *r* в границах, заданных двумя подряд расположенными ячейками памяти *m*, и генерация прерывания INT 5 при нарушении границ.

### 3.3. Вопросы и упражнения

1) Какую выгоду при сложении многобайтных чисел произвольной длины можно извлечь из факта, что команда INC не модифицирует флаг CF?

2) Какое значение будет в регистре AL и флагах ZF, CF, OF после выполнения следующих двух операторов: MOV AL, 80h; SAR AL, 1?

3) Какие значения будут иметь регистр AL и флаги ZF, CF, OF после выполнения команды ADD AL, 0A3 при исходном значении в регистре AL, равном 81h?

4) Проанализировать команды условного перехода и по аналогии с макросами из примера 2.7 разработать макросы MAX32, MAX8\_4, MIN32, MIN32\_4, в которых в регистре EAX формируются соответственно максимумы и минимумы для беззнаковых 32-разрядных переменных. Отладить эти макросы на нескольких макровыводах.

5) Проанализировать команды сложения и вычитания и разработать макросы ADD64 и SUB64 для соответственно сложения и вычитания 64-разрядных переменных по схеме операторов Си  $*p1 += *p2$  и  $*p1 -= *p2$ , где в качестве  $*p1$  и  $*p2$  выступают адреса, передаваемые через параметры *p1* и *p2* макросов. В качестве временного регистра в макросах можно использовать регистр EAX. Учесть, что перенос/заем, возникающий в ходе обработки младших 32-разрядных частей операндов, должен быть задействован при обработке старших частей.

6) Проанализировать команды сдвига и разработать макросы RCR64, RCL64, SHL64, SHR64, ASL64 и ASR64, которые являются 64-разрядными аналогами соответствующих команд сдвига. Разместить



макровывозовы в сегменте кода и отладить макросы в среде отладчика OllyDbg.

7) Проанализировать команду NEG и разработать макрос NEG64, который выполняет операцию  $x = -x$  для 64-разрядного операнда.

8) Проанализировать возможности команды MOVSD и префикса повторения REP, разработать на их основе копирование одной записи со структурой Stat (пример 2.9) в другую без использования обращения к отдельным полям записей. Проанализировать возможности команд сдвига и организации циклов LOOPZ/LOOPNZ. Разработать макрос формирования в регистре EDX числа единиц в регистре EAX, а в регистре ECX номера разряда EAX, котором находится самая правая единица. При этом использовать команду сдвига вправо и использование либо выдвигаемого бита, либо содержимого младшего бита EAX. Выход из цикла должен быть организован после обработки самой правой единицы числа в EAX.

9) Проанализировать команды обработки бит и реализовать задачу предыдущего упражнения на основе этих команд, предполагая, что операнд в EAX всегда содержит одну группу подряд следующих единиц, которая слева и справа может быть окружена нулями.

10) Проанализировать команды CMOV и CMPXCHG. Реализовать макрос IMIN32\_4 из примера 2.7 с использованием этих команд.

11) Запрограммировать последовательность команд строчной обработки, в которой фигурируют команды обработки байтов, слов, двойных слов. Проанализировать в листинге результат трансляции этих команд с целью поиска ответа на вопрос: «Как в машинной программе представляется длина операнда?». Подвергнуть трассировке в отладчике последовательности команд строчной обработки, отслеживая изменения содержимого регистров ESI и EDI.

## **4. Реализация управления обработкой данных на ассемблере**

### **4.1. Базовые понятия и соглашения**

Управление обработкой данных заключается прежде всего в организации последовательности исполнения команд. Процессор с архитектурой x86 читает машинные команды из памяти, адресуясь указателем команд EIP/IP. На линейном участке машинной программы по мере обработки очередной команды указатель продвигается на следующие элементы данных, образуя так называемый продвинутый адрес. Продвинутый адрес сначала указывает на продолжение текущей команды, а по завершению обработки всех ее полей становится адресом следующей команды. Именно так в архитектуре x86 реализуется естественная адресация команд.

Естественная адресация в течение многих лет была реализована в таких компьютерных архитектурах, где каждая машинная команда занимала одно машинное слово. Это обстоятельство закрепило за регистром адреса команды название «счетчик команд». На линейном участке для вычисления адреса следующей команды в таких архитектурах к содержимому счетчика команд достаточно прибавить единицу.

В архитектуре x86 последовательность адресов памяти  $A, A+1, A+2, \dots$

адресует байты, расположенные в памяти подряд. Различные машинные команды могут иметь различную длину. Более того, адрес, смещение или непосредственный операнд, являющиеся частями машинных команд, могут занимать от одного до 8 байтов. Это означает, что процессор после включения в обработку очередного элемента данных команды продвигает EIP/IP путем прибавления к нему длины размера этого элемента.

Команды перехода, обеспечивающие управление последовательностью исполнения команд, способны изменить продвинутый адрес таким образом, чтобы программа продолжилась с иной точки, нежели следующая команда. Такое изменение достигается либо занесением в

указатель команд адреса целевой точки перехода, либо подсуммированием смещения к продвинутому адресу, что обеспечивает «прыжок» вперед или назад от текущей точки программы на расстояние, заданное смещением.

Лучший способ осмыслить процессы манипулирования счетчиком команд в исходных точках перехода – это покомандная трассировка программы в отладчике. Если текущая машинная команда в ходе трассировки является командой условного перехода, то полезно сначала проанализировать текущее значение указателя команд, значение поля смещения в команде и значение задействованных в ней признаков результата (флагов). Затем нужно выполнить переход с помощью команды Step трассировщика (F7 для OllyDbg) и проанализировать новое значение указателя команд.

Управление переходами связано с результатами обработки данных, прежде всего признаками вида «операнд равен/не равен нулю», «операнды равны/не равны», «первый операнд больше/не больше второго» и т. п. Чтобы избежать повторения объявления данных, рассматриваемые ниже примеры предполагают, что в сегменте данных имеются объявления, метки которых формируются из композиции «Буква-Разрядность».

Например:

```
.data
; переменные типа char / unsigned char / byte
a8      db      ?
b8      db      ?
n8      db      ?
; переменные типа short / unsigned short / word
a16     dw      ?
b16     dw      ?
n16     dw      ?
; переменные типа long / unsigned long / dword
a32     dd      ?
b32     dd      ?
n32     dd      ?
```

Для большинства примеров в качестве комментария записывается текст фрагмента на языке Си, который представляется ниже соответствующим фрагментом на языке ассемблера. При этом предполагается, что переменным ассемблер-программы a8, a16, a32, b8 и т. п. соответствуют такие же имена переменных Си-программы.

В ассемблере MASM имеются встроенные макросы организации ветвлений и циклов, однако изучать механизмы управления обработкой данных в ассемблере лучше без их применения. Это связано с тем, что по статистике большинство ошибок программирования на ассемблере концентрируется как раз вокруг команд организации ветвлений. Обнаружение этих ошибок требует трассировки программы, отслеживая порядок исполнения команд. Макросы скрывают команды, что создает некоторый семантический разрыв между исходным текстом и трассируемой машинной программой.

## **4.2. Программирование ветвлений**

### **4.2.1 Ветвление по условиям**

Ветвления по условиям в ассемблер-программе реализуются через команды условного перехода. Каждая такая команда анализирует значение признаков результата ранее выполненных операций, фиксируемых в регистре флагов RF, т. е. флагов ZF, CF и т. п. Если значения используемых командой перехода флагов соответствуют условию перехода, то процессор прибавляет к регистру адреса команд EIP/IP смещение, размещенное в команде перехода после кода операции. Если условие перехода не соблюдается, то продвинутый адрес в EIP/IP после исполнения команды условного перехода будет указывать на следующую в памяти команду. Например, переход JC по признаку переноса CF будет вызывать подсуммирование смещения к указателю команд EIP/IP в случае, когда  $CF = 1$ , а в случае  $CF=0$  подсуммирование не выполнится.

Описанный процесс выполнения команд ветвления обуславливает реализацию ветвлений в программах как минимум в два шага:

- а) выполнение одной или более команд обработки с целью формирования признаков результата в одном или нескольких флагах;
- б) выполнение команды условного перехода.

При использовании продуктов формирования признаков результата в разрядах регистра флагов необходимо учитывать два описываемых ниже обстоятельства.

Во-первых, любая команда в фазе фиксации значения конкретного флага в регистре RF может вести себя следующим образом:

- а) не менять;
- б) заносить константу, например, сбрасывать в 0;
- в) заносить произвольное значение;
- г) устанавливать в 0 или 1 в зависимости от результата операции.

При планировании ветвления по результату операции указанное обстоятельство заставляет контролировать, влияет ли применяемая операция на используемый флаг. Например, в языке Си мы можем написать оператор ветвления «`if((a32 = b32) == 0) x32 = 5;`», поскольку присвоение в языке Си есть операция, а не оператор, и у этой операции имеются два признака: нуль/не нуль, положительно/отрицательно. Однако команда MOV, с помощью которой в x86 обычно реализуется операция (a32=b32), не влияет на признак результата ZF. Это значит, что перед командой условного перехода «по нулю» JZ или «не нулю» JNZ придется после команды пересылки MOV выполнять какую-то команду обработки, не меняющую переменные a32 и b32, но влияющие на флаг ZF. Например, команду OR, как это показано ниже.

```
mov    eax, a32;
mov    b32,  eax
or     eax, eax ; формирование признака ZF
jnz    Lelse
mov    dword ptr x32, 5
Lelse:
```

При трансляции приведенного выше примера ассемблер в первом проходе определяет длину команды `jnz else`, еще не зная расстояния прыжка до метки `else`. Это обстоятельство вынуждает ассемблер резервировать несколько байтов для поля смещения. Если мы явно зададим короткое смещение через `jnz short Lelse`, то транслятор сгенерирует двухбайтную команду `jnz`.

Различные команды влияют на флаги по-разному. Например, команда `add op1, op2` подсуммирует операнд `op2` к операнду `op1` и формирует признак переноса, как и положено команде сложения, – `CF` это перенос из старшего разряда суммы операндов. Команда `inc op` выполняет подсуммирование единицы к операнду `op`, однако признак `CF` эта команда не меняет. Указанное обстоятельство требует быть внимательным по отношению к правилам формирования флагов. Наиболее опасны проявления такой невнимательности при оптимизации кода. Во время оптимизации мы сосредоточены на задаче уменьшения размера кода или времени его выполнения, и в этих условиях легко упустить факт, что изменение непосредственно перед командой условного перехода нарушает требуемую связь между признаками результата и условиями перехода.

Во-вторых, ветвления по неравенствам зависят от наличия знака у сравниваемых величин. Пусть в байте `a8` находится число, содержащее 8 единиц `11111111`. Для беззнаковой величины типа `unsigned char` это есть самое большое число 255, и по отношению ко всем другим числам величина `a8` будет больше. Если же `a8` представляет собой число со знаком, то все единицы в дополнительном коде независимо от разрядности операнда всегда представляют число “–1”: для байта, слова или двойного слова. В этом легко убедиться, если вычесть 1 из двоичного нуля. Диапазон 8-разрядных знаковых чисел равен  $-128 \div +127$ , а это значит, что существуют 128 чисел больше –1.

Для сравнения чисел обычно используется команда `cmp op1, op2`, которая формирует значения признаков результата путем вычитания

op1 – op2, не меняя операндов. Арифметическое устройство процессора в дополнительном коде обрабатывает знаковый разряд точно так же, как и все остальные. В ходе вычитания нет ничего такого, что отличало бы процесс для знаковых величин от процесса для беззнаковых. Различие должно учитываться в следующих случаях:

а) ввод-вывод; для чтения человеком беззнаковый байт со всеми единицами должен выводиться как подстрока “255”, а при наличии знака в таком байте, как подстрока “–1”;

б) выравнивание длин операндов; при подсуммировании беззнакового байта `unsigned char 11111111b` к слову должен быть расширен до слова нулями в старшем байте, чтобы полученное слагаемое равнялось 255, а такой же байт, но типа `signed char`, должен быть дополнен единицами, чтобы в случае всех единиц в полученном слове сохранилось значение –1;

в) организация переходов; например, ветвление `if (a8<=b8)` для знаковых переменных `a8` и `b8` должно быть реализовано иными командами ветвления, нежели для беззнаковых; при беззнаковом операнде `b8=11111111b` переход по условию `(a8<=b8)` будет иметь место для любых беззнаковых значений `a8`, а в случае переменных со знаком переход по этому условию при том же самом значении `b8` будет истинным только для половины возможных значений `a8`.

### **Ветвление по условию if-then**

На машинном уровне эта конструкция обычно реализуется следующим образом: сначала вычисляются условия, затем выполняется переход по противоположному набору признаков результата так, чтобы перепрыгнуть через фрагмент, который должен выполняться при истинности условий.

#### **Пример 4.1. Реализация if-then**

Реализация фрагмента Си-программы `if (a8 != 0) { b8 = 3; c8 = 7; }` указанным выше образом, имеет вид:

```
cmp    byte ptr a8, 0
je     else ; перепрыгиваем через then по a8==0
```

```

; блок then { b8 = 3; c8 = 7;}
mov    byte ptr b8, 3
mov    byte ptr c8, 7
after: ; продолжение программы

```

В режиме IA-16 расстояние прыжка определяется всего одним байтом смещения. Это байт со знаком, поэтому прыжок может быть не далее 127 байтов вперед и 128 байтов назад от точки программы, которая находится сразу за командой ветвления. Это определяется тем обстоятельством, что указатель инструкций IP во время выборки команды ветвления из памяти продвигается вперед и в момент прибавления смещения к IP последний указывает уже на следующую машинную команду. Когда транслятор транслирует блок then, он не знает его размера, и до фазы оптимизации транслятор вынужден будет использовать пару команд перехода – условный переход на блок then и безусловный переход по метке after, как это показано в примере 4.2.

#### **Пример 4.2. Реализация if-then двумя командами перехода**

Реализация if (a8 != 0) { b8 = 3; c8 = 7;} двумя командами перехода имеет вид:

```

cmp     byte ptr a8, 0
jne     then ; перепрыгиваем через then по a==0
jmp     after
        ; блок then { b8 = 3; c8 = 7;}
then:   mov    byte ptr b8, 3
        mov    byte ptr c8, 7
after:  ; продолжение программы

```

Оптимизатор компилятора вычисляет размеры блока и выполняет замену пары jne-jmp на один оператор je, как запрограммировано в примере 4.1.

#### **Ветвление по условию if-then-else**

При реализации этой управляющей конструкции необходимо выполнить один из двух блоков. Если переход организуется непосредственно по полученному значению условного выражения, то



вслед за фрагментом вычисления условного выражения идет блок `else`. Если же используется инверсное условие перехода, то идет блок `then`.

#### **Пример 4.3. Реализация if-then-else**

Требуется реализовать следующий фрагмент Си-программы, в котором `a8` и `b8` беззнаковые:

```
if ( a8 > b8 ) { a8 = 3; b8 = 7; }  
else { a8 = 1; b8 = 2;}
```

На ассемблере это будет выглядеть так:

```
; используется регистр al, т.к. у cmp не может быть оба операнда в памяти  
mov     al, a8  
; если не(a > b), идти к блоку else  
cmp     al, b8 ; RF <= флаги разности (a8-b8)  
jna     else  
        ; блок then { a8 = 3; b8 = 7; }  
        mov byte ptr a8, 3  
        mov byte ptr b8, 7  
        jmp short after ; перепрыгиваем через else  
        ; блок else { a8 = 1; b8 = 2; }  
else:   mov byte ptr a8, 1  
        mov byte ptr b8, 2  
after:
```

#### **Ветвление по условиям if-elseif-elseif...-else**

На ассемблере цепь `if-elseif-elseif...-else` строится как ряд действий: `<проверка_условия, переход_по_невыполнению, блок_операторов_j>`, где `j` – номер ветви.

#### **Пример 4.4. Реализация последовательности ветвлений**

Пусть требуется реализовать следующий фрагмент Си-программы, в котором все переменные являются байтами со знаком:

```
if ( a8 = 1 ) { b8 = 5; }  
else if ( a8 = 2 ) b8 = 3;  
else if ( a8 = 7 ) b8 = - 3;  
else b8 = 0;
```

На ассемблере это будет выглядеть так:

```

cmp byte ptr a8, 1
jne cond2 ; прыгнуть к проверке условия 2
    ; блок then по условию 1
    mov byte ptr b8, 5
    jmp shorpafter
    ; блок then по условию 2
cond2: cmp byte ptr a8, 2
    jne cond3 ; прыгнуть на проверку условия 3
    mov byte ptr b8, 3
    jmp short after
    ; блок then по условию 3
cond3: cmp byte ptr a8, 7
    jne else
    mov byte ptr b8, -3
    jmp short after
    ; блок else
else: mov byte ptr b8, 0
after:

```

## 4.2.2 Реализация оператора switch

### Реализация на основе ветвлений

Оператор выбора задает переход к одной из нескольких точек в зависимости от значения переменной. Наиболее очевидный способ реализации оператора выбора базируется на только что рассмотренной конструкции if-elseif-...elseif-else.

### Пример 4.5. Реализация switch с помощью ветвлений

Пусть требуется реализовать следующий фрагмент Си-программы:

```

switch ( n32 )
{
    case 2 : b8 = 5; break;
    case 3 : a8 = 6; break;
    case 4 : c8 = 15; break;
    case 5 : b8 = 7; break;
    case 8 : b8 = 4; break;
    default: b8 = 20;
}

```

Реализация этого фрагмента на ассемблере аналогично реализации конструкции if-elseif-...-else даст текст, приведенный ниже.

```
        mov     eax, n32
        cmp     eax, 2
        jne     case3
        mov     byte ptr b8, 5
        jmp     short after
case3:   cmp     eax, 1
        jne     case4
        mov     byte ptr a8, 6
        jmp     short after
case4:   cmp     eax, 2
        jne     case5
        mov     byte ptr c8, 15
        jmp     short after
case5:   cmp     eax, 3
        jne     case8
        mov     byte ptr b8, 7
        jmp     short after
case8:   cmp     eax, 6
        jne     dflt
        mov     byte ptr b8, 4
        jmp     short after
dflt:   mov     byte ptr b8, 20
after:
```

Недостатком рассмотренного решения являются слишком большие затраты времени на выполнение пар команд `cmp-jne` для тех значений выражения в операторе `switch`, которые соответствуют ветвям `case`, далеко отстоящим от начала проверки.

### **Реализация switch на основе таблицы переходов**

Указанный выше недостаток реализации устраняется, если организовать переход по таблице меток на основе косвенного перехода. Метод заключается в том, что для значений аргумента оператора `switch(n)`, где `n` лежит в диапазоне `[n_min..n_max]` создается таблица меток, которая содержит  $(n\_max - n\_min + 1)$  элементов. Величина  $(n - n\_min)$  используется в качестве индекса массива. Естественно, что эту

величину можно использовать только для тех значений  $n$ , которые принадлежат диапазону  $n\_min..n\_max$ , иначе индекс  $(n - n\_min)$  будет указывать на элемент за пределами массива меток. Чтобы этого не произошло, необходимо сначала отфильтровать значения  $n$ , не входящие в диапазон значений ветвей `case`.

#### **Пример 4.6. Реализация `switch` с помощью таблицы меток перехода**

Пусть требуется реализовать фрагмент Си-программы, фигурирующий в примере 4.5, при условии, что `n32` является величиной со знаком.

```
.data
; Таблица меток. Для значений n32=6 и n32=7 ветви не определены,
; поэтому перед case8 идут метки dflt
tblsw dd case2, case3, case4, case5, dflt
      dd dflt, case8
.code
mov    eax, n32
; сначала проверяем вхождение eax в диапазон 2..8
cmp    eax, 2
jl     dflt ; ЕСЛИ n32 до диапазона, идти к dflt
cmp    eax, 8
jg     dflt ; ЕСЛИ n32 после диапазона, идти к dflt
; Прижимаем n32 к нулю
sub    eax, 2 ; n32 -= n_min
; Здесь переход по таблице меток
; Индекс eax умножаем на 4, поскольку элементы таблицы
; имеют длину 4 байта, а адресация должна получать
; смещение к стартовому адресу таблицы tblsw
jmp    dword ptr tblsw[4*eax]
dflt:  mov byte ptr b8, 20
      jmp short after
case2: mov byte ptr b8, 5
      jmp short after
case3: mov byte ptr a8, 6
      jmp short after
case4: mov byte ptr c8, 15
      jmp short after
case5: mov byte ptr b8, 7
```

```

        jmp short after
case8: mov byte ptr b8, 4
after:

```

Как видно из текста программы, каждая ветвь `case` уже не содержит пары операторов `cmp-jne`. Время входа в любую ветвь `case` здесь не зависит от местоположения этой ветви в программе, как это было в примере 4.5. Недостатком рассмотренного метода является слишком большое число меток `default` в таблице, если `n_max - n_min` много больше числа ветвей `case`.

### Реализация `switch` на основе таблиц значений и меток переходов

Пусть в операторе `switch` примера 4.5 вместо ветви «`case 8:`» имеется ветвь «`case 100:`». В этом случае в таблице `tblsw` будет 94 метки `dflt`. Если это по каким-то соображениям нас не устраивает, то составляются две таблицы: таблица значений выражения ветвях `case` и таблица адресов перехода. Теперь переход будет организован через две фазы: поиск в таблице значений и, если в *i*-й позиции таблицы найдено значение выражения оператора `switch`, то в качестве адреса перехода нужно взять *i*-й элемент таблицы адресов перехода.

#### Пример 4.7. Реализация `switch` с помощью двух таблиц

Пусть требуется реализовать фрагмент Си-программы, фигурирующий в примере 4.6, при условии, что вместо ветви “`case 8`” фигурирует ветвь “`case 100`”.

```

.data
; Таблица значений
tblval dd 2, 3, 4, 5, 100
; Таблица меток. Теперь она без dflt
tblsw dd case2, case3, case4, case5, case100

.code
mov eax, n32
; сначала ищем значение в первой таблице
; пусть поиск использует ecx как индекс первой таблицы
mov ecx, 0

```

```

; Цикл поиска do if(tblval[ecx] == eax) goto sw; while (++ecx < 5);
do:      cmp    eax, tblval[4*ecx]
        je     sw2      ; найдено, переход на вторую фазу switch
        inc    ecx
        cmp    ecx, 5
        jb     do
; Здесь точка, когда не найдено
jmp     dflt
sw2:      ; вторая фаза switch – косвенный переход
jmp     dword ptr tblsw[4*ecx]
dflt:    mov    byte ptr b8, 20
        jmp    short after
case2:    mov    byte ptr b8, 5
        jmp    short after
case3:    mov    byte ptr a8, 6
        jmp    short after
case4:    mov    byte ptr c8, 15
        jmp    short after
case5:    mov    byte ptr b8, 7
        jmp    short after
case100:  mov     byte ptr b8, 4
after:

```

Развитые системы программирования, в том числе Си, умеют строить переход по многим направлениям всеми рассмотренными методами, причем, если число ветвей case невелико, то компьютер построит последовательность команд `cmp-jne`. Если в диапазоне от `n_min` до `n_max` в ветвях case пропусков мало, то компьютер построит одну таблицу адресов перехода, если диапазон `n_min .. n_max` велик и в нем много не использованных значений в ветвях case, то компилятор построит две таблицы. Выбор варианта реализации case зависит также от значений ключей оптимизации в командной строке вызова транслятора.

## 4.2.3 Программирование циклов

### Цикл while-do

Цикл с предусловием характеризуется тем, что его тело может не выполниться ни одного раза. Для реализации этого цикла организуется ветвление в начале цикла.

#### Пример 4.8. Подсчет числа единичных разрядов в цикле while-do

В качестве примера рассмотрим подсчет числа единичных разрядов в переменной `a32` согласно приводимому ниже фрагменту Си-программы.

```
unsigned long a32, t32, n32;
n32 = 0; t32 = a32;
while ( t32 != 0 ) // ПОКА не все биты нулевые
{ // сравнивать и выйти из цикла, если символ найден
    if((t32 & 1) != 0) n32++; // учет единицы в младшем разряде
    t32 >>= 1; // сдвиг вправо
}
```

В качестве переменной `t32` выгодно использовать регистр, поскольку это переменная является временной, используемой, чтобы не испортить исходную переменную `a32`. Целесообразно выделить регистр для подсчета числа единиц.

```
.data
n32      dd  ?
a32      dd  ?
. code
xor      ecx, ecx      ; Очистка счетчика единиц
mov      eax, a32      ; Двигать будем eax
wh:      ; точка входа в тело цикла
          ; когда в теле есть continue, то это переход в данную точку
          ; NULL в eax будем определять по (eax or eax) == 0
or       eax, eax ; это короче, чем cmp eax, 0
jz       after
test     eax, 1 ; ZF(eax&1)
jz       sh           ; при нуле перепрыгиваем через n++
inc      ecx
sh:      shr     eax, 1 ; сдвиг
jmp      wh
br:      ; когда в теле цикла есть break, то это переход в данную точку
mov      n32, ecx ; сохранение результата
```

## Цикл do-while

Если известно, что тело цикла хотя бы один раз выполнится, то может быть целесообразным цикл с постусловием. При реализации такого цикла на ассемблере ветвление организуется в конце тела цикла.

### Пример 4.9. Подсчет числа единичных разрядов в цикле do-while

Рассмотрим вариант программы подсчета числа единичных разрядов в переменной `a32`, в котором используется постусловие. На языке Си соответствующий фрагмент выглядит так:

```
unsigned long a32, t32, n32;
n32 = 0; t32 = a32;
do {
    if((t32 & 1) != 0) n32++; // учет единицы в младшем разряде
    t32 >>= 1; // сдвиг вправо
} while (t32 != 0) // ПОКА не все нули
```

При сохранении соглашений об использовании регистров программа на ассемблере будет такой:

```
. code
xor     ecx, ecx      ; Очистка счетчика единиц
mov     eax, a32       ; Берем операнд в eax, чтобы двигать в нем
do:     ; точка входа в тело цикла
        ; когда в теле есть continue, то это переход в данную точку
        ; NULL в eax будем определять по (eax or eax) == 0
        test eax, 1 ; ZF(eax&1)
        jz   sh       ; при нуле перепрыгиваем через n++
        inc  ecx
sh:     shr   eax, 1 ; сдвиг
wh:     or    eax, eax ; проверка на 0
        jnz  do       ; возврат в тело цикла
br:     ; когда в теле цикла есть break, то это переход в данную точку
mov     n32, ecx ; сохранение результата
```

В цикле `do` на одну команду перехода меньше, нежели в цикле `while`. Уменьшение числа команд перехода важно по двум обстоятельствам: во-первых, вероятность ошибок программирования уменьшается с уменьшением числа команд ветвления, во-вторых, переходы существенно снижают эффект повышения быстродействия, достигаемый за счет



конвейеризации процесса исполнения потока команд. Для рассматриваемого примера команду ветвления по результатам анализа младшего бита можно избежать, если использовать факт, что сдвиг «shr eax, 1» переносит младший бит регистра eax в разряд CF регистра флагов. В то же время система команд x86 содержит команду «adc op1, op2», которая прибавляет к операнду op1 не только op2, но и флаг CF. Значит, мы можем убрать ветвление по младшему разряду eax и взамен команды inc ecx использовать команду adc. Фрагмент кода, использующего данную идею, приведен в примере 4.10.

#### **Пример 4.10. Подсчет единиц в слове без ветвления**

```
. code
xor     ecx, ecx      ; Очистка счетчика единиц
mov     eax, a32      ; Двигать будем eax
do:     shr     eax, 1
        adc     ecx, 0
        or      eax, eax; while(eax!=0)
        jnz     do      ; возврат в тело цикла
; Точка за телом цикла – сохранение результата
mov     n32, ecx
```

#### **Цикл for**

Во многих языках цикл for это цикл, где переменная пробегает ряд значений с определенным шагом, а в языке Си – это, по сути дела, модификация цикла while, в которой начальные установки и изменение значений параметров концентрируются в заголовке цикла.

#### **Пример 4.11. Подсчет числа единиц с помощью цикла for**

Рассмотрим вариант программы подсчета числа единичных разрядов в переменной a32, в котором зоной подсчета являются только младшие 20 разрядов. При сохранении соглашений об использовании регистров и использовании команды adc ассемблер программа будет такой:

```
. code
xor     ecx, ecx      ; Очистка счетчика единиц
mov     edx, ecx      ; edx будет в качестве k
mov     eax, a32      ; Двигать будем eax
```

```

for:      ; точка входа в тело цикла
        cmp     edx, 20; проверка k<20 ?
        jae     after ; выход из цикла
        shr     eax, 1
        adc     ecx, 0
        inc     edx      ; k++
        jmp     short for
after:
mov      n32, ecx ; сохранение результата

```

### **Циклы на основе команд loop, loopz/loope, loopnz/loopne**

В системах команд многих ЭВМ существуют специальные цикловые операторы, которые ориентированы на организацию циклов с убывающим счетчиком. В архитектуре x86 это команды: loop, loopz/loope, loopnz/loopne. В качестве убывающего счетчика эти команды используют регистр ECX/CX.

Команда loop в режиме IA-32 использует регистр ECX и работает по алгоритму `if(--ECX != 0) EIP += offs`, где offs – однобайтное смещение в команде loop, добавление которого к регистру адреса команды EIP возвращает управление в начало цикла.

#### **Пример 4.12. Использование команды LOOP для подсчета числа единиц**

Рассмотрим использование команды loop для подсчета числа единиц в младших 20 разрядах переменной a32. Поскольку команда loop использует ECX в качестве параметра цикла, будем использовать в качестве счетчика единиц другой регистр – EDX.

```

. code
xor      edx, edx      ; Очистка счетчика единиц
mov      ecx, 20        ; ecx будет в качестве k
mov      eax, a32       ; Двигать будем eax
loo:     shr     eax, 1
        adc     edx, 0
        loop   loo      ; if(--ecx != 0) goto do
mov      n, edx         ; сохранение результата

```

Машинная команда `loopz` в качестве условия продолжения цикла использует факт не достижения убывающим счетчиком `ECX` значения 0 и значения флага `ZF` = 1. Это означает, что цикл будет выполняться не более `ECX` раз, но пока имеет место признак `ZF` = 1. Альтернативным мнемокодом этой команды является `loope`.

Машинная команда `loopnz/loopne` наряду с базовым условием (`--ECX != 0`) для продолжения цикла имеет дополнительное условие `ZF = 0`.

Важно, чтобы перед командой `loopz` или `loopnz` выполнялась такая машинная команда, признак результата `ZF` которой свидетельствует о нуле или не нуле в соответствующей переменной.

#### **Пример 4.13. Использование команды LOOPNZ**

Рассмотрим такой процесс подсчета числа единиц в младших 20 разрядах переменной `a32`, при котором процесс заканчивается как при обработке 20 разрядов, так и при обнаружении 0 в обрабатываемой переменной.

```
. code
xor     edx, edx      ; Очистка счетчика единиц
mov     ecx, 20       ; ecx будет в качестве k
mov     eax, a32      ; Двигать будем eax
loo:    shr     eax, 1
        adc     edx, 0
        or      eax, eax ; формируем ZF = (eax == 0)
        loopnz loo    ; if((--ecx != 0) && (ZF == 0)) goto do
mov     n, edx        ; сохранение результата
```

#### **Циклы на основе строковых команд**

Строковые команды `x86` используются для обработки массивов на основе автоинкрементной или автодекрементной адресации. Автоинкрементная соответствует конструкции языка Си «`*p++`» и при ее использовании в одной машинной команде происходит обращение к операнду через адрес `p` и затем увеличение самого указателя на размер операнда в байтах. Автодекрементная адресация в `x86` реализуется в соответствии с конструкцией языка «`*p--`». В архитектуре `IA-32`

допускаются три размера обрабатываемого элемента строковых команд: 1 – byte, 2 – word и 4 – dword. В IA-16 используются только byte и word.

В качестве указателя *r* в строковых командах используются либо регистры ESI/SI, либо EDI/DI. Причем это не задается адресуемым операндом, а определено алгоритмами выполнения команды. Соответствующие соглашения представлены в пункте 3.2.9. Нужно обратить внимание на тот факт, что ESI/SI используется в паре с сегментным регистром DS, а EDI/DI – в паре с сегментным регистром ES.

Выбор между автоинкрементной и автодекрементной адресациями задается значением разряда DF регистра флагов. Чтобы адресация была автоинкрементной, в программе нужно выполнять команду CLD, которая обнуляет флаг DF. Задание автодекрементной требует установки DF в единицу с помощью команды STD.

Существенно упрощают циклическую обработку массивов префиксы повторения REP, REPZ/REPE, REPNZ/REPNE. Каждый из префиксов в машинном коде представляет собой один байт, предшествующий машинной команде. Строчная команда также имеет длину 1 байт и выполняется в контексте этого префикса.

Префикс REP задает выполнение строковой команды ECX/CX раз. Без этого префикса строковую команду *cmd* можно выполнить ECX/CX раз с помощью цикла:

```
loop: cmd ; некоторая строковая команда
      loop loop
```

При выполнении этого цикла процессор для строковой команды каждый раз выполняет командный цикл «выборка, дешифрация кода операции, формирование адресов, выборка операндов, выполнение, сохранение результата и его признаков, модификация одного или двух адресных регистров». Аналогично выполняются все фазы для команды loop. При наличии оператора *rep cmd* процессор фазы командного

цикла «выборка, дешифрация кода операции» выполняет всего один раз как для префикса, так и для самой команды. Вся остальная работа направлена исключительно на обработку операндов.

Префикс REPZ/REPЕ работает так же, как если бы без префикса цикл был организован на основе команды loopz/loope. Префикс REPNZ/REPNE аналогичен использованию loopnz/loope в обычной реализации обработки массива без использования префикса повторения. Рассмотрим примеры широко распространенных видов обработки массивов на основе строковых команд.

#### **Пример 4.14. Копирование массива байтов**

```
.data
src      db      40 dup(?)
dst      db      40 dup(?)
; копирование 40 байт из массива src в массив dst
. code
lea      esi, src ; инициализация указателя источника данных
lea      edi, dst ; инициализация указателя приемника данных
cld      ; Очистка DF для автоинкремента
mov      ecx, 40  ; копируется 40 байт
rep      movsb   ; собственно копирование
```

Если копирование организуется в 16-разрядном режиме, то необходимо учитывать факт, что команда movs копирует данные из сегмента памяти, адресуемого регистром DS, в сегмент, адресуемый регистром ES. В случае, если массивы src и dst находятся в одном сегменте данных, необходимо занести содержимое DS в ES. Фрагмент кода, который копирует данные в 16-разрядном режиме, имеет вид:

```
lea      si,  src ; инициализация указателя источника данных
lea      di,  dst ; инициализация указателя приемника данных
mov      ax,  ds   ; копирование в одном сегменте
mov      es,  ax   ; поэтому es = ds
cld      ; Очистка DF для автоинкремента
mov      cx,  40   ; копируется 40 байт
rep      movsb   ; собственно копирование
```

#### **Пример 4.15. Копирование элементов массива, номера которых четны**

Пусть в отличие от примера 4.14 нужно скопировать только элементы `src[0]`, `src[2]`, ..., `src[38]` в элементы `dst[0]`, `dst[1]`, ... , `dst[19]`. В этом случае не удастся применить `rep movsb`. Придется использовать `loop` в сочетании с автоинкрементной адресацией строчных команд:

```
. code
lea     esi, src ; инициализация указателя источника данных
lea     edi, dst ; инициализация указателя приемника данных
cld     ; Очистка DF для автоинкремента
mov     ecx, 20 ; копируется 20 байт
loo:    movsb    ; копирование байта *edi++ = *esi++
        lodsb    ; al = *esi++ // пропуск с нечетным номером
loop    loo
```

#### **Пример 4.16. Копирование отрицательных элементов массива**

Пусть в отличие от примера 4.14 нужно скопировать только такие элементы массива `src`, которые меньше нуля – `signed char src[40]`.

```
. code
lea     esi, src ; инициализация указателя источника данных
lea     edi, dst ; инициализация указателя приемника данных
cld     ; Очистка DF для автоинкремента
mov     ecx, 40  ; просматриваются все 40 байт src
loo:    lodsb    ; al = *esi++ // взять src[i]
        or      al, al
        jns     nocy ; al >= 0: перепрыгиваем через копирование
        stosb    ; *edi++ = al
nocy:   loop    loo
```

### **4.3. Организация подпрограмм**

#### **4.3.1 Базовые механизмы и операторы**

Подпрограмма – это фрагмент текста программы, использование которого в процессе ее выполнения может быть неоднократным. В отличие от тела цикла, которое также используется многократно, одна и та же подпрограмма вызывается из разных мест. Возврат из

подпрограммы должен происходить всегда в то место программы, где подпрограмма была вызвана, а точнее, – к той команде, которая располагается за командой вызова подпрограммы.

Это обеспечивается с помощью специального механизма исполнения процессором команд вызова подпрограмм. Механизм заключается в том, что в ходе обработки команды процессор сначала формирует адрес точки перехода (точка входа в подпрограмму), а по завершению обработки команд продвинутый адрес в EIP/IP запоминается в стеке в качестве адреса возврата из подпрограммы. Выход из подпрограммы выполняется с помощью специальной команды возврата, которая извлекает из стека адрес возврата и заносит в EIP/IP. Естественно, что после этого программа продолжается от точки за командой вызова подпрограммы.

Вызов подпрограммы осуществляет оператором `call`. При программировании в 32-разрядном режиме и flat-модели памяти следует различать два вида оператора:

а) `call метка` – прямой вызов; при выполнении соответствующей машинной команды в процессоре будет выполнено три действия:

- извлечение 32-разрядного смещения из 2..5 байтов самой команды с продвижением указателя команд в точку за команду `call`;
- сохранение адреса возврата в стеке: `mem[--ESP] = EIP`;
- подсуммирование `EIP += offset`; смещение `offset` играет роль расстояния прыжка от точки возврата к точке входа в подпрограмму – прыжок может быть вперед (в сторону увеличения EIP) и назад, поскольку смещение является 32-разрядной величиной в дополнительном коде;

б) `call dword ptr АдресноеВыражение` – косвенный вызов; при выполнении соответствующей машинной команды выполняются три действия:

- формирование эффективного адреса ЕА, опираясь на содержание адресных полей, сгенерированных ассемблером согласно адресному выражению в машинной команде; ЕА задает адрес

ячейки памяти, в которой находится адрес точки входа в подпрограмму; в ходе формирования ЕА указатель команд EIP продвигается за команду call;

- сохранение адреса возврата в стеке:  $\text{mem}[\text{--ESP}] = \text{EIP}$ ; указатель стека сначала уменьшается на 4 (это длина EIP в байтах), а затем используется;
- извлечение из памяти, адресуясь ЕА адреса точки входа в EIP:  $\text{EIP} = \text{mem}[\text{EA}]$ .

Для возврата используется два вида команд:

а) `ret` – возврат с помощью извлечения в указатель команд EIP адреса возврата из стека:  $\text{EIP} = \text{mem}[\text{ESP}++]$ ; указатель стека ESP увеличивается после использования на 4;

б) `ret N` – возврат, в ходе которого из вершины стека в указатель команд EIP сначала извлекается адрес возврата, а затем указатель стека увеличивается на число N, являющееся частью машинной команды; потребность в таком возврате будет объяснена в пункте 4.3.3.

Для подпрограмм принято различать формальные параметры и фактические. Формальные параметры – это программно-доступные компоненты (регистры, ячейки памяти), определяемые соглашением о местоположении параметров, которое обычно фиксируется в комментариях к подпрограмме. Фактические параметры – это значения, которые до выполнения команды `call` должны быть скопированы в программно-доступные компоненты, назначенные для формальных параметров.

В тексте ассемблер-программы подпрограмма начинается с оператора заголовка `PROC` и завершается оператором `ENDP`, перед каждым из которых фигурирует имя подпрограммы. Перед заголовком обычно формируют комментарий, который своим форматом сразу говорит, что это подпрограмма. В комментарии приводятся назначение подпрограммы и соглашение о параметрах и, возможно, возвращаемом значении, если это подпрограмма-функция.



#### Пример 4.17. Подсчет числа единичных бит в EAX

Самая простая подпрограмма подсчета числа единичных бит в регистре EAX, может иметь следующий вид:

```
; -----  
; BitCounter - Подсчет числа единичных бит в регистре EAX  
; Результат возвращается в EAX.  
; Подпрограмма портит EDX и ECX  
; -----  
BitCounter PROC  
    xor     edx, edx      ; Очистка счетчика единиц  
    mov     ecx, 32       ; ecx организует обработку 32 битов EAX  
loop:     shr     eax, 1   ; выдвигаем младший бит EAX в CF  
          adc     edx, 0   ; прибавляем CF к счетчику единиц  
    loop   loop          ; продолжать, если -ECX != 0  
    mov     eax, edx      ; возврат результата  
    ret  
BitCounter ENDP
```

Недостатком приведенной подпрограммы является порча регистров EDX и ECX, что требует сохранения содержимого этих регистров в любой фазе вызова, где эти регистры используются до вызова подпрограммы. Чтобы избавиться от этого недостатка, достаточно сохранить регистры сразу после входа в подпрограмму и восстановить перед выходом из нее. Наиболее просто организовать сохранение в стеке, поскольку это не потребует добавлять в программу какие-то именованные ячейки памяти.

#### Пример 4.18. Сохранение вспомогательных регистров в стеке

Подпрограмма, не портящая вспомогательные регистры, имеет вид:

```
; -----  
; BitCounter - Подсчет числа единичных бит в регистре EAX  
; Результат возвращается в EAX.  
; -----  
BitCounter PROC  
    push    edx  
    push    ecx  
    xor     edx, edx      ; Очистка счетчика единиц
```

```

        mov     ecx, 32      ; ecx организует обработку 32 бит EAX
loo:      shr    eax, 1      ; выдвигаем младший бит EAX в CF
          adc    edx, 0      ; прибавляем CF к счетчику единиц
        loop   loo          ; продолжать, если -ECX != 0
        mov     eax, edx     ; возврат результата
        ; восстанавливаем регистры из стека в обратном порядке
        pop     ecx
        pop     edx
        ret
BitCounter ENDP

```

Сохранение и восстановление регистров можно организовать также с помощью оператора USES, который записывается в заголовке процедуры после оператора PROC. Вслед за оператором USES в той же заголовочной строке перечисляются через пробел используемые регистры. Макропроцессор самостоятельно сгенерирует команды push после строки заголовка и команды pop перед командой ret.

#### **Пример 4.19. Использование USES для сохранения регистров**

Подпрограмма BitCounter с сохранением регистров в стеке с помощью оператора USES имеет вид:

```

; -----
; BitCounter - Подсчет числа единичных бит в регистре EAX
; Результат возвращается в EAX.
; -----
BitCounter PROC USES edx ecx
        xor     edx, edx     ; Очистка счетчика единиц
        mov     ecx, 32      ; ecx организует обработку 32 бит EAX
loo:      shr    eax, 1      ; выдвигаем младший бит EAX в CF
          adc    edx, 0      ; прибавляем CF к счетчику единиц
        loop   loo          ; продолжать, если -ECX != 0
        mov     eax, edx     ; возврат результата
        ; восстанавливаем регистры из стека в обратном порядке
        ret
BitCounter ENDP

```

Последовательность машинных команд, полученная в ходе трансляции этой процедуры, будет точно такой же, что после трансляции процедуры из примера 4.18.

У начинающих программистов при организации подпрограмм часто возникает ситуация, когда возврат из процедуры происходит не в точку возврата. Программа при этом чаще всего зависает. Причина такого явления – неодинаковое количество команд `push` и `pop` в процедуре. Если, например, число команд `push` в процессе выполнения окажется на одну больше, чем число команд `pop`, то в момент выполнения команды возврата `ret` в вершине стека будет находиться не адрес возврата, а данное, загруженное в него самой последней командой `push`. Если же на одну больше окажется команд `pop`, то адрес возврата будет извлечен из вершины стека самой последней командой `pop`. Во время трассировки подпрограммы в среде отладчика необходимо следить за состоянием стека и контролировать факт, что непосредственно перед выполнением команды `ret` в вершине стека находится адрес возврата.

При использовании оператора `USES` в исходном тексте процедуры отсутствуют команды `push` и `pop`, касающиеся регистров из списка параметров этого оператора. Во время трассировки в среде отладчика эти команды присутствуют и должны учитываться в процессе слежения за состоянием стека.

В зависимости от местоположения формальных параметров различают несколько способов передачи параметров, основными из которых являются передача через регистры, через стек и через статическую область памяти.

### **4.3.2 Передача параметров через регистры**

Передача параметров через регистры является самой распространенной при программировании на ассемблере. Подпрограмму `BitCounter` в примере 4.17 легко использовать для подсчета единиц в любом 32-разрядном числе. Для этого достаточно перед вызовом

подпрограммы скопировать это число в регистр EAX. Такое копирование как раз и обеспечивает передачу параметра через регистр.

Например:

```
; подсчет числа единиц в слове a32
mov eax, a32
call BitCounter
; подсчет числа единиц в регистре ESI
mov eax, esi
call BitCounter
```

Очень часто регистры выбираются не случайным образом, а с учетом характера обработки. Например, для копирования массива 32-разрядных чисел из одной области памяти в другую целесообразно использовать строчную команду `movsd` с префиксом повторения `rep`. Очевидно, что в процедуру копирования адреса исходной и целевой области памяти целесообразно передавать через ESI и EDI, а размер исходного массива — через ECX.

#### **Пример 4.20. Использование регистров в подпрограмме копирования**

```
; -----
; CpyRg – копирование ECX 32-разрядных слов из области данных,
; адресуемой через ESI, в область памяти с адресом в EDI
; -----
CpyRg PROC
    cld
    rep movsd
    ret
CpyRg ENDP
```

Рассмотрим два случая использования подпрограммы CpyRg: копирование 20-словного массива M1 в массив M2 и копирование 30-словного массива M3 в массив M4. Исходный текст соответствующей фазы вызова подпрограмм имеет вид:

```
; фаза вызова начинается с передачи параметров через регистры
lea ecx, 20
mov esi, m1
mov edi, m2
```

```
call CpuRg ; фаза вызова завершается собственно вызовом процедуры  
; начало фазы вызова для копирования M3 в M4  
lea ecx, 20  
mov esi, m1  
mov edi, m2  
call CpuRg ; фаза вызова завершается собственно вызовом процедуры
```

### 4.3.3 Передача параметров через стек

Этот способ передачи доминирует в системах программирования на языках высокого уровня. Именно через стек передаются параметры в программах, написанных на Паскале, Си и многих других языках. Знание и использование соответствующих соглашений по передаче параметров в подпрограммы нужно в смешанном программировании (ЯВУ + ассемблер), при создании эффективных библиотек процедур, а также при отладке программ на уровне CPU.

Передача через стек дает три основных преимущества перед передачей через регистры:

- число параметров может быть больше числа доступных регистров;
- легко организуются рекурсивные подпрограммы;
- механизм передачи параметров единообразен для произвольного количества аргументов, в т. ч. для случаев, когда количество фактических параметров в разных вызовах может быть различным.

Фаза вызова подпрограммы начинается с помещения в стек всех параметров с помощью команды `push`, после чего выполняется команда `call`. В подпрограммах на языке Паскаль в фазе ее вызова параметры помещаются в стек слева направо, а в подпрограммах на языке Си – справа налево.

#### Пример 4.21. Передача параметров в стек слева направо

Рассмотрим подпрограмму нахождения минимального числа из трех целочисленных параметров, которая на Паскале имеет следующий заголовок:

```
Function min(p1,p2,p3:integer)
```

Пусть требуется определить минимальное значение среди переменных `a32`, `b32`, `c32` согласно оператору `d32 := min(a32, b32, c32)`.

Фаза вызова подпрограммы будет иметь следующий вид:

; фаза вызова начинается с загрузки в стек первого параметра

```
push    a32
```

```
push    b32
```

```
push    c32
```

```
call    min ; собственно вызов
```

; здесь фаза вызова завершена и выполняется собственно оператор присваивания

; при соглашении, что функция возвращает значение через регистр `eax`

```
mov     d32, eax
```

С учетом того, что стек растет в сторону уменьшения адресов, сразу после входа в подпрограмму стек содержит такие данные (в левой колонке адрес, в правой – содержимое):

ESP:                   адрес возврата из подпрограммы

ESP + 4:               c32 на месте формального параметра `p3`

ESP + 8:               b32 на месте формального параметра `p2`

ESP + 12:              a32 на месте формального параметра `p1`

При обращении к формальным параметрам внутри подпрограммы можно использовать следующие адресные выражения:

`p1 – [ESP+12], p2 – [ESP+8], p3 – [ESP+4]`.

Однако, если бы наша процедура требовала сохранения в стеке рабочих регистров или организации в нем локальных переменных, то указатель стека при его таком использовании изменится, и смещения к параметрам нужно пересчитывать. Чтобы избежать этого, принято использовать базовый регистр `EBP` для базирования стековых данных подпрограммы. Это делается двумя командами:

```
push    ebp           ; сохранение ebp для защиты от порчи
```

```
mov     ebp, esp       ; инициализация базы стековых данных
```

После выполнения этих двух команд стековые данные могут адресоваться относительно неизменного `EBP` без необходимости подстраивать адресные выражения под возможно изменяющееся значение `ESP`. Стековые данные, адресуемые базовым регистром, называют стековым фреймом, или кадром. При отладке на уровне машинных команд, поиске ошибок в программе, для которой нет исходного кода, а также в технологиях обратного проектирования анализ стековых фреймов, адресуемый регистром `EBP`, является очень важным процессом.

Сразу после выполнения двух команд организации базирования стековых данных через регистр EBP, размещение стековых данных в случае передачи параметров вызова `min(a32,b32,c32)` слева направо имеет следующий вид:

```
EBP:           сохраненное старое содержимое EBP
EBP + 4:       адрес возврата из подпрограммы
EBP + 8:       p3
EBP + 12:      p2
EBP + 16:      p1
```

Пусть текст подпрограммы на ассемблере строится на основе следующего фрагмента на Паскале:

```
min:=p1;
if p2<min min:=p2;
if p3<min min:=p3;
```

Тогда ассемблер процедура будет выглядеть так:

```
min    PROC
        push ebp
        mov  ebp, esp
        mov  eax, [ebp+16];    min := p1
        cmp  [ebp+12], eax; if p2 < min
        jge  LP3 ; перепрыгиваем через смену минимума
        mov  eax, [ebp+12];    min := p2
LP3:    cmp  [ebp+16], eax; if p3 < min
        jge  LRET ; перепрыгиваем через смену минимума
        mov  eax, [ebp+16];    min := p3
LRET:   ; фаза выхода из подпрограммы начинается с восстановления ebp
        pop  ebp
        ret  3*4 ; возврат с освобождением области параметров
min    ENDP
```

Команда возврата с параметром (формат `RET N`) используется в приведенной процедуре для того, чтобы после выхода из процедуры указатель стека ESP указывал на вершину стека, имевшую место до начала фазы вызова. Эта команда после извлечения адреса возврата из стека увеличивает ESP на значение своего параметра N. В частности, приведенная выше фаза вызова процедуры `min` для реализации оператора Паскаль-программы `d32 := min(a32, b32, c32)`

будет корректной, если указатель стека ESP в точке пересылки eax в d32 будет содержать ровно то значение, что было в нем до команды загрузки в стек первого параметра.

В языке Си одна и та же подпрограмма может вызываться с различным числом параметров. Самая популярная программа, обладающая таким свойством, это printf. Указанное свойство приводит к необходимости придерживаться четырех правил построения подпрограммы:

а) поскольку только в фазе вызова известно число используемых параметров, освобождение области стека выполняется в каждой фазе вызова после оператора call;

б) выход из подпрограммы выполняется обычной командой ret без параметра;

в) в подпрограмму всегда передается число N используемых фактических параметров через первый дополнительный аргумент подпрограммы;

г) параметры в фазе вызова передаются справа налево, так чтобы число обрабатываемых параметров загружалось в стек самым последним – это обеспечивает нахождение этого числа сразу после адреса возврата при любом количестве параметров в фазе вызова.

Рассмотрим выполнение этих правил на примере.

#### **Пример 4.22. Передача разного числа параметров в стек**

Пусть, например, требуется разработать подпрограмму нахождения минимального числа среди различного количества параметров функции. На языке Си возможность иметь разное число параметров задается в заголовке функции многоточием в конце списка параметров. Функция нахождения минимума может быть объявлена следующим образом:

```
int min(int num, int p1, int p2, ...)
{ int m = p1, *pp = &p2, n = num - 1;
  do {
    if (*pp < m) m = *pp;
    pp++;
  } while (--n);
  return m;
}
```



В этой функции первый параметр задает количество параметров, среди которых ищется минимум. Многоточие в заголовке функции сообщает компилятору, что вызов этой функции может иметь 3 и более параметра.

Реализация приведенной программы на ассемблере имеет вид:

```
min    proc
        push ebp ;   создаем в указатель на стековые данные
        mov  ebp, esp
        ; int m = p1, *pp = &p2, n = num - 1;
        ; eax = m, edx = pp, ecx = n
        mov  eax, [ebp+12]
        lea  edx, [ebp+16]
        mov  ecx, [ebp+8]
        dec  ecx          ; do{
L_do:   cmp  eax, [edx] ;   if (*pp < m)
        jle  next
        mov  eax, [edx] ;       m = *pp;
next:   add  edx, 4          ;   pp++
        dec  ecx          ; } while (--n);
        jne  L_do
        ; return m – минимум уже в eax
        pop  ebp
        ret
min    endp
```

Рассмотрим два случая вызова этой подпрограммы с различным количеством параметров:  $x = \min(2, a32, b32)$ ;  $y = \min(3, a32, b32, c32)$ . Фазы соответствующих вызовов будут иметь вид, представленный ниже.

```
; x = min(2, a, b)
push b32
push a32
push 2
call min
add esp, 3*4 ; удаление из стека 3 параметров
; x = min(3, a, b, c)
push c32
push b32
push a32
push 3
```

```
call min
```

```
add esp, 4*4 ; удаление из стека 4 параметров
```

Состояние стека в подпрограмме после инициализации `ebp` в последнем входе в подпрограмму будет таким:

`EBP`:            сохраненное старое содержимое `EBP`

`EBP + 4`:       адрес возврата из подпрограммы

`EBP + 8`:       3 – число параметров

`EBP + 12`:      `a32`

`EBP + 16`:      `b32`

`EBP + 20`:      `c32`

Очевидно, что в предыдущем входе в подпрограмму состояние стека отличалось числом параметров (2, а не 3) и отсутствием в стеке параметра `c32`.

#### 4.3.4 Передача параметров через статические данные

Передача параметров через статические данные предполагает наличие области памяти, в которую перед вызовом подпрограммы копируются фактические параметры. В общем случае такой механизм сильно уступает передаче параметров через стек:

- требуется создание именованных областей памяти;
- статические данные занимают память все время исполнения программы, в то время как часть стека, занимаемая параметрами, после выхода из подпрограммы освобождается;
- команды копирования и обращения к данным содержат довольно длинные абсолютные адреса, в то время как при обращении к стеку в командах фигурируют чаще всего однобайтные смещения к `EBP`;
- не поддерживается рекурсия.

Несмотря на указанные недостатки, передача через статические данные может иметь смысл, если она сочетается с адресацией записей через указатели, передаваемые в подпрограмму через стек или регистры. Чисто технически здесь полное совпадение с передачей указателя на запись через регистр или стек, но если сама запись формируется как интерфейс между вызывающим модулем и подпрограммой, то правильнее все же считать это передачей через статические данные. Наибольший положительный эффект от такой передачи параметров получается в

случае, когда часть фактических данных из фазы одного вызова используются в фазе другого вызова либо той же самой подпрограмм, либо какой-то другой, базирующей на таком же интерфейсе.

#### 4.3.5 Организация локальных переменных

Лучшее место для локальных переменных – это регистры, поскольку обработка регистровых данных выполняется эффективнее обработки данных в памяти. Однако в архитектуре x86 слишком мало регистров и не уместяющиеся в них локальные переменные размещают в стеке. Перед выходом из подпрограммы соответствующую область стека освобождают.

Резервирование в стеке области памяти для локальных переменных достигается очень просто – из указателя стека ESP вычитается размер, необходимый для всех переменных, размещаемых в стеке. Тем самым стековый фрейм расширяется в сторону отрицательных смещений относительно базового регистра EBP.

##### Пример 4.23. Локальные переменные в стеке

Пусть, например, требуется написать процедуру, которая переставляет местами 2 строки длиной не более 32 байт, например, в целях упорядочивания. На языке Си такая процедура будет иметь вид:

```
void str_chng ( char * s1, char *s2 )
{ char buf [30]; // буфер обмена
  strcpy ( buf, s1); // скопировать 1-ю строку в буфер
  strcpy ( s1, s2 ); // скопировать 2-ю строку на место 1-ой
  strcpy ( s2, buf ); // скопировать буфер на место 2-ой строки
}
```

На ассемблере эта процедура имеет следующий вид:

```
-----
; void str_chng ( char * s1, char *s2 ) – перестановка строк s1 и s2
;-----

str_chng  proc
    push    ebp      ; создаем указатель на стековые данные
    mov     ebp, esp
    sub     esp, 32   ; создаем buf в стеке
```

```

; strcpy ( buf, s1); // скопировать 1-ю строку в буфер
push    [ebp+8]
lea     eax,    [ebp-32]
push    eax
call    strcpy
add     esp, 8
; strcpy ( s1, s2); // скопировать 2-ю строку на место 1-й
push    [ebp+12];
push    [ebp+8]
call    strcpy
add     esp, 8
; strcpy ( s2, buf); // скопировать буфер на место 2-й строки
lea     eax, [ebp-32]
push    eax
push    [ebp+8]
call    strcpy
add     esp, 8
mov     esp, ebp ; удаление локальных переменных из стека
pop     ebp      ; восстановление EBP
ret
str_chng endp

```

### 4.3.6 Рекурсивные подпрограммы

Рекурсия заключается в том, что во время выполнения подпрограммы происходит вызов ее самой. Каждый очередной рекурсивный вызов порождает новый стековый фрейм, и в какой-то момент в стеке может находиться несколько стековых фреймов, относящихся к разным фазам вызова. Осмысление логики использования нескольких стековых фреймов очень важно для отладки и эффективной реализации рекурсивных процедур независимо от уровня языка программирования. Это осмысление лучше всего рассмотреть на примере.

#### Пример 4.23. Рекурсивная подпрограмма генерации текста

Пусть требуется разработать процедуру генерации текста `gentxt`, на вход которой поступает исходный текст, содержащий конструкции вида *#ключевое слово#*, и указатель результирующего текста. Во время генерации в результирующий текст копируются все символы вне таких конструкций, а

вместо конструкции вставляется текст, извлекаемый из словаря по ключевому слову `kword` с помощью вызова процедуры `getval(kword)`. Предполагается, что в словаре, каждый элемент которого содержит пару «ключевое слово – результирующая фраза», результирующая фраза может также содержать конструкции вида *#ключевое слово#*.

Суть рекурсивной организации процесса генерации в том, что в точке подпрограммы `gentxt`, где выделено ключевое слово, вызывается `gentxt` с передачей в качестве исходного текста результата вызова процедуры `getval`. Процедура `gentxt` возвращает указатель на точку результирующего текста, в которую вписан байт `'\0'` как признак конца строки. Функция может вернуть `NULL`, если ключевое слово не найдено, поэтому в подпрограмме `gentxt` предусматривается контроль на нулевое значение указателя исходной строки.

Процедура `gentxt` на языке Си представлена ниже.

```
// gentxt(src, dest) - копирует из src в dest с рекурсивной подстановкой из словаря
// Возвращает продвинутое значение указателя формируемой строки dest
char * gentxt(char *src, char *dest)
{
    char kword[16]; // буфер ключевого слова
    char *str, letter; // сканер ключевого слова и текущий символ
    //Защита от пустой строки
    if(src == NULL) return dest;
    // Цикл копирования src в dest с отслеживанием
    #слово#
    while((letter = *src++) != '\0')
    {
        if (letter == '#')
        {
            // Цикл формирования ключевого слова
            str=kword;
            while( (letter = *src++) != '#')
                *str++ = letter;
            *str = '\0'; // завершаем kword символом '\0'
            // рекурсивный вызов с передачей результата getval
            dest = gentxt(getval(kword), dest);
        }
        else *dest++ = letter; // просто копирование
    }
    *dest = '\0'; // завершаем результирующий текст
    // возврат продвинутого указателя для продолжения генерации
    return dest;
}
```

При использовании передачи параметров через стек на ассемблере эта процедура может быть такой:

```
;-----  
; gentxt(char *src, char *dest) – генерация текста путем копирования из src в dest  
; с подстановкой взамен конструкции #ключевое слово# извлекаемого из словаря  
; значения с помощью процедуры char *getval(char *kword)  
;-----  
gentxt proc  
    push ebp  
    mov  ebp, esp  
    sub  esp, 16      ; создание локального буфера kword  
    push esi          ; в esi будет указатель src  
    push edi          ; в edi будет указатель dest  
    mov  esi, dword ptr [ebp+8] ; esi = src  
    mov  edi, dword ptr [ebp+12]; edi = dest  
    test esi, esi     ; if (src == NULL)  
    je   LRet         ;   идти к return dest  
    mov  eax, edi  
LWhile: ; while((letter = *src++) != '\0')  
        lodsb          ; al= *esi++  
        test al, al    ; if(letter == '\0')  
        je   LRet     ;   идти к return dest;  
        cmp  al, 35    ; if (letter != '#')  
        je   LCopy    ; идти к просто копированию без подстановки  
        ; Формирование ключевого слова  
        lea  edx, [ebp-16] ; edx = str  
LKWord: ; Тело цикла формирования ключевого слова  
        lodsb          ; letter = *src++  
        cmp  al, 35    ; if (letter == '#')  
        je   short KWordDone ; выход из цикла формирования kword  
        mov  byte ptr [edx], al ; *str = letter  
        inc  edx        ; *str++  
        jmp  short LKWord  
        mov  byte ptr [edx], 0 ; *str = '\0';  
        ; Рекурсивный вызов dest = gentxt(getval(kword), dest);  
        push edi ; передача параметра dest  
        lea  eax, dword ptr [ebp-16] ; передача параметра kword
```

```

    push eax ;
    call getval
    pop ecx ; извлекаем из стека str
; dest осталось в стеке от вызова getval, оставляем его для вызова gentxt
    push eax ; передача адреса, возвращенного из из getval
    call gentxt
    add esp, 8 ; удаление двух параметров в стеке
    mov edi, eax ; фиксируем в desc продвинутый адрес
    jmp short LWhile ; в начало тела цикла копирования
; Просто копирование без подстановки
LCopy: stosb ; *edi++ = letter – копирование без подстановки
    jmp LWhile
LRet: ; *dest = '\0' – завершение результирующего текста
    mov byte ptr [edi], 0
; return dest - возврат значения;
    mov eax, edi
; восстановление регистров
    pop esi
    pop edi
    mov esp, ebp
    pop ebp
    ret
gentxt endp

```

Состояние стековой памяти в точке LWhile для случая, когда функция gentxt вызвана рекурсивно третий раз, имеет вид (справа слова в стеке):

```

; третий стековый фрейм
ESP:      значение ESI во втором вызове
ESP + 4:   значение EDI во втором вызове
EBP - 16:  буфер kword третьего вызова
EBP:      база стекового фрейма второго вызова
EBP + 4:   адрес возврата из третьего вызова gentxt
EBP + 8:   src – первый параметр третьего вызова
EBP + 12:  dest – второй параметр третьего вызова
; второй стековый фрейм
ESP + 28:  значение ESI в первом вызове
ESP + 32:  значение EDI в первом вызове
EBP - 16:  буфер kword второго вызова
EBP:      база стекового фрейма первого вызова

```

EBP + 4: адрес возврата из второго вызова gentxt  
 EBP + 8: src – первый параметр второго вызова  
 EBP + 12: dest – второй параметр второго вызова  
 ; первый стековый фрейм  
 ESP: значение ESI до первого вызова  
 ESP + 4: значение EDI до первого вызова  
 EBP – 16: буфер kword первого вызова  
 EBP: содержимое EBP до первого вызова  
 EBP + 4: адрес возврата из первого вызова gentxt  
 EBP + 8: src – адрес исходной строки  
 EBP + 12: dest – адрес результирующей строки

#### 4.4. Вопросы и упражнения

1) Имеется последовательность операторов:

```

MOV  AL,    07Fh
INC  AL
JNO  L
DEC  AL

```

L:

Какое число будет находится в регистре AL в точке L?

2) Пусть после каждой команды сложения беззнаковых чисел требуется увеличивать на единицу переменную CF\_COUNT в случае, если при сложении возник перенос. Можно ли избежать применения команды ветвления при реализации такой операции подсчета числа случаев беззнакового переноса?

3) Пусть имеется такая последовательность операторов:

```

MOV  EDX, 16
MOV  ECX, EDX
L1:  TEST EAX, 11b
      JE   L2
      SHR  EAX, 2
      LOOP L1
L2:  SUB  EDX, ECX

```

Какие значения примут регистры EAX и ECX после выполнения приведенного фрагмента программы при исходном значении EAX = 555h?

4) Определите смысл приведенного ниже фрагмента ассемблер-программы и запишите его на языке Си:



```

        LEA     ESI, AR
        MOV     ECX, 40
L:      CMP     EAX, [ESI]
        CMOVL   EAX, [ESI]
        ADD     ESI, 4
        LOOP    L

```

5) Реализовать на ассемблере x86 (IA-32) ветвление в алгоритме поиска минимального числа  $\text{if}(x < \text{min}) \text{ min} = x$ , где  $x$  и  $\text{min}$  задаются описанием:

```
long long x, min; // тип long long имеет длину 64 бит.
```

6) Разработать подпрограмму LenMeandr подсчета длины такой последовательности бит в 64-разрядном коде, которая располагается от младшего разряда к старшему и имеет характер чередования разрядов (1, 0, 1, 0, 1, 0...). Например, при передаче в подпрограмму переменной, объявленной на языке Си как `long long x = 0x155`, подпрограмма должна вернуть значение 5. Операнд передается в подпрограмму через регистровую пару EDX\_EAX (младший разряд EAX содержит первый разряд последовательности), результат возвращается через AL.

7) Широко распространенным примером рекурсивных подпрограмм для многих языков программирования является вычисление факториала. На языке Си такая функция для случая, когда аргумент функции не превосходит 12, может быть представлена следующим образом:

```

long fact(long n) {
    if (n == 0) return 1;
    return n * fact(n - 1);
}

```

Реализуйте на ассемблере процедуру fact двумя способами:

а) используя рекурсию; б) используя цикл. Выполните сравнение полученных реализаций по затратам времени и памяти программы, а также сложности отладки.

8) Аналогично предыдущей задаче требуется разработать фрагменты двух ассемблер-программ, но операнды Y, X, Z представляют собой упакованные десятиразрядные беззнаковые двоично-десятичные числа (десять десятичных цифр).

## 5. Организация ввода-вывода

### 5.1. Базовые средства ввода-вывода

Чаще всего в ассемблер-программах применяется консольный ввод-вывод текстовых строк, операции которого рассматриваются ниже.

Консольный ввод-вывод осуществляется либо с помощью функций операционной системы, либо посредством библиотечных процедур. Библиотечные процедуры обычно имеют более простой интерфейс. В процедуре `puts` примера 1.9 для вывода строки использована библиотечная процедура `StdOut`, исходный текст которой легко найти в директории `m32lib` MASM32 SDK. Исходный модуль процедуры `StdOut` находится в файле `stdout.asm`, который содержит следующий текст:

```
.386
.model flat, stdcall
option casemap :none    ; case sensitive
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
StrLen PROTO :DWORD
.code
StdOut proc lpszText:DWORD
LOCAL hOutPut :DWORD
LOCAL bWritten :DWORD
LOCAL sl      :DWORD
invoke GetStdHandle,STD_OUTPUT_HANDLE
mov hOutPut, eax
invoke StrLen,lpszText
mov sl, eax
invoke WriteFile,hOutPut,lpszText,sl,\
        ADDR bWritten,NULL
mov eax, bWritten
ret
StdOut endp
end
```

Обращение к функции Windows API в этой процедуре происходит с помощью макровывода `invoke WriteFile`, организующего вызов подпрограммы `WriteFile` с передачей ей в качестве параметров файлового дескриптора консольного вывода `stdout`, адрес выводимого текста `lpzText` и его длину, а также адрес слова для кода возврата `bWriteln`. Процедура `WriteFile` способна выводить данные не только в консоль, но и в любой файл, дескриптор которого получен с помощью процедуры `CreatFile`.

Сама библиотечная функция `StdOut` получает в качестве своего параметра только адрес выводимой строки, скрывая сложность вызова собственно функции Windows API. Выводимая строка в качестве символа завершения должна иметь байт 0 подобно `'\0'` в строках программ на языке Си.

Имея в наличии упрощенные процедуры ввода-вывода, естественно строить общение разрабатываемой программы с пользователем с помощью именно их.

Ввод строки текста выполняется процедурой `StdIn MASM32 SDK`, формат вызова которой имеет следующий вид:

```
invoke StdIn ADDR Buffer, LENGTHOF Buffer.
```

В качестве первого параметра в этом вызове используется адрес статического массива байтов, куда помещается вводимая строка, а в качестве второго – размер этого массива. Следует отметить, что процедура `StdIn` не только скрывает сложность вызова используемой функции API `ReadFile`, но и заменяет на байт 0 первый символ двухсимвольного завершителя строки `CR LF`, который попадает в буфер ввода по нажатию клавиши `Enter`.

Использование адреса для строки ввода неизбежно, однако при выводе текстовых констант целесообразно избежать дополнительных усилий по размещению константы в памяти. Это достигается за счет применения макроса с объявлением области данных, помеченной локальной меткой. Данный прием демонстрируется ниже в примере 5.1.

При вводе-выводе кириллицы консольные процедуры Windows используют кодировку `CP866`, в то время как инструментальные средства

разработки программ в среде Windows используют кодировку Windows-1251. Это порождает проблемы, которые решаются с помощью процедур перекодировки. В примере 1.9 проблема решена использованием процедуры CharToOemA. Если бы процедуры перекодировки не было, то строки массива txt после вывода были бы нечитабельны, поскольку текстовый редактор среды RadAsm использует кодировку Win-1251, что предопределяет генерацию кодов символов именно в этой кодировке в директивах размещения строк с метками s1, s2, s3. Если в редакторе исходного текста использовать кодировку CP866, то размещенные в программе текстовые строки можно выводить без перекодировки.

Ввод строк, содержащих символы кириллицы, не требует перекодировки, если в дальнейшей обработке не задействованы тексты в иной кодировке, нежели CP866. Если вводимая строка используется, например, в качестве образца для поиска строк, созданных в иной кодировке, то ее необходимо перекодировать, как показано в следующем примере.

#### **Пример 5.1. Ввод и вывод строк с использованием кириллицы**

Пусть требуется написать программу, которая выполняет следующие шаги:

- а) вводит с предварительным запросом ключевое слово в текстовый буфер inword;
- б) ищет во введенном слове четырехсимвольную ключевую последовательность, объявленную в программе под именем key;
- в) в зависимости от результата поиска выводит на экран либо сообщение «Слово W - ключевое», либо сообщение «Слово W не является ключевым»; при этом в качестве W должно быть выведено то слово, что было введено на первом шаге;
- г) выводит запрос «Повторить (0-нет, 1-да): » и в зависимости от ответа либо передает управление на начало программы, либо завершает ее.

Особенностью программы является использование вводимого слова в строке вывода и в операции сравнения с ключевым словом. Очевидно, что для вывода введенное в кодировку CP866 слово не должно перекодироваться, а

для сравнения с ключом, имеющим кодировку Win-1251, перекодировка нужна. Ответа на запрос «Повторить» также не нужно перекодировать, поскольку в нем нет символов кириллицы. Поскольку в программе несколько раз используется вывод текстовой константы с кириллицей, целесообразно оформить этот вывод как макрос, содержащий локальный массив для константы, вызов процедуры перекодировки и вызов процедуры вывода.

Код программы приводится ниже.

```
.386
.model flat, stdcall
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
include \masm32\include\masm32.inc

includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\masm32.lib
; Макрос вывода константной строки
; с предварительной перекодировкой
PutsRuConst MACRO RuText: VARARG
    LOCAL LocText
    .data
        LocText db RuText, 0
    .code
        invoke CharToOemA, ADDR LocText, ADDR TextCP866
        invoke StdOut, ADDR TextCP866
ENDM
.data
TextCP866      db 50 dup(0) ; текст, выводимый на консоль
InText         db 20 dup (0) ; сюда вводится слово
InText1251     db 20 dup (0) ; перекодированный InText
Key            db "прог"      ; ключ в кодировке Win-1251
.code
start:
    PutsRuConst 'Введите слово: '
    invoke StdIn, ADDR InText, LENGTHOF InText
    Invoke OemToCharA, ADDR InText, ADDR InText1251
    mov     eax, dword ptr Key ; ВЗЯТЬ КЛЮЧ
```

```

        cmp      dword ptr InText1251, eax ;сравнение
        je       short ok
        PutsRuConst 'Слово "'
        invoke StdOut, ADDR InText
        PutsRuConst '"' не является ключевым.'
        jmp short con
ok:      PutsRuConst 'Слово "'
        invoke StdOut, ADDR InText ;вывод без перекодировки
        PutsRuConst '"' - ключевое.'
con:     PutsRuConst ' Продолжить (0-нет, 1-да): '
        invoke StdIn, ADDR InText, LENGTHOF InText
        cmp      byte ptr InText, '1'
        je start
fin:     invoke ExitProcess,0
end start

```

Если бы программа не предполагала повторное исполнение, то в ней можно было обойтись без области данных результата перекодировки TextCP866, размещая его в LocText. Каждый вызов макроса PutsRuConst порождает в ходе макрогенерации (но не в ходе выполнения программы) новую область данных сегмента data, содержащую фактический параметр макроса. При использовании LocText также и для результата перекодировки, повторение процесса «ввод-обработка-вывод» приведет к повторной перекодировке внутри каждого продукта макрогенерации LocText, что недопустимо.

В приведенной программе вместо традиционного сравнения текстовых строк для обнаружения ключа во введенном слове использовано сравнение 32-разрядных слов. Такое упрощение удалось осуществить только потому, что ключ четырехсимвольный, т. е. умещается в 32-разрядное слово. Для интерпретации меток байтов key и InText1251 как адресов 32-разрядных слов использован спецификатор типа «dword ptr».

## **5.2. Ввод целочисленных данных и преобразование из внешнего представления во внутреннее**

Во внешнем представлении целочисленные данные являются текстовыми строками, т. е. последовательностями кодов символов. Внутреннее представление – это бинарный код, в котором числа со знаком кодируются в дополнительном коде (старший бит кодирует «плюс/минус»), а числа без знака используют все двоичные разряды для представления положительного числа. Таким образом, ввод числовых данных сводится к обычному вводу строк с последующим преобразованием текстовых представлений чисел во внутреннее бинарное представление.

Для внутреннего представления всегда должна быть определена длина числа в байтах. Допустим, что длина равна 1 байт. Тогда ввод беззнакового «255» и знакового «-1» должен дать одно и то же внутреннее представление байта, в котором все 8 разрядов равны единице. В исходном тексте программы используется внешнее представление, т. е. символьное, а не бинарное. Сформировать байт с единицами во всех 8 разрядах можно несколькими способами:

```
Byte255 db 0FFh, -1, 11111111b, 255, 377o.
```

Приведенный оператор породит 5 одинаковых байтов, готовых для обработки самых коротких чисел с помощью команд целочисленной арифметики. Преобразование параметров приведенного оператора db из текстового в бинарное представление осуществляет ассемблер. Если же мы вводим данные либо с консоли, либо из текстового файла, то преобразование их во внутреннее представление должна делать наша программа, опираясь на два соглашения: размер внутреннего представления и систему счисления внешнего представления.

Влияние соглашения о размере внутреннего представления легко проиллюстрировать случаем ввода числа «-1». Если размер равен одному байту, то при вводе должен быть порожден байт с единицами во всех 8 разрядах. Если размер равен 2 байтам, то таких байтов будет два. Если

вводится беззнаковое число 255, то в случае многобайтового внутреннего представления только младший байт должен содержать все 8 единиц – все старшие байты должны содержать нули.

При вводе чисел в различной системе счисления принято вводить только цифры без явного указания системы счисления наподобие 'h' / 'b' / 'o', имеющего место в ассемблер-программе. Соглашение о системе счисления реализуется в программе через использование вполне конкретной подпрограммы, например: десятичный ввод байтов, двоичный ввод слов, шестнадцатеричный ввод двойных слов и т. п.

Все преобразования между внешним и внутренним представлениями для любой q-ичной позиционной системы счисления базируются на разложении, для n-разрядного числа в q-ичной системе имеющем следующий вид:

$$D = d_{n-1} * q^{n-1} + d_{n-2} * q^{n-2} + \dots + d_1 * q + d_0 .$$

Для десятичной системы разряд  $d_0$  хранит единицы, разряд  $d_1$  – десятки и т. д. При вводе чисел входные строки содержат символьные представления отдельных цифр  $d_i$ , т. е. в каждом байте находится код символа цифры. В приведенном выше разложении предполагается числовое представление, чтобы, например, при вводе строки «255» можно было получить внутреннее представление числа  $255 = 2*100 + 5*10 + 5$ . Это означает, что каждый введенный символ цифры  $dchar$  необходимо преобразовать в числовой эквивалент  $d$ .

Чтобы избежать возведение в степень, применяют схему Горнера:

$$D = (..(d_{n-1} * q + d_{n-2}) * q + \dots + d_1) * q + d_0,$$

которая для примера с числом 255 дает следующее:  $255 = (2*10+5)*10+5$ .

Исходное число перед преобразованием во внутреннее представление чаще всего является массивом символов, обычно введенным с консоли с помощью программы ввода строк. Первый символ строки содержит либо знак числа '-', либо старший разряд числа. В случае использования языка Си со строками, завершающимися байтом 0, формирование внутреннего представления из внешнего может быть таким, как представлено ниже:



```

D = 0;
for (char *p = strnum; *p != '\0'; p++)
    D = D * q + digit(*p);

```

В приведенном фрагменте строка `strnum` содержит исходное число, а функция `digit` превращает код цифры в ее числовой эквивалент.

Кодировка цифр 0..9 устроена так, что все они располагаются в таблице кодов подряд. Это означает, что для получения числового эквивалента достаточно от кода цифры отнять код цифры «0». Иначе говоря, вместо `digit(*p)` можно применить `(*p - '0')`.

В случае, когда  $q > 10$  цифры с числовыми эквивалентами 10 и более представляются буквами, например, для шестнадцатеричной системы это символы из диапазона 'A'..'F' либо 'a'..'f'. Коды символов 'A' или 'a' не следуют за кодом цифры '9', что усложняет реализацию функции `digit`. Если используются символы 'a'..'f', то для них числовой эквивалент цифры вычисляется очевидным образом:  $\text{digit}(\text{dchar}) = \text{dchar} - 'a' + 10$ . Это справедливо для любой системы счисления, цифры которой, начиная с десятки, кодируются последовательно символами 'a', 'b', ... .

Таким образом, функцию `digit` для случая  $q > 10$  легко вычислить следующим образом:

```

dchar -= '0'; if (dchar > 9) dchar -= 'a' - '0' - 10.

```

Если система счисления кратна степени 2, т. е.  $q = 2^k$ , то вместо умножения на  $2^k$  обычно используют сдвиг влево на  $k$  разрядов. В логической интерпретации этого процесса можно заметить, что при сдвиге влево в формируемом внутреннем бинарном представлении образуется  $k$  младших разрядов, равных нулю, к которым в дальнейшем согласно схеме Горнера подсуммируется числовой эквивалент цифры, имеющей длину не более  $k$  разрядов. При формировании многобайтового внутреннего представления подсуммирование очередной цифры можно выполнять только в младшем байте, поскольку при сложении с нулями в  $k$  младших разрядах переноса не возникает.

Рассмотрим процесс преобразования на примерах.

### Пример 5.2. Разработка процедуры преобразования десятичного числа

Рассмотрим задачу разработки процедуры, которая преобразует строчное представление десятичного числа, адресуемого регистром ESI, в 32-разрядное бинарное число, возвращаемое через EAX. Программа приведена ниже.

```
; -----  
; StrToDec32 – Преобразование в 32-разрядное слово десятичного числа  
; со знаком, представленного строкой, адресуемой ESI.  
; Результат возвращается в EAX. Подпрограмма портит регистры ECX, BL, EDX.  
; -----  
StrToDec32 proc  
    mov     edi, 10      ; q = 10  
    xor     eax, eax     ; D = 0  
    xor     bl, bl       ; по умолчанию число положительно  
    mov     cl, [esi]    ; cl = *p  
    inc     esi          ; p++  
    or      cl, cl       ; if(*p == '\0') break;  
    jz      cycle  
    cmp     cl, '-'  
    jne     Lsum         ; идти к подсуммированию первой цифры  
    inc     bl           ; отметить отрицательность  
cycle: mov     cl, [esi] ; cl = *p  
    inc     esi          ; p++  
    or      cl, cl       ; if(*p == '\0') break;  
    jz      Ldone  
    mul     edi  
Lsum:  sub     cl, '0'  
    add     al, cl  
    jmp     short cycle  
Ldone: test    bl, 1      ; если отрицательное число  
    jz      Lret  
    neg     eax          ; то eax = 0 - eax  
Lret:  ret  
StrToDec32 endp
```

### Пример 5.3. Разработка процедуры преобразования шестнадцатеричного числа

Пусть требуется разработать процедуру преобразования строчного представления шестнадцатеричного числа, адресуемого регистром ESI, в 32-разрядное бинарное, возвращаемого через EAX. Программа имеет вид:

```
; -----  
; StrToHex32 – Преобразование в 32-разрядное слово шестнадцатеричного числа,  
; представленного строкой с адресом в регистре ESI.  
; Результат возвращается в EAX. Подпрограмма портит регистр ECX.  
; -----  
StrToHex32 proc  
        xor     eax, eax    ; D = 0  
cycle:  mov     cl, [esi]   ; cl = *p  
        inc     esi        ; p++  
        or      cl, cl      ; if(*p == '\0') break;  
        jz      Lret  
        shl     eax, 4  
        sub     cl, '0'     ; десятичная цифра => число  
        cmp     cl, 9       ; ЕСЛИ символ не 'a'..'f'?  
        jle     Lsum        ; ТО идем к подсуммированию  
        sub     cl, 'a'-'0' - 10 ; ИНАЧЕ коррекция цифры  
Lsum:   add     al, cl       ; подсуммирование цифры  
        jmp     short cycle  
Lret:   ret  
StrToHex32 endp
```

### 5.3. Вывод целочисленных данных и преобразование из внутреннего представления во внешнее

Вывод целочисленных данных предполагает преобразование бинарных внутренних представлений в строковое представление для вывода на консоль или в файл. Правило такого преобразования легко выводится из степенного представления числа в позиционной системе

счисления. Однако теперь у нас задача обратная. При преобразовании из внутреннего представления имеется только бинарное число  $D$  и нужно каким-то образом получить все цифры  $d_0, d_1, \dots, d_{n-1}$ . Легко догадаться, что целочисленное деление числа  $D$  на основание системы счисления  $q$  дает в остатке младшую цифру  $d_0$ , то есть:

$$D \% q = d_0.$$

$$D / q = d_{n-1} * q^{n-2} + d_{n-2} * q^{n-3} + \dots d_2 * q + d_1.$$

Очевидно, что если результат первого деления снова поделить на  $q$ , то в остатке получится следующая цифра  $d_1$ . Индуктивный вывод позволяет сформулировать следующий цикл формирования последовательности цифр числа:

```
for( i = 0; D != 0; i++)  
{ d[i] = D % q;  
  D = D / q;  
}
```

При получении текстового представления числа необходимо учитывать, что в текстовом представлении фигурируют коды символов цифр и числа записываются старшими разрядами вперед. Казалось бы, что учет указанных обстоятельств реализуется простой заменой первого оператора приведенного выше цикла на оператор присваивания  $str[n-i-1] = dchar(D \% q)$ , где  $str$  – формируемая строка,  $dchar$  – функция преобразования числового эквивалента цифры в код соответствующего символа, которая при  $q \leq 10$  реализуется простым прибавлением  $d[i] + '0'$ .

Однако разрядность  $n$  формируемого числа обычно заранее неизвестна. Если заранее задать максимально возможную разрядность и организовать выход из цикла `for` не по нулевому значению  $D$ , а по формированию всех разрядов, то мы получим числа с префиксными нулями. Для формирования шестнадцатеричных, восьмеричных и двоичных дампов это вполне приемлемо, однако для десятичных чисел чаще всего недопустимо. Ситуация легко решается загрузкой цифр  $dchar(d[i])$  в стек младшими разрядами вперед с последующим извлечением в массив старшими разрядами вперед. Рассмотрим этот процесс на примере.

**Пример 5.4. Разработка процедуры преобразования 32-разрядного числа  
в десятичное текстовое представление с возможным знаком**

Пусть требуется разработать процедуру, которая преобразует число в регистре EAX в строку, адресуемую регистром EDI. Процедура имеет вид:

```
; -----  
; Dec32ToStr – преобразование 32-разрядного слова EAX в десятичное текстовое  
; по адресу EDI. В EAX возвращается разрядность числа. Портит EDX.  
; -----  
Dec32ToStr proc  
    push    ecx  
    push    esi  
    mov     esi, 10      ; делить будем на esi  
    ; формирование знака '-', если число отрицательно  
    test    eax, eax     ; ЕСЛИ положительное  
    jns     Lbegin       ; ТО к началу преобразования  
    neg     eax          ; отрицательное => прямой код  
    mov     byte ptr [edi], '-' ; вписать '-' в строку  
    inc     edi  
Lbegin: xor     ecx, ecx   ; счетчик полученных цифр  
cycle:  xor     edx, edx   ; div делит EDX:EAX / ESI  
    div     esi          ; edx = остаток, т.е. цифра числа  
    push    edx          ; цифру помещаем в стек  
    inc     ecx          ; счет цифр  
    test    eax, eax     ; ЕСЛИ не все цифры получены?  
    jnz     cycle        ; ТО продолжить  
    ; теперь в стеке все цифры от старших к младшим  
    mov     eax, ecx      ; формируем код возврата  
Lpop:   pop     edx        ; извлечение из стека от последних цифр  
    add     dl, '0'       ; превращение цифры в символ  
    mov     [edi], dl     ; размещение в результирующей строке  
    inc     edi  
    loop    Lpop          ; ЕСЛИ не все извлечено, то продолжать  
    mov     [edi], byte ptr 0 ; '\0' в конец  
    pop     esi  
    pop     ecx  
    ret  
Dec32ToStr endp
```

При преобразовании чисел для систем счисления, кратных степени двойки, т. е.  $2^k$ , деление может быть заменено сдвигом вправо на  $k$  разрядов. В случае, когда требуется получить текстовое представление числа в системе счисления  $2^k$  с префиксными нулями, можно обойтись вообще без деления и числовой интерпретации исходного бинарного числа, используя известный алгоритм преобразования двоичного в  $2^k$ -ичное через последовательную перекодировку  $k$ -разрядных битовых полей исходного числа в отдельные цифры целевого числа.

Например, для исходного бинарного 0000000110011110 в целевой восьмеричной системе процесс должен охватывать по 3 бита с началом разбиения на битовые поля от младших разрядов: “000636”, что соответствует порядку получения цифр при используемом выше алгоритме получения остатков от деления на  $q$ .

При шестнадцатеричной целевой системе счисления мы получим строку “019E” как результат отдельного преобразования полубайтов исходного бинарного числа. Этот процесс может идти от старших разрядов к младшим, т. к. каждый байт исходного бинарного числа дает ровно две цифры шестнадцатеричного числа. Для того чтобы получить четыре старших разряда числа, необходимо применить сдвиг влево на четыре разряда в таком операнде двойной длины, у которого исходное число находится в младшей половине. Если исходное в байте AL, то сдвиг AX на четыре разряда влево дает старший полубайт в AH. Если исходное в AX, то старший полубайт получается сдвигом влево регистра EAX. Если же исходное имеет длину в 32 разряда и располагается в EAX, то при выполнении команды `shld edx, eax, 4` старшие 4 разряда eax будут выдвинуты в младшие разряды DL, однако при этом само содержимое EAX останется неизменным.

Рассмотрим преобразование для шестнадцатеричных на примере.

### Пример 5.5. Разработка процедуры преобразования 32-разрядного числа в шестнадцатеричное текстовое представление

Пусть требуется разработать процедуру, которая преобразует число в регистре EAX в строку, адресуемую регистром EDI. Процедура имеет вид:

```
; -----  
; Hex32ToStr – преобразование 32-разрядного слова EAX в десятичное текстовое  
; по адресу EDI. В EAX возвращается разрядность числа. Портит EDX.  
; -----  
Hex32ToStr PROC  
    push    ecx  
    mov     ecx, 8  
cycle: xor     edx, edx  
    shld    edx, eax, 4 ; выдвигаем старший полубайта в DL  
    shl     eax, 4      ; поддвигаем следующие полубайты  
    add     dl, '0'      ; превращаем цифру в ее символ  
    cmp     dl, '9'      ; ЕСЛИ цифра больше '9'  
    jle     Ldec  
    add     dl, 'A'-'9'-1 ; ТО коррекция для получения 'A'..'F'  
Ldec:  mov     [edi], dl  
    inc     edi  
    loop    cycle  
    mov     [edi], byte ptr 0  
    pop     ecx  
    ret  
Hex32ToStr endp
```

## 5.4. Вопросы и упражнения

1) Почему в подпрограмме примера 5.5 вслед за 64-разрядным сдвигом EDX\_EAX (команда shld) приходится выполнять 32-разрядный разрядный сдвиг EAX?

2) Разработать подпрограмму QGets, которая получает через регистр ESI адрес строки запроса, содержащей символы кириллицы, через регистр EDI – адрес области памяти, куда должна быть введена строка ответа, не содержащего символы кириллицы, а через регистр ECX – предельно допустимую строку ввода. Подпрограмма должна

выводить строку запроса и вводить строку ответа. Через регистр EAX подпрограмма должна возвращать количество реально введенных символов ответа.

3) Разработать подпрограмму QGetsRu, которая отличается от подпрограммы QGets предыдущего упражнения тем, что в строке ответа могут быть символы кириллицы.

4) Упростить программу примера 5.1 за счет применения подпрограммы QGetsRu.

5) Разработать программу десятичного консольного калькулятора, использующего подпрограммы QGets, StrToDec32 и Dec32ToStr. Программа должна циклически выполнять следующие действия:

- вводить по запросу два операнда;
- преобразовывать операнды во внутреннее представление;
- вводить по запросу знак операции “+” или “-”;
- выполнять заданную операцию;
- преобразовывать результат во внешнее текстовое представление;
- выводить результат;
- выводить запрос на продолжение или выход из программы.

6) Разработать программу шестнадцатеричного калькулятора по аналогии с десятичным калькулятором предыдущего примера.

7) Разработать подпрограммы Oct32ToStr и StrToOct32, обеспечивающие преобразование между внутренним и внешним представлениями восьмеричных чисел. Операнды передаются в подпрограммы, а результаты возвращаются из подпрограмм аналогично подпрограммам Dec32ToStr и StrToDec32 примеров 5.2 и 5.4.

8) Разработать подпрограммы Bin8ToStr и StrToBin8, обеспечивающие преобразование между внутренним однобайтным и внешним строчным представлениями восьмеричных чисел. Операнды передаются в подпрограммы, а результаты возвращаются из подпрограмм аналогично подпрограммам Hex32ToStr и StrToHex32 примеров 5.3 и 5.5.

9) Разработать бинарный консольный калькулятор на основе подпрограмм Bin8ToStr и StrToBin8 аналогично упражнению 4.



## 6. Обработка массивов и адресная арифметика

### 6.1. Объявление массивов и последовательный доступ к их элементам через адресные переменные

В подразделе 2.4, посвященном декларации данных, показано использование оператора DUP, с помощью которого легко объявляются массивы с элементами, которые представляются базовыми типами данных byte, word, dword, qword. Иногда приходится объявлять массивы с элементами других размеров. В этом случае обычно используют кратность размеров таких элементов размерам типовых. Например, массивы элементов размером в 24 байта можно объявить следующими способами:

```
Ar24b db 24 dup(0)
Ar24w dw 12 dup(0)
Ar24d dd 6 dup(0)
Ar24q dq 3 dup(0)
```

Различают последовательный и произвольный доступ к элементам массивов. Последовательному доступу соответствует использование сканера массива, т. е. адресной переменной. В языке Си последовательный доступ осуществляется либо через конструкции (\*p++), (\*++p), (\*p--), (\*--p), либо различными сочетаниям просто обращения к элементу через указатель (\*p) и такой модификации значения p, которая продвигает сканер к соседнему элементу. Произвольный доступ заключается в использовании индекса подобно конструкции p[j], хотя часто значения индекса j в теле цикла обработки массива меняются на единицу, что по сути последовательности обработки совпадает с последовательным доступом.

Переход к обработке соседнего элемента массива при использовании адресной переменной в качестве сканера всегда требует либо увеличения, либо уменьшения значения этой переменной на размер элемента массива. Это одно из базовых правил адресной арифметики.

### Пример 6.1. Фрагменты программ формирования сумм элементов массива

Пусть требуется построить реализации функций подсчета сумм элементов одномерного массива `ar20` с размерностью 20 при различных целочисленных типах элементов. На языке Си это достигается простым циклом:

```
sum = 0; p = ar40; cnt = 20;
do sum += *p++; while (--cnt != 0);
```

Фрагмент цикла подсчета суммы элементов массива байтов имеет вид:

```
    ; элементы имеют тип byte
xor   al,    al      ; sum = 0
lea   esi,   ar20    ; p = ar20
mov   ecx,   20      ; cnt = 20
L:    add    al,    [esi] ; sum += *p
      inc    esi      ; p++
      loop   L        ; while (--cnt != 0)
```

В случае массива слов вместо `AL` должен использоваться `AX`, а для реализации `p++` – подсуммирование `add esi, 2`. Если элементы массива имеют размер 4 байта, то придется использовать `EAX` и `add esi, 4`.

### 6.2. Произвольный доступ к элементам массивов

Произвольный доступ базируется на вычислении адресов на основе значений индексов, базового адреса и минимальных значений индексов.

Для одномерного массива `AR` с базовым адресом `BAR`, минимальным значением индекса `MinInd` и размером элемента `SE` справедливо:

Адрес  $AR[ind] = BAR + (ind - MinInd) * SE$ ,

а при `MinInd = 0`:

Адрес  $AR[ind] = BAR + ind * SE$ .

Для двумерного массива с номером строки `row`, числом элементов в строке `LenRow`, номером элемента в строке `col` при размещении строки в подряд расположенных элементах памяти при нулевых минимальных значениях индексов получаем:

Адрес  $AR[row, col] = BAR + (row * LenRow + col) * SE$ .

Для многомерного массива с индексами  $j_1, j_2, \dots, j_K$ , нулевыми значениями минимальных элементов индексов и количествами самих значений, выражаемых  $L_1, L_2, \dots$ , получаем:

$$\text{Адрес } AR[j_1, j_2, \dots] = \text{BAR} + (j_1 * L_2 + j_2) * L_3 + \dots + j_K * SE.$$

Адресная арифметика, т. е. механизмы вычисления адресов элементов массивов, поддерживается на уровне машинных команд через адресные выражения. Масштабный коэффициент *Scale*, который для IA-32 может принимать значения из множества  $\{1, 2, 4, 8\}$ , позволяет встроить умножение на *SE* непосредственно в метод адресации для всех случаев, когда *SE* принадлежит данному множеству. Использование констант в адресном выражении позволяет задавать базовый адрес *BAR* или смещение к конкретным элементам массива. Фигурирование базовых и индексных регистров в сумме адресного выражения дает еще больше возможностей, особенно в случаях использования многомерных массивов и динамических данных.

### **Пример 6.2. Обращение к элементам одномерных массивов**

Рассмотрим несколько примеров обращения к элементам массивов, объявленным на языке Си как следующие статические данные:

```
char ar8[20], *p8=ar8;
short ar16[20], *p16=ar16;
long ar32[20], *p32=ar20;
unsigned long ind = 7;
```

```
.data
ar8:   db    40 dup(0)
p8:    dd    ar8
ar16:  dw    40 dup(0)
p16:   dd    ar16
ar32:  dd    40 dup(0)
p8:    dd    ar8
ind:   dd    24

. code
mov     byte ptr [ar8+5], 3      ; ar8[5] = 3
mov     word ptr [ar16+6], 4     ; ar16[3] = 4
mov     dword ptr [ar32+20], 9   ; ar32[5] = 9
```

```

mov     edi, p8                ; edi = p8
mov     esi, p16               ; esi = p16
mov     ebx, p32               ; ebx = p32
mov     edx, ind               ; edx = 24
mov     [edi], 10              ; *p8 = 10
mov     [edi+3], 8             ; p8[3] = 8
mov     [esi+8], 16            ; p16[8] = 16
mov     [ebx+16], 11           ; p32[4] = 11
mov     ar8[edx], 10           ; ar8[ind] = 10
mov     ar16[edx], 5           ; ar16[ind/2] = 5
mov     [esi + edx], 6         ; p16[ind/2] = 6
mov     [esi + 2*edx], 2       ; p16[ind] = 2
mov     ar32[edx], 12          ; ar32[ind/4] = 12
mov     [ebx + edx], 8         ; p32[ind/4] = 8
mov     [ebx + 4*edx], 3       ; p32[ind] = 3

```

**Пример 6.3. Обращение к элементам одномерных массивов,  
размещаемых в стеке в качестве локальных данных**

Пусть те же самые данные, что имели место в предыдущем примере, на языке Си объявлены внутри процедуры MyFunc и все размещены в стеке. Поскольку общий размер области этих локальных данных равен 296 байт, доступ к локальным данным может быть организован следующим образом (регистры не сохраняются для простоты восприятия смещений к данным стека):

```

.code
AR8     = -296 ; смещение от еbp до начала массива ar8
P8      = -256 ; смещение от еbp до указателя p8
AR16    = -252
P16     = -172
AR32    = -168
P32     = -8
IND     = -4
MyFunc proc
    push ebp
    mov  ebp, esp
    sub  esp, 296
    lea  edi, AR8[ebp] ; edi = ar8
    mov  P8[ebp], edi  ; p8 = ar8
    inc  P8[ebp]        ; p8++

```

```

lea  edi,   AR16[ebp]   ; edi = ar16
mov  P16[ebp], edi      ; p16 = ar16
sum  P16[ebp], 10       ; p16
lea  edi,   AR32[ebp]   ; edi = ar32
mov  P32,   edi         ; p32 = ar32
mov  AR8[ebp+3], 1      ; ar8[3] = 1
mov  AR16[ebp+8], 2     ; ar16[4] = 2
mov  AR32[ebp+16], 3    ; ar32[4] = 3
mov  eax, P8[EBP]       ; eax = p8
mov  byte ptr [eax+3], 7 ; p8[3] = 7
mov  edx, P32[EBP]      ; eax = p32
mov  dword ptr [edx+12], 0 ; *p32[3] = 0
mov  eax, IND[EBP]      ; eax = p32
mov  dword ptr AR32[4*eax], 10 ; ar32[ind] = 10
mov  dword ptr [4*eax+edx], 15 ; p32[ind] = 15

```

....

MyFunc endp

### 6.3. Организация обработки текстов

Текст представляет собой совокупность строк, каждая из которых является массивом символов. Основная проблема представления строк заключается в том, что разные строки обычно имеют разную длину, а сами строки в ходе обработке эту длину активно изменяют.

Чаще всего используют три способа представления строк:

- а) массив символов с нулевым байтом в качестве завершителя ('`\0`' в языке Си);
- б) массив символов с указателем длины в первом байте или слове (тип `string` в Паскале с одним байтом вначале для указания длины);
- в) массив символов определенной длины с заполнением неиспользуемых символов пробелами.

Представление текстов на основе первого из этих способов подобно тому, что имело место в примере 1.9, т. е. для каждой строки создается указатель на первый символ. Наличие массива указателей существенно

упрощают реализацию таких операций, как удаление и вставка строк, лексикографическое упорядочивание строк.

Представление текста на основе второго и третьего способов предполагает создание двумерного массива, в каждой строке которого храниться одна строка текста. В рамках такой организации текстов облегчается обработка колонок и прямоугольников текста, однако усложняется вставка и замена внутренних частей текста, поскольку приходится копировать довольно большие фрагменты памяти.

#### **6.4. Вопросы и упражнения**

1) Пусть на языке программирования имеется указатель `long *p`, который реализован через переменную `P` типа `DWORD` ассемблер-программы. Какое значение должно быть подсуммировано к переменной `P` для реализации СИ-оператора `p += 7`?

2) Каким образом задается размер элементов массивов, копируемых оператором `REP MOVS`?

3) Разработать подпрограмму нахождения минимального элемента в массиве 32-разрядных знакопеременных чисел. В подпрограмму через стек передаются адрес массива и его длина. Результат возвращается через `EAX`.

4) Разработать подпрограмму `PSum`, которая получает через `ESI` адрес массива 32-разрядных чисел, через `ECX` – длину этого массива, через `EDX` и `EDI` – диапазон значений `[min..max]`. Подпрограмма должна возвращать через `EAX` сумму всех элементов, значения которых входят в заданный диапазон.

5) Разработать подпрограмму `SOR`, которая получает через `ESI` и `EDI` адреса записей, в каждой из которых первое 32-разрядное слово задает адрес исходного массива 32-разрядных чисел, а второе 32-разрядное слово – число элементов этого массива. Через `EBX` в подпрограмму передается адрес результирующего массива. Подпрограмма должна в качестве результирующего массива сформировать результат

объединения множеств чисел, заданных в двух исходных массивах. Через EAX должна возвращаться длина полученного массива.

6) Разработать подпрограмму SAND, которая отличается от подпрограммы SOR предыдущего примера только тем, что формирует пересечение множеств, а не объединение.

7) Разработать подпрограмму StrCmp сравнения двух строк, заданных двумя указателями, передаваемыми через стек. Подпрограмма должна возвращать через EAX одно из следующих значений: 0 – строки равны, 1 – первая строка в лексикографическом порядке должна следовать за второй, -1 – вторая строка в лексикографическом порядке должна следовать за первой.

8) Разработать подпрограмму SearchText поиска строки в тексте, которая получает через стек два указателя: указатель массив адресов строк и указатель искомой строки. Все строки завершаются байтом 0. Через EAX подпрограмма должна возвращать либо 0, если в тексте нет искомой строки, либо адрес найденной строки.

9) Разработать подпрограмму PSim, которая проверяет симметричность матрицы 32-разрядных чисел. В подпрограмму через ESI передается базовый адрес массива, через ECX – число строк, через EDX – число столбцов. Через EAX возвращается 0, если матрица несимметрична, либо 1, если симметрична.

10) Разработать подпрограмму удаления всех повторяющихся элементов массива 32-разрядных чисел, адрес и размер которого передается в подпрограмму через стек. Через EAX должен быть возвращен размер результирующего массива, в котором каждое число фигурирует только один раз.

## Библиографический список

1. Ирвин, К. Язык ассемблера для процессоров Intel / К. Ирвин. – М. : Изд. дом «Вильямс», 2005.
2. Пахомов, М. М. Программирование на ассемблере MASM32. Изучение среды разработки RADasm и отладчика OllyDbg : методические указания к лабораторной работе / М. М. Пахомов. – М. : МАТИ, 2012.
3. Зубков, С. В. Assembler для DOS, Windows и Unix / С. В. Зубков. – М. : ДМК Пресс, 2004.
4. Пирогов, В. Ю. Ассемблер : учебный курс / В. Ю. Пирогов. – СПб. : БХВ-Петербург, 2003.
5. Финогенов, К. Г. Язык Ассемблера: уроки программирования / К. Г. Финогенов, П. И. Рудаков. – М. : Диалог-МИФИ, 2001.
6. Юров, В. И. Справочник по языку Ассемблера IBM PC / В. И. Юров. – СПб. : Питер, 2004.



Учебное электронное издание

НЕГОДА Виктор Николаевич

МАШИННО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Учебное пособие

Редактор Н. А. Евдокимова

ЭИ № 805. Объем данных 2,71 Мб. Заказ ЭИ № 1128.

Печатное издание

Подписано в печать 01.12.2015. Формат 60×84/16.

Усл. печ. л. 9,53. Тираж 50 экз.

Ульяновский государственный технический университет  
432027, Ульяновск, Северный Венец, 32.  
ИПК «Венец» УлГТУ, 432027, Ульяновск, Северный Венец, 32.  
Тел.: (8422) 778-113  
E-mail: [venec@ulstu.ru](mailto:venec@ulstu.ru)  
<http://www.venec.ulstu.ru>