

Рудольф Марек

АССЕМБЛЕР

НА ПРИМЕРАХ Базовый курс



Большое количество наглядных примеров.
Освой язык ассемблера быстро и легко!

НиТ
издательство

Rudolf Marek

Učíme se programovat v jazyce Assembler pro PC

Computer Press
Brno

Рудольф Марек

АССЕМБЛЕР

на примерах

Базовый курс



Наука и Техника, Санкт-Петербург
2005

Рудольф Марек.

Ассемблер на примерах. Базовый курс. —

СПб: Наука и Техника, 2005. — 240 с.: ил.

ISBN 5-94387-232-9

Серия «Просто о сложном»

Эта книга представляет собой великолепное практическое руководство по основам программирования на языке ассемблера. Изложение сопровождается большим количеством подробно откомментированных примеров, что способствует наилучшему пониманию и усвоению материала. Доходчиво объясняются все основные вопросы программирования на этом языке.

Вы узнаете, как писать ассемблерные программы под разные операционные системы (Windows, DOS, Linux), как создавать резидентные программы, как писать ассемблерные вставки в программы на языках высокого уровня и многое другое. Попутно вам будут разъяснены основные моменты работы процессора, операционных систем, управления памятью и взаимодействия программ с аппаратными устройствами ПК – то есть все то, без знания чего нельзя обойтись при программировании на языке низкого уровня, которым и является ассемблер.

Книга написана доступным языком. Лучший выбор для начинающих.

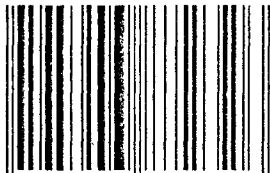
Русское издание под редакцией Финкова М.В. и Березкиной О.И.

Copyright © Computer Press 2004 Učime se programovat
v jazyce Assembler pro PC by Rudolf Marek, ISBN: 80-722-6843-0.
All rights reserved

Контактные телефоны издательства
(812) 567-70-25, 567-70-26
(044) 516-38-66

Официальный сайт www.nit.com.ru

- © Перевод на русский язык,
Наука и Техника, 2005
- © Издание на русском языке,
оформление, Наука и Техника, 2005



9 795943 872326

ISBN 5-94387-232-9

ООО «Наука и Техника».

Лицензия №000350 от 23 декабря 1999 года.

198097, г. Санкт-Петербург, ул. Маршала Говорова, д. 29.

Подписано в печать 08.08.05. Формат 70×100 1/16.

Бумага газетная. Печать офсетная. Объем 15 п. л.

Тираж 5000 экз. Заказ № 293

Отпечатано с готовых диапозитивов в ОАО «Техническая книга»
190005, Санкт-Петербург, Измайловский пр., 29

Содержание

Введение	10
Глава 1. Базовые системы счисления и термины	11
1.1. Системы счисления и преобразования между ними	12
1.2. Типы данных. Их представление в компьютере	15
Глава 2. Введение в семейство процессоров x86	19
2.1. О компьютерах.....	20
2.2. История процессоров x86	22
2.3. Процессоры и их регистры: общая информация.....	23
2.4. Процессор 80386.....	25
Регистры общего назначения.....	25
Индексные регистры	27
Сегментные регистры	27
Регистры состояния и управления.....	27
2.5. Прерывания	28
Глава 3. Анатомия команд и как они выполняются процессором	30
3.1. Как команды выполняются процессором.....	31
3.2. Операнды	33
3.3. Адресация памяти.....	34
3.4. Команды языка ассемблера	35
Глава 4. Основные команды языка ассемблера	36
4.1. Команда MOV.....	37
4.2. «Остроконечники» и «тупоконечники»	39
4.3. Арифметические команды.....	40
4.3.1. Инструкции сложения ADD и вычитания SUB	41
4.3.2. Команды инкрементирования INC и декрементирования DEC	43
4.3.3. Отрицательные числа — целые числа со знаком	44
4.3.4. Команды для работы с отрицательными числами	46
Команда NEG	46
Команда CBW.....	46
Команда CWD	47
Команда CDQ.....	47
Команда CWDE	47
4.3.5. Целочисленное умножение и деление	48
Команды MUL и IMUL	48
Команды DIV и IDIV	50
4.4. Логические команды.....	51
Команда AND	51
Команда OR	52
Команда XOR	52
Команда NOT	53
Массивы битов (разрядные матрицы).....	53

Глава 5. Управляющие конструкции 55

5.1. Последовательное выполнение команд	56
5.2. Конструкция «IF THEN» — выбор пути	57
5.2.1. Команды CMP и TEST	57
5.2.2. Команда безусловного перехода — JMP	58
5.2.3. Условные переходы — Jx	59
5.3. Итерационные конструкции — циклы	63
Цикл со счетчиком с помощью конструкций IF и GOTO	63
LOOP — сложная команда, простая запись цикла	65
Цикл со счетчиком и дополнительным условием.	
Команды LOOPZ и LOOPNZ	66
5.4. Команды обработки стека	67
Что такое стек и как он работает?	67
Команды PUSH и POP: толкнуть и вытолкнуть	68
Команды PUSHA/POPA и PUSHAD/POPAD:	
«толкаем» все регистры общего назначения	70
Команды PUSHF/POPF и PUSHFD/POPDF:	
«толкаем» регистр признаков	71
Команды CALL и RET: организуем подпрограмму	71
Команды INT и IRET: вызываем прерывание	73

Глава 6. Прочие команды 76

6.1. Изменение регистра признаков напрямую	77
Команды CLI и STI	77
Команды STD и CLD	77
6.2. Команда XCHG — меняем местами операнды	78
6.3. Команда LEA — не только вычисление адреса	78
6.4. Команды для работы со строками	79
Команды STOSx — запись строки в память	79
Команды LODSx — чтение строки из памяти	80
Команды CMPSx — сравнение строк	80
Команды SCASx — поиск символа в строке	80
Команды REP и REPZ — повторение следующей команды	80
6.5. Команды ввода/вывода (I/O)	84
Команды IN и OUT — обмен данными с периферией	84
Организация задержки. Команда NOP	86
6.6. Сдвиг и ротация	86
Команды SHR и SHL — сдвиг беззнаковых чисел	87
Команды SAL и SAR — сдвиг чисел со знаком	89
Команды RCR и RCL — ротация через флаг переноса	89
Команды ROR и ROL — ротация с выносом во флаг переноса	90
6.7. Псевдокоманды	90
Псевдокоманды DB, DW и DD — определение констант	90
Псевдокоманды RESB, RESW и RESD — объявление переменных ..	91
Псевдокоманда TIMES — повторение следующей псевдокоманды ..	91
Псевдокоманда INCBIN — подключение двоичного файла	92
Псевдокоманда EQU — вычисление константных выражений	92
Оператор SEG — смена сегмента	93
6.8. Советы по использованию команд	93
Директива ALIGN — выравнивание данных в памяти	93
Загрузка значения в регистр	94
Оптимизируем арифметику	94
Операции сравнения	95
Разное	96

Глава 7. Полезные фрагменты кода	97
7.1. Простые примеры	98
Сложение двух переменных	98
Сложение двух элементов массива	99
Суммируем элементы массива	99
Чет и нечет	100
Перестановка битов в числе	101
Проверка делимости числа нацело	101
7.2. Преобразование числа в строку	102
7.3. Преобразование строки в число	107
 Глава 8. Операционная система	 111
8.1. Эволюция операционных систем	112
8.2. Распределение процессорного времени. Процессы	113
Процессы	113
Планирование процессов	114
Состояния процессов	114
Стратегия планирования процессов	115
8.3. Управление памятью	116
Простое распределение памяти	116
Свопинг (swapping) — организация подкачки	117
Виртуальная память и страничный обмен	117
8.4. Файловые системы	120
Структура файловой системы	120
Доступ к файлу	121
Физическая структура диска	122
Логические диски	123
8.5. Загрузка системы	123
Этапы загрузки	123
Немного о том, что такое BIOS	124
 Глава 9. Компилятор NASM	 125
9.1. Предложения языка ассемблера	126
9.2. Выражения	126
9.3. Локальные метки	127
9.4. Препроцессор NASM	128
Однострочные макросы — %define, %undef	128
Сложные макросы — %macro %endmacro	129
Объявление макроса — %assign	130
Условная компиляция — %if	130
Определен ли макрос? Директивы %ifndef, %ifndef	131
Вставка файла — %include	131
9.5. Директивы Ассемблера	131
Директива BITS — указание режима процессора	132
Директивы SECTION и SEGMENT — задание структуры программы	132
Директивы EXTERN, GLOBAL и COMMON — обмен данными с другими программными модулями	134
Директива CPU — компиляция программы для выполнения на определенном процессоре	134
Директива ORG — указание адреса загрузки	134
9.6. Формат выходного файла	135
Создание выходного файла: компиляция и компоновка	135

Формат bin — готовый исполняемый файл.	136
Формат OMF — объектный файл для 16-битного режима.	136
Формат win32 — объектный файл для 32-битного режима.	137
Форматы aout и aoutb — старейший формат для UNIX.	137
Формат coff — наследник a.out.	138
Формат elf — основной формат в мире UNIX.	138
Символическая информация.	138

Глава 10. Программирование в DOS. 139

10.1. Адресация памяти в реальном режиме.	140
10.2. Организация памяти в DOS.	142
10.3. Расширение памяти свыше 1 MB.	143
10.4. Типы исполняемых файлов в DOS.	144
10.5. Основные системные вызовы.	146
Немедленное завершение программы.	146
Вывод строки. Пишем программу «Hello, World!».	147
Ввод с клавиатуры.	148
10.6. Файловые операции ввода-вывода.	153
Открытие файла.	153
Закрытие файла.	154
Чтение из файла.	154
Запись в файл.	155
Открытие/создание файла.	158
Поиск позиции в файле (SEEK).	160
Другие функции для работы с файлами.	161
Длинные имена файлов и работа с ними.	162
10.7. Работа с каталогами.	163
Создание и удаление каталога (MKDIR, RMDIR).	163
Смена текущего каталога (CHDIR).	163
Получение текущего каталога (GETCWD).	163
10.8. Управление памятью.	165
Изменение размера блока памяти.	165
Выделение памяти.	166
Освобождение памяти.	166
10.9. Аргументы командной строки.	166
10.10. Коды ошибок.	167
10.11. Отладка.	168
10.11.1. Что такое отладка программы и зачем она нужна.	168
10.11.2. Отладчик grdb.exe.	169
Методика использования.	169
Основные команды отладчика grdb.	172
Пример разработки и отладки программы.	172
10.12. Резидентные программы.	180
10.13. Свободные источники информации.	185

Глава 11. Программирование в Windows. 186

11.1. Введение.	187
11.2. «Родные» Windows-приложения.	187
11.2.1. Системные вызовы API.	187
11.2.2. Программа «Hello, World!» с кнопкой под Windows.	188
11.3. Программная совместимость.	190
11.4. Запуск DOS-приложений под Windows.	190
11.5. Свободные источники информации.	190

Глава 12. Программирование в Linux.	191
12.1. Введение	192
12.2. Структура памяти процесса	193
12.3. Передача параметров командной строки и переменных окружения	194
12.4. Вызов операционной системы	194
12.5. Коды ошибок	195
12.6. Map-страницы	195
12.7. Программа «Hello, World!» под Linux.	197
12.8. Облегчим себе работу: утилиты Asmutils	199
12.9. Макросы Asmutils	200
12.10. Операции файлового ввода/вывода (I/O)	201
Открытие файла	201
Закрытие файла	202
Чтение из файла	202
Запись в файл	203
Поиск позиции в файле	206
Другие функции для работы с файлами	207
12.11. Работа с каталогами	209
Создание и удаление каталога (MKDIR, RMDIR)	209
Смена текущего каталога (CHDIR)	210
Определение текущего каталога (GETCWD)	210
12.12. Ввод с клавиатуры. Изменение поведения потока стандартного ввода.	
Системный вызов IOCTL	210
12.13. Распределение памяти	211
12.14. Отладка. Отладчик ALD.	212
12.15. Ассемблер GAS	215
12.16. Свободные источники информации.	216
12.17. Ключи командной строки компилятора	216
 Глава 13. Компоновка — стыковка ассемблерных программ с программами, написанными на языках высокого уровня.	 217
13.1. Передача аргументов	218
13.2. Что такое стек-фрейм?	219
13.2.1. Стек-фрейм в языке C (32-битная версия)	220
13.2.2. Стек-фрейм в языке C (16-битная версия)	223
13.3. Компоновка с C-программой	224
13.4. Компоновка с Pascal-программой	226
 Глава 14. Заключение	 229
 Глава 15. Часто используемые команды.	 230

Введение

Эта книга — начальный курс и практическое руководство по программированию на языке ассемблера для процессоров серии x86 — самых распространенных в современных ПК. Она предназначена для студентов и старшеклассников, которые хотят познакомиться с языком программирования низкого уровня, позволяющим писать компактные, быстрые и эффективные программы, взаимодействующие с аппаратным обеспечением компьютера напрямую, минуя любую операционную систему.

Книга содержит подробные объяснения и множество практических примеров.

Последовательное изложение предмета дает читателю ясное понимание того, как язык ассемблера связан с физической архитектурой компьютера, как ассемблер работает с регистрами процессора, как реализовать основные программные конструкции, как скомпилировать и запустить законченную программу, как из ассемблерной программы вызывать операционную систему и обращаться к файловой системе.

Вместе с автором читатель проходит шаг за шагом от решения типовых, не зависящих от платформы, задач к написанию практически полезных программ, работающих в среде DOS, Windows и Linux, а также узнает, как компоновать подпрограммы на языке ассемблера с подпрограммами, написанными на языках высокого уровня.

Книга написана простым, доступным языком, а все примеры программ тщательно прокомментированы.

Особое внимание обращено на следующие вопросы:

- Архитектура процессора, функции операционной системы, машинный код и символическое представление команд и адресов;
- Различные системы счисления и перевод чисел из одной в другую;
- Основные и сложные команды;
- Управляющие конструкции и их реализация;
- Готовые фрагменты кода, выполняющие самые типичные задачи;
- Использование свободно распространяемого компилятора Netwide Assembler (NASM);
- Практическое программирование в среде DOS, Windows и Linux;
- Написание ассемблерных вставок в программы на языках высокого уровня (C и Паскаль).

Автор книги — деятельный разработчик свободного программного обеспечения, соавтор самого маленького в мире веб-сервера размером в 514 байт, участник разработки пакета Asmtutils и создатель программного модуля для Linux-проигрывателя MPlayer.

Глава 1

Базовые системы счисления и термины

Системы счисления
и преобразования между ними

Типы данных. Их представление
в компьютере

Архитектура компьютера тесно связана с двоичной системой счисления, которая состоит всего из двух цифр — 0 и 1. С технической точки зрения, двоичная система (с основой 2) идеально подходит для компьютеров, поскольку обе цифры могут отображать два состояния — включено (1) и выключено (0).

Как мы вскоре увидим, «большие» числа становятся огромными в двоичной системе, поэтому для более удобного представления чисел были разработаны восьмеричная и шестнадцатеричная системы счисления (с основами 8 и 16 соответственно). Обе системы легко преобразуются в двоичную систему, но позволяют записывать числа более компактно.

1.1. Системы счисления и преобразования между ними

Различные системы счисления отличаются не только базовым набором чисел, но и основными концепциями, которые лежат в их основе. Взять, например, систему счисления, которая использовалась древними римлянами: она довольно трудна для восприятия, в ней очень сложно производить вычисления и невозможно представить 0. Данная система неудобна даже для человека, не говоря уж о том, чтобы научить компьютер «понимать» ее.

Десятичная система, которую мы используем всю жизнь, относится к классу так называемых позиционных систем, в которых число A может быть представлено в виде:

$$A = a_n * z^n + a_{n-1} * z^{n-1} + \dots + a_1 * z^1 + a_0 * z^0$$

Здесь a_n — это цифры числа, а Z — основание системы счисления, в нашем случае — 10.

Например, число 1234 можно представить так:

$$1234 = 1 * 10^3 + 2 * 10^2 + 3 * 10^1 + 4 * 10^0$$

«Вес» каждой цифры определяется позицией цифры в числе и равен степени основания, соответствующей ее позиции.

При работе с различными системами счисления мы будем записывать само число в скобках, а за скобками — основание системы. Например, если написать просто число 1100, то не понятно, в какой системе оно записано — это может быть одна тысяча сто, а может быть 12, если число записано в двоичной системе. А если представить число в виде $(1100)_2$, то сразу все становится на свои места: число записано в двоичной системе. Кстати, двоичная система тоже является позиционной, поэтому число 1100 в двоичной системе мы можем представить так:

$$(1100)_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$$

После сложения $8+4$ мы получим, что $(1100)_2$ равно 12. Как видите, все точно так же, как и с десятичной системой. Обратите внимание, что для представления числа 12 в двоичной системе использованы только четыре разряда. Наибольшее число, которое можно записать четырьмя двоичными цифрами, равно 15, потому что $(1111)_2 = 1 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 + 1 \cdot 1 = 15$. Давайте рассмотрим первые 16 чисел:

Десятичное число	Двоичное число	Десятичное число	Двоичное число
0	0	8	1000
1	1	9	1001
2	10	10	1010
3	11	11	1011
4	100	12	1100
5	101	13	1101
6	110	14	1110
7	111	15	1111

Числа растут равномерно, и нетрудно предположить, что 16 будет представлено в двоичной системе как $(10000)_2$.

Восьмеричная система счисления (по основанию 8) состоит из большого количества цифр — из восьми (от 0 до 7). Преобразование из этой системы в десятичную систему полностью аналогично преобразованию из двоичной системы, например:

$$(77)_8 = 7 \cdot 8^1 + 7 \cdot 8^0 = 63$$

Восьмеричная система счисления использовалась в очень популярных ранее 8-битных компьютерах ATARI, ZX Spectrum и др. Позже она была заменена шестнадцатеричной системой, которая также будет рассмотрена в этой книге.

В шестнадцатеричной системе цифрами представлены только первые 10 чисел, а для представления остальных 5 чисел используются символы A-F:

$$A = 10, B = 11, C = 12, D = 13, E = 14, F = 15$$

Представим, как изменяется наш возраст в шестнадцатеричной системе: вы получили свой паспорт в 10 лет и стали совершеннолетним в 12 лет.

Для шестнадцатеричной системы сохраняются те же принципы преобразования:

$$\begin{aligned}\text{Число } (524D)_{16} &= 5 \cdot 16^3 + 2 \cdot 16^2 + 4 \cdot 16^1 + 13 \cdot 16^0 = \\ &= 20\,480 + 512 + 64 + 13 = 21\,069\end{aligned}$$

$$\text{Число } (DEAD)_{16} = 13 \cdot 16^3 + 14 \cdot 16^2 + 10 \cdot 16^1 + 13 \cdot 16^0 = 57\,005$$

$$\begin{aligned}\text{Число } (DEADBEEF)_{16} &= 13 \cdot 16^7 + 14 \cdot 16^6 + 10 \cdot 16^5 + 13 \cdot 16^4 + \\ &+ 11 \cdot 16^3 + 14 \cdot 16^2 + 14 \cdot 16^1 + 15 \cdot 16^0 = 3\,735\,928\,559\end{aligned}$$

$$\text{Число } (C001)_{16} = 12 \cdot 16^3 + 0 \cdot 16^2 + 0 \cdot 16^1 + 1 = 49\,153$$

Итак, мы научились преобразовывать любое число, представленное в двоичной, восьмеричной и шестнадцатеричной системах, в десятичную систему. А теперь займемся обратным преобразованием — из десятичной системы в систему с основанием n . Для обратного преобразования мы должны делить наше число на n и записывать остатки от деления до тех пор, пока частное от предыдущего деления не станет равно 0. Например, преобразуем 14 в двоичную систему:

$$14/2 = 7 \quad \text{остаток } 0$$

$$7/2 = 3 \quad \text{остаток } 1$$

$$3/2 = 1 \quad \text{остаток } 1$$

$$1/2 = 0 \quad \text{остаток } 1$$

Мы завершили процесс деления, когда последнее частное стало равно 0. Теперь запишем все остатки подряд от последнего к первому, и мы получим число в двоичной системе — $(1110)_2$.

Рассмотрим еще один пример — преобразование числа 13 в двоичную систему:

$$13/2 = 6 \quad \text{остаток } 1$$

$$6/2 = 3 \quad \text{остаток } 0$$

$$3/2 = 1 \quad \text{остаток } 1$$

$$1/2 = 0 \quad \text{остаток } 1$$

Как и в прошлом случае, мы делили до тех пор, пока частное не стало равно 0. Если записать остатки снизу вверх, мы получим двоичное число $(1101)_2$.

А теперь потренируемся с шестнадцатеричной системой — преобразуем число 123456 в эту систему:

$$123456/16 = 7716 \quad \text{остаток } 0$$

$$7716/16 = 482 \quad \text{остаток } 4$$

$482/16 = 30$	остаток 2
$30/16 = 1$	остаток 14 = E
$1/16 = 0$	остаток 1

После записи всех остатков получим, что число $123\,456 = (1E240)_{16}$.

Запись со скобками и нижним индексом в тексте программы неудобна, поэтому мы будем использовать следующие обозначения для записи чисел в различных системах счисления:

- Запись шестнадцатеричного числа начинается с 0x или \$0 либо заканчивается символом «h». Если первая цифра шестнадцатеричного числа — символ A-F, то перед таким числом нужно обязательно написать 0, чтобы компилятор понял, что перед ним число, а не идентификатор, например, 0DEADh.

Таким образом, записи 0x1234, \$01234 и 01234h представляют число $(1234)_{16}$.

- Десятичные числа могут записываться без изменений либо они заканчиваться постфиксом «d». Например, 1234 и 1234d представляют число $(1234)_{10}$.
- Двоичные цифры должны заканчиваться постфиксом «b», например, 1100b — это $(1100)_2$.
- Восьмеричные цифры заканчиваются на «q»: 12q — это $(12)_8$.

Далее в этой книге шестнадцатеричные числа мы будем записывать в виде «0x...», двоичные — «...b», а десятичные — без изменений. В вашем собственном коде основание системы счисления (постфикс «d» или «h») лучше указывать явно, потому что одни ассемблеры рассматривают число без приставок как десятичное, а другие — как шестнадцатеричное.

1.2. Типы данных. Их представление в компьютере

Основной и неделимой единицей данных является бит. Слово «bit» — это сокращение от «binary digit» — двоичная цифра. **Бит может принимать два значения — 0 и 1 — ложь или истина, выключено или включено.** На логике двух состояний основаны все логические цепи компьютеров, поэтому поговорим о бите более подробно.

Двоичное число содержит столько битов, сколько двоичных цифр в его записи, поэтому диапазон допустимых значений выводится из количества рядов (цифр), отведенных для числа. Возьмем положительное целое двоичное число, состоящее из четырех битов: оно может выражать 2^4 или шестнадцать различных значений.

Биты (разряды) двоичного числа нумеруются справа налево, от наименее значимого до наиболее значимого. Нумерация начинается с 0. *Самый правый бит числа — это бит с номером 0 (первый бит).* Этот бит называется LSB-битом (*Least Significant Bit — наименее значимый бит*). Подобно этому самый левый бит называется MSB-битом (*Most Significant Bit — наиболее значимый бит*).

Биты могут объединяться в группы, группа из четырех битов называется полубайтом (nibble). Компьютер не работает с отдельными битами, обычно он оперирует группами битов, например, группа из восьми битов образует базовый тип данных, который называется *байтом*. Восемь битов в байте — это не закон природы, а количество, произвольно выбранное разработчиками IBM, создававшими первые компьютеры.

Большие группы битов называются словом (word) или двойным словом (dword — double word). Относительно PC-совместимых компьютеров мы можем сказать следующее:

1 байт = 8 бит

1 слово (word) = 2 байта = 16 бит

1 двойное слово (dword) = 4 байта = 32 бит

Один байт — это наименьшее количество данных, которое может быть прочитано из памяти или записано в нее, поэтому каждый байт памяти имеет индивидуальный адрес. Байт может содержать число в диапазоне 0 — 255 (то есть 2^8 — 256 различных чисел). В большинстве случаев этого значения недостаточно, поэтому используется следующая единица данных — слово. Слово может содержать число в диапазоне 0 — 65 535 (то есть 2^{16} = 65 536 различных значений). Двойное слово имеет диапазон значений 0 — 4 294 967 295 (2^{32} = 4 294 967 296 значений).

Давным-давно, еще во времена первых компьютеров, емкость носителей информации представлялась в байтах. Со временем технологии усовершен-

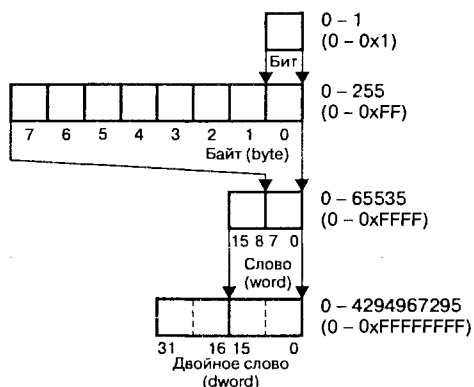


Рис. 1.1. От бита до двойного слова

ставались, цены на память падали, а требования к обработке информации возрастали. Поэтому выражать емкость в байтах стало неудобно.

Решили, что емкость будет выражаться в килобайтах (KB, Kb, Кб или просто k). Но, в отличие от международной системы SI, приставка «кило» означает не 1000, а 1024. Почему именно 1024? Поскольку все в компьютере было завязано на двоичной системе, для простоты любое значимое число должно было выражаться как степень двойки. 1024 — это 2^{10} . Следующие приставки — М (мегабайт, MB, Mb, Мб), G (гигабайт, GB, Гб), Т (терабайт, TB, ТБ) и P (петабайт, PB, Пб) — вычисляются умножением 1024 на предыдущее значение, например, $1 \text{ Кб} = 1024$, значит, $1 \text{ Мб} = 1 \text{ Кб} * 1024 = 1024 * 1024 = 1\,048\,576$ байт. Думаю, с этим все понятно, давайте вернемся к типам данных.

В языках программирования высокого уровня есть специальные типы данных, позволяющие хранить символы и строки. В языке ассемблера таких типов данных нет. Вместо них для **представления одного символа используется байт**, а для **представления строки — группа последовательных байтов**. Тут все просто. Каждое значение байта соответствует одному из символов ASCII-таблицы (American Standard Code for Information Interchange). Первые 128 символов — управляющие символы, латинские буквы, цифры — одинаковы для всех компьютеров и операционных систем. Давайте рассмотрим таблицу кодов ASCII (рис. 1.2).

Шестнадцатеричные цифры в заголовках строк и столбцов таблицы представляют числовые значения отдельных символов. Например, координаты заглавной латинской буквы А — 40 и 01. Сложив эти значения, получим 0x41 (то есть 65 в десятичной системе) — код символа 'А' в ASCII-коде.

Печатаемые символы в ASCII-таблице начинаются с кода 0x20 (или 32d). Символы с кодом ниже 32 представляют так называемые управляющие символы. Наиболее известные из них — это 0xA или LF — перевод строки, и 0xD — CR — возврат каретки.

Важность управляющих символов CR и LF обусловлена тем, что они обозначают конец строки — тот символ, который в языке программирования C обозначается как `\n`. К сожалению, в различных операционных системах он представляется по-разному: например, в Windows (и DOS) он представляется двумя символами (CR, LF — 0xD, 0xA), а в операционной системе UNIX для обозначения конца строки используется всего один символ (LF — 0xA).

Символы с кодами от 128 до 256 и выше стали «жертвами» различных стандартов и кодировок. Обычно они содержат национальные символы, например, у нас это будут символы русского алфавита и, возможно, некоторые символы псевдографики, в Чехии — символы чешского алфавита и т.д. Следует отметить, что для русского языка используются кодировки CP 866 (в DOS) и CP 1251 (Windows).

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00		Ⓐ	Ⓑ	Ⓓ	Ⓔ	Ⓕ	Ⓖ	Ⓢ		⓪						
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
10	▶	◀		!		—		↑	↓	→	←		↔	▲	▼	
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
20		!	π	#	\$	%	&	'	()	*	+	,	-	.	/
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
30	0	1	2	3	4	5	6	7	8	9	:	<	=	>	?	
	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
50	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
70	p	q	r	s	t	u	v	w	x	y	z	{		}	~	□
	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
80	ç	ü	é	â	ä	à	ä	ç	ê	ë	è	ï	î	ì	ä	å
	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
90	é	æ	æ	ô	ö	ö	û	ù	ÿ	ö	ü	ç	æ	æ	æ	f
	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A0	í	ó	ú	ñ	ñ	ª	º	¿	¡	¬	½	¾	¿	¿	¿	¿
	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B0	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣
	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C0	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣
	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D0	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣
	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E0	α	β	Γ	π	Σ	σ	μ	τ	φ	θ	Ω	δ	∞	φ	ε	η
	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F0	≡	±	≥	≤			÷	∞	°	•	•	√	π	ε	■	□
	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Рис. 1.2. Таблица кодов ASCII

Глава 2

Введение в семейство процессоров x86

О компьютерах...
История процессоров x86
Процессоры и их регистры: общая информация
Процессор 80386
Прерывания

Мы начинаем знакомиться с языком ассемблера. Это язык программирования низкого уровня, то есть максимально приближенный к «железу» — аппаратному обеспечению компьютера. Для каждого процессора характерен свой уникальный набор действий, которые процессор способен выполнить, поэтому языки ассемблера разных процессоров отличаются друг от друга. Например, если процессор не умеет выполнять умножение, то в его языке ассемблера не будет отдельной команды «умножить», а перемножать числа программисту придется при помощи нескольких команд сложения.

Собственно говоря, язык ассемблера — это всего лишь ориентированная на человека форма записи инструкций процессора (которые называются также машинным языком), а сам ассемблер — это программа, переводящая символические имена команд в машинные коды.

Вот почему, прежде чем приступить к изучению команд языка ассемблера, нам нужно побольше узнать о процессоре, для которого этот язык предназначен.

2.1. О компьютерах...

Первым популярным компьютером стал компьютер ENIAC (Electronic Numerical Integrator And Calculator), построенный из электронных ламп и предназначенный для решения дифференциальных уравнений. Программирование этого компьютера, которое заключалось в переключении тумблеров, было очень трудоемким процессом.

Следующим компьютером после ENIAC был не столь популярный EDVAC (Electronic Discrete Variable Automatic Computer), построенный в 1946 г. Принципы, заложенные в этом компьютере, используются и по сей день: эта машина, подобно современным компьютерам, хранила заложенную программу в памяти. Концепция компьютера EDVAC, разработанная американским ученым венгерского происхождения Джоном фон Нейманом, основывалась на следующих принципах:

1. Компьютер должен состоять из следующих модулей: управляющий блок (контроллер), арифметический блок, память, блоки ввода/вывода.

2. Строение компьютера не должно зависеть от решаемой задачи (это как раз относится к ENIAC), программа должна храниться в памяти.
3. Инструкции и их операнды (то есть данные) должны также храниться в той же памяти (гарвардская концепция компьютеров, основанная на концепции фон Неймана, предполагала отдельную память для программы и данных).
4. Память делится на ячейки одинакового размера, порядковый номер ячейки считается ее адресом (1 ячейка эквивалентна 1 байту).
5. Программа состоит из серии элементарных инструкций, которые обычно не содержат значения операнда (указывается только его адрес), поэтому программа не зависит от обрабатываемых данных (это уже прототип переменных). Инструкции выполняются одна за другой, в том порядке, в котором они находятся в памяти (к слову, современные микропроцессоры позволяют параллельное выполнение нескольких инструкций).
6. Для изменения порядка выполнения инструкций используются инструкции условного или безусловного (jump) перехода.
7. Инструкции и данные (то есть операнды, результаты или адреса) представляются в виде двоичных сигналов и в двоичной системе счисления.

Оказалось, что концепция фон Неймана настолько мощна и универсальна, что она до сих пор используется в современных компьютерах.

Однако продвижение компьютера в наши дома потребовало долгого времени — почти сорока лет. В 1950-ых годах был изобретен транзистор, который заменил большие, склонные ко всяким сбоям, вакуумные лампы. Со временем размер транзисторов уменьшился, но самым большим их компонентом оставался корпус. Решение было простым: разместить много транзисторов в одном корпусе. Так появились интегральные микросхемы (чипы). Компьютеры, построенные

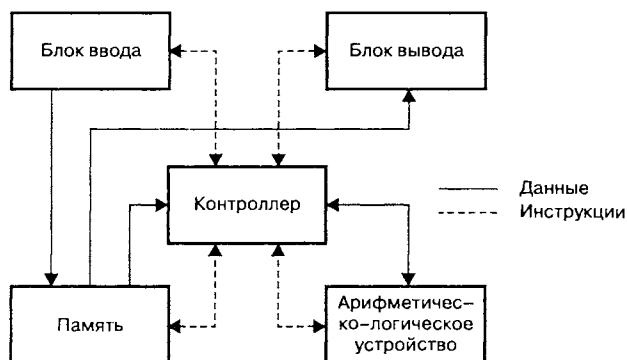


Рис. 2.1. Концепция фон Неймана

на интегральных микросхемах, стали намного меньше в размере, однако они все еще занимали целые комнаты. К тому же эти компьютеры были очень чувствительны к окружающей среде: без кондиционера они не работали.

В конце 1970-ых интегральные микросхемы упали в цене до такой степени, что стали доступны рядовым потребителям. Почему же люди в то время не покупали персональные компьютеры? Потому что их не было в продаже! Люди покупали микросхемы и собирали простые восьмиразрядные компьютеры в порядке хобби — так, в знаменитом гараже, началась история фирмы Apple. За Apple последовали другие компании, начавшие производство восьмиразрядных компьютеров, которые можно было подключить к обычному телевизору и играть или заняться их программированием.

В 1981 году лидером рынка универсальных ЭВМ — компанией IBM — был выпущен персональный компьютер IBM PC XT. Это был полноценный персональный компьютер — с монитором, клавиатурой и системным блоком. Компьютер IBM PC XT был оборудован 8-разрядным микропроцессором Intel 8088. Эта модель стала началом огромной серии персональных компьютеров (PC, Personal Computer), которые производились вплоть до нашего времени.

2.2. История процессоров x86

История первых 16-разрядных процессоров класса x86, 8086, была начата компанией Intel в 1978 году. Чипы того времени работали на частоте 5, 8 или 10 МГц и благодаря 20-разрядной шине адреса позволяли адресовать 1 Мб оперативной памяти.

В то время были популярны 8-битные компьютеры, поэтому Intel разработала другой чип — 8088, который был аппаратно и программно совместим с 8086, но оснащен только 8-разрядной шиной.

В 1982 году Intel представила процессор 80286, который был обратно совместим с обеими предыдущими моделями, но использовал более «широкую», 24-разрядную, шину адреса. Этот процессор позволял адресовать 16 Мб оперативной памяти. Кроме расширенного набора команд (появилось несколько новых команд), данный процессор мог работать в двух режимах — реальном и защищенном.

Защищенный режим обеспечивал механизмы страничной организации памяти, прав доступа и переключения задач, которые необходимы для любой многозадачной операционной системы. Реальный режим использовался для обратной совместимости с предыдущими моделями x86.

Четыре года спустя, в 1986 году, Intel выпустила процессор 80386 DX, у которого обе шины (шина данных и шина адреса) были 32-разрядными. В то же время был выпущен процессор 80386 SX, который был во всем идентичен 80386 DX,

но только с 16-разрядной внешней шиной данных. Оба процессора работали на частоте 20, 25 или 33 МГц. Процессор 80386 не имел интегрированного математического сопроцессора, математический сопроцессор поставлялся в виде отдельного чипа — 80387.

В 1989 году было выпущено следующее поколение микропроцессоров Intel — 80486DX, 80486DX/2 и 80486DX/4, которые отличались только рабочей частотой. Выпущенная тогда же версия 80486SX, в отличие от 80486DX, поставлялась без математического сопроцессора. Новые возможности интеграции позволили разместить на чипе 8 Кб кэш-памяти.

В 1993 году был выпущен первый чип под названием Pentium. С него началась новая линия чипов, которые не только используются сейчас, но и все еще могут выполнять программы, написанные 20 лет назад для процессора 8086.

Процессоры, совместимые с x86, выпускались не только компанией Intel, но также и другими компаниями: AMD, Cyrix, NEC, IBM. Мы более подробно рассмотрим 80386, который с точки зрения программирования полностью совместим даже с самыми современными процессорами.

2.3. Процессоры и их регистры: общая информация

Поговорим о внутреннем строении процессора. **Процессор** — это кремниевая плата или «подложка» с логическими цепями, состоящими из транзисторов, скрытая в пластмассовом корпусе, снабженном контактными ножками (выводами, pin). Большинство ножек процессора подключено к шинам — шине адреса, шине данных и шине управления, связывающим чип процессора с остальной частью компьютера. Остальные ножки служат для подачи питания на сам чип. Каждая шина состоит из группы проводников, которые выполняют определенную функцию.

Пункт 7 концепции фон Неймана говорит: **ИНСТРУКЦИИ И ДАННЫЕ (ТО ЕСТЬ ОПЕРАНДЫ, РЕЗУЛЬТАТЫ ИЛИ АДРЕСА) ПРЕДСТАВЛЯЮТСЯ В ВИДЕ ДВОИЧНЫХ СИГНАЛОВ И В ДВОИЧНОЙ СИСТЕМЕ СЧИСЛЕНИЯ.**

Это означает, что один проводник шины компьютера может «нести» один бит. Значение этого бита (1 или 0) определяется уровнем напряжения в проводнике. Значит, процессор с одной 16-разрядной шиной и одной 8-разрядной должен иметь 24 (16 и 8) ножки, соединенные с различными проводниками. Например, при передаче числа 27 (00011011 в двоичной системе) по 8-разрядной шине проводник, по которому передается самый правый бит (LSB), покажет логический уровень 1, следующий провод также покажет 1, следующий — 0 и т.д.

Пока мы сказали, что процессор состоит из логических контуров. Эти цепи реализуют все модули, из которых процессор должен состоять согласно концепции фон Неймана: контроллер, арифметико-логическое устройство (АЛУ) и регистры.

Контроллер управляет получением инструкций из памяти и их декодированием. Контроллер не обрабатывает инструкцию: после декодирования он просто передает ее по внутренней шине управления к другим модулям, которые выполняют необходимое действие.

Арифметико-логическое устройство (АЛУ) выполняет арифметические и логические действия над данными. Для более простых процессоров достаточно АЛУ, умеющего выполнять операции отрицания и сложения, поскольку другие арифметические действия (вычитание, умножение и целочисленное деление) могут быть сведены к этим операциям.

Другая, логическая, часть АЛУ выполняет основные логические действия над данными, например, логическое сложение и умножение (ИЛИ, И), а также исключительное ИЛИ. Еще одна функция АЛУ, которую выполняет устройство циклического сдвига (barrel-shifter), заключается в сдвигах битов влево и вправо.

Для выполнения процессором инструкции необходимо намного меньше времени, чем для чтения этой инструкции из памяти. Чтобы сократить время ожидания памяти, процессор снабжен временным хранилищем инструкций и

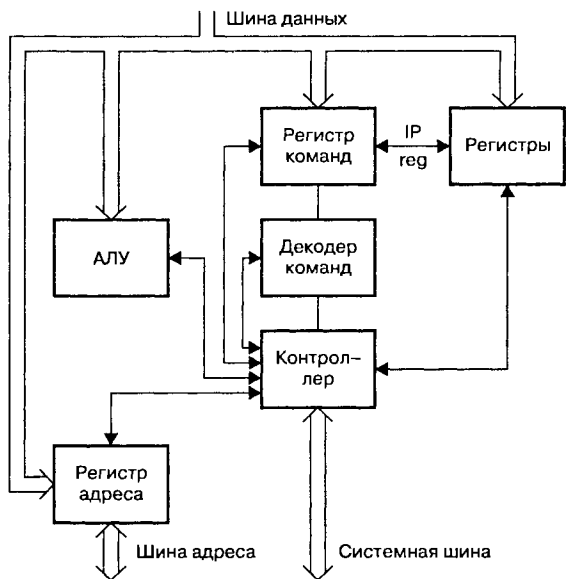


Рис. 2.2. Структура микропроцессора

данных — **регистрами**. Размер регистра — несколько байтов, но зато доступ к регистрам осуществляется почти мгновенно.

Среди регистров обязательно должны присутствовать следующие группы: регистры общего назначения, регистры состояния и счетчики. Регистры общего назначения содержат рабочие данные, полученные из памяти. Регистры состояния содержат текущее состояние процессора (или состояние АЛУ). Последняя группа — это счетчики. Согласно теории фон Неймана, должен быть хотя бы один регистр из этой группы — счетчик команд, содержащий адрес следующей инструкции. Как все это работает, мы расскажем в следующей главе.

2.4. Процессор 80386

Микропроцессор 80386 полностью 32-разрядный, что означает, что он может работать с 4 Гб оперативной памяти (2^{32} байтов). Поскольку шина данных также 32-разрядная, процессор может обрабатывать и хранить в своих регистрах число «шириной» в 32 бита (тип данных `int` в большинстве реализаций языка C как раз 32-разрядный).

Чтобы научиться программировать на языке ассемблера, мы должны знать имена регистров (рис. 2.3) и общий принцип работы команд. Сами команды обсуждаются в следующих главах.

Регистры общего назначения

Сначала рассмотрим регистры общего назначения. Они называются EAX, EBX, ECX и EDX (Аккумулятор, База, Счетчик и Данные). Кроме названий, они больше ничем другим не отличаются друг от друга, поэтому рассмотрим только первый регистр — **EAX** (рис. 2.4).

Процессор 80386 обратно совместим с процессором 80286, регистры которого 16-разрядные. Как же 80386 может выполнять команды, предназначенные для регистров меньшего размера? Регистр EAX может быть разделен на две части — 16-разрядный регистр AX (который также присутствует в 80286) и верхние 16 битов, которые никак не называются. В свою очередь, регистр AX может быть разделен (не только в 80386, но и в 80286) на два 8-битных регистра — AH и AL.

Если мы заносим в регистр EAX значение 0x12345678, то регистр AX будет содержать значение 0x5678 (0x56 в AH и 0x78 в AL), а значение 0x1234 будет помещено в верхнюю часть регистра EAX.

«Младшие» регистры других регистров общего назначения называются по такому же принципу: EBX содержит BX, который, в свою очередь, содержит BH и BL и т.д.

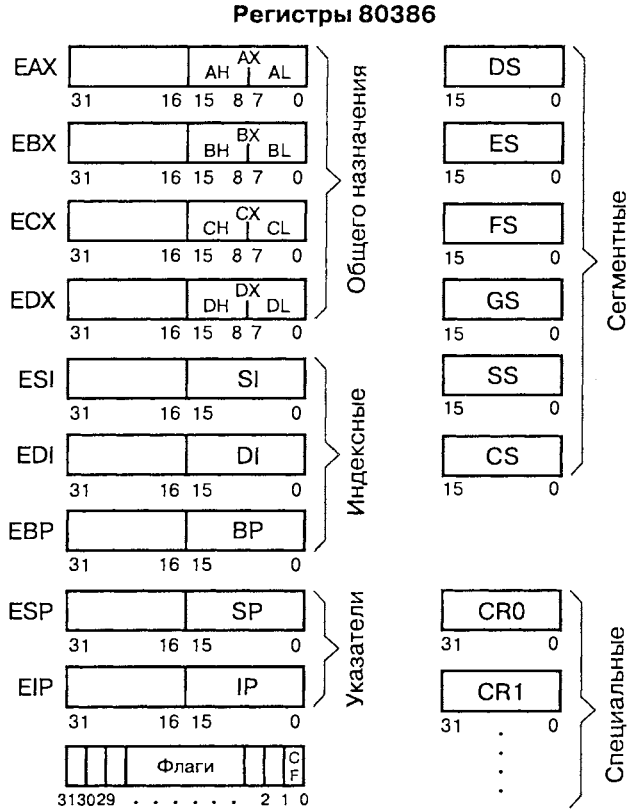


Рис. 2.3. Основные регистры процессора 80386

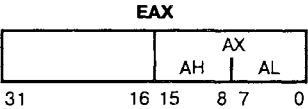


Рис. 2.4. Регистр EAX

Индексные регистры

К регистрам общего назначения иногда относят и индексные регистры процессора 80386 — ESI, EDI и EBP (или SI, DI и BP для 16-разрядных действий). Обычно эти регистры используются для адресации памяти: обращения к массивам, индексирования и т.д. Отсюда их имена: индекс источника (Source Index), индекс приемника (Destination Index), указатель базы (Base Pointer).

Но хранить в них только адреса совсем необязательно: регистры ESI, EDI и EBP могут содержать произвольные данные. Эти регистры программно доступны, то есть их содержание может быть изменено программистом. Другие регистры лучше «руками не трогать».

У регистров ESI, EDI и EBP существуют только в 16-разрядная и 32-разрядная версии.

Сегментные регистры

Эту группу регистров можно отнести к регистрам состояния. Регистры из этой группы используются при вычислении реального адреса (адреса, который будет передан на шину адреса). Процесс вычисления реального адреса зависит от режима процессора (реальный или защищенный) и будет рассмотрен в следующих главах. Сегментные регистры только 16-разрядные, такие же, как в 80286.

Названия этих регистров соответствуют выполняемым функциям: CS (Code Segment, сегмент кода) вместе с EIP (IP) определяют адрес памяти, откуда нужно прочитать следующую инструкцию; аналогично регистр SS (Stack Segment, сегмент стека) в паре с ESP (SS:SP) указывают на вершину стека. Сегментные регистры DS, ES, FS, и GS (Data, Extra, F и G сегменты) используются для адресации данных в памяти.

Регистры состояния и управления

Регистр ESP (SP) — это указатель памяти, который указывает на вершину стека (x86-совместимые процессоры не имеют аппаратного стека). О стеке мы поговорим в следующих главах. Также программно не может быть изменен регистр EIP (IP, Instruction Pointer) — указатель команд. Этот регистр указывает на инструкцию, которая будет выполнена следующей. Значение этого регистра изменяется непосредственно контроллером процессора согласно инструкциям, полученным из памяти.

Нам осталось рассмотреть только регистр флагов (иногда его называют регистром признаков) — EFLAGS. Он состоит из одnorазрядных флагов, отображающих в основном текущее состояние арифметико-логического устройства. В наших программах мы будем использовать все 32 флага, а пока рассмотрим только самые важные из них:

- ♦ **Признак нуля ZF (Zero Flag)** — 1, если результат предыдущей операции равен нулю.
- ♦ **Признак знака SF (Sign Flag)** — 1, если результат предыдущей операции отрицательный.
- ♦ **Признак переполнения OF (Overflow Flag)** — 1, если при выполнении предыдущей операции произошло переполнение (overflow), то есть результат операции больше, чем зарезервированная для него память.
- ♦ **Признак переноса CF (Carry Flag)** — 1, если бит был «перенесен» и стал битом более высокого порядка (об этом мы поговорим в четвертой главе, когда будем рассматривать арифметические операции).
- ♦ **Признак прерывания IF (Interrupt Flag)** — 1, если прерывания процессора разрешены.
- ♦ **Признак направления DF (Direction Flag)** — используется для обработки строк, мы рассмотрим подробнее этот регистр в шестой главе.

Другие регистры процессора относятся к работе в защищенном режиме, описание принципов которого выходит за рамки этой книги.

Если 80386 процессор оснащен математическим сопроцессором 80387 (это отдельный чип на вашей материнской плате), он будет быстрее обрабатывать числа с плавающей точкой.

Современным процессорам отдельный математический процессор не нужен — он находится «внутри» процессора. Раньше вы могли немного сэкономить и купить компьютер без математического сопроцессора — его наличие было необязательно, и компьютер мог работать без него. Если математический процессор не был установлен, его функции эмулировались основным процессором, так что производительность операций над числами с плавающей точкой была очень низкой.

Примечание.

Когда мы будем говорить сразу о 16- и 32-разрядных регистрах, то мы будем использовать сокращение (E)AX — вместо AX и EAX.

2.5. Прерывания

Событие прерывания состоит в том, что процессор прекращает выполнять инструкции программы в нормальной последовательности, а вместо этого начинает выполнять другую программу, предназначенную для обработки этого события. После окончания обработки прерывания процессор продолжит выполнение прерванной программы.

Давайте рассмотрим пример. Я сижу за столом и читаю книгу. С точки зрения компьютера, я выполняю процесс чтения книги. Внезапно звонит телефон —

я прерываю чтение, кладу в книгу закладку (на языке процессора это называется «сохранить контекст») и беру трубку. Теперь я «обрабатываю» телефонный звонок. Закончив разговор, я возвращаюсь к чтению книги. Найти последнее прочитанное место помогает та самая закладка.

Процессоры семейства x86 и совместимые с ними могут порождать 256 прерываний. Адреса всех 256 функций обработки прерываний (так называемые векторы прерываний) хранятся в специальной таблице векторов прерываний.

Прерывания могут быть программными и аппаратными.

Аппаратные прерывания происходят по запросу периферийных устройств и называются IRQ (Interrupt Requests). Архитектура шины ISA ограничивает их число до 16 (IRQ0 — IRQ15).

К аппаратным прерываниям относятся также специальные прерывания, которые генерирует сам процессор. Такие прерывания используются для обработки «исключительных ситуаций» — неверный операнд, неизвестная команда, переполнение и другие непредвиденные операции, когда процессор сбит с толку и не знает, что делать. Эти прерывания имеют свои обозначения и никак не относятся к зарезервированным для периферии прерываниям IRQ0-IRQ15.

Все аппаратные прерывания можно разделить на две группы: прерывания, которые можно игнорировать («замаскировать») и те, которые игнорировать нельзя. Первые называются маскируемыми (maskable), а вторые — немаскируемыми (non-maskable). Аппаратные прерывания могут быть отключены путем установки флага IF регистра признаков в 0. Единственное прерывание, которое отключить нельзя — это NMI, немаскируемое прерывание, генерирующееся при сбое памяти, сбое в питании процессора и подобных форс-мажорных обстоятельствах.

Программные прерывания генерируются с помощью специальной команды в теле программы, то есть их порождает программист. Обычно программные прерывания используются для «общения» вашей программы с операционной системой.

Глава 3

Анатомия команд и как они выполняются процессором



3.1. Как команды выполняются процессором

Команда микропроцессора — это команда, которая выполняет требуемое действие над данными или изменяет внутреннее состояние процессора.

Существует две основные архитектуры процессоров. Первая называется **RISC** (Reduced Instruction Set Computer) — компьютер с уменьшенным набором команд. Архитектура RISC названа в честь первого компьютера с уменьшенным набором команд — RISC I. Идея этой архитектуры основывается на том, что процессор большую часть времени тратит на выполнение ограниченного числа инструкций (например, переходов или команд присваивания), а остальные команды используются редко.

Разработчики RISC-архитектуры создали «облегченный» процессор. Благодаря упрощенной внутренней логике (меньшему числу команд, менее сложным логическим контурам), значительно сократилось время выполнения отдельных команд и увеличилась общая производительность. Архитектура RISC подобна «архитектуре общения» с собакой — она знает всего несколько команд, но выполняет их очень быстро.

Вторая архитектура имеет сложную систему команд, она называется **CISC** (Complex Instruction Set Computer) — компьютер со сложной системой команд. Архитектура CISC подразумевает использование сложных инструкций, которые можно разделить на более простые. Все x86-совместимые процессоры принадлежат к архитектуре CISC.

Давайте рассмотрим команду «загрузить число 0x1234 в регистр AX». На языке ассемблера она записывается очень просто — `MOV AX, 0x1234`. К настоящему моменту вы уже знаете, что каждая команда представляется в виде двоичного числа (пункт 7 концепции фон Неймана). Ее числовое представление называется машинным кодом. Команда `MOV AX, 0x1234` на машинном языке может быть записана так:

0x11xx: предыдущая команда

0x1111: 0xB8, 0x34, 0x12

0x1114: следующие команды

Мы поместили команду по адресу 0x1111. Следующая команда начинается тремя байтами дальше, значит, под команду с операндами отведено 3 байта. Второй и третий байты содержат операнды команды MOV. А что такое 0xB8? После преобразования 0xB8 в двоичную систему мы получим значение 10111000b.

Первая часть — 1011 — и есть код команды MOV. Встретив код 1011, контроллер «понимает», что перед ним — именно MOV. Следующий разряд (1) означает, что операнды будут 16-разрядными. Три последние цифры определяют регистр назначения. Три нуля соответствуют регистру AX (или AL, если предыдущий бит был равен 0, указывая таким образом, что операнды будут 8-разрядными).

Чтобы декодировать команды, контроллер должен сначала прочитать их из памяти. Предположим, что процессор только что закончил выполнять предшествующую команду, и IP (указатель команд) содержит значение 0x1111. Прежде чем приступить к обработке следующей команды, процессор «посмотрит» на шину управления, чтобы проверить, требуются ли аппаратные прерывания.

Если запроса на прерывание не поступало, то процессор загружает значение, сохраненное по адресу 0x1111 (в нашем случае — это 0xB8), в свой внутренний (командный) регистр. Он декодирует это значение так, как показано выше, и «понимает», что нужно загрузить в регистр AX 16-разрядное число — два следующих байта, находящиеся по адресам 0x1112 и 0x1113 (они содержат наше число, 0x1234). Теперь процессор должен получить из памяти эти два байта. Для этого процессор посылает соответствующие команды в шину и ожидает возвращения по шине данных значения из памяти.

Получив эти два байта, процессор запишет их в регистр AX. Затем процессор увеличит значение в регистре IP на 3 (наша команда занимает 3 байта), снова проверит наличие запросов на прерывание и, если таких нет, загрузит один байт по адресу 0x1114 и продолжит выполнять программу.

Если запрос на прерывание поступил, процессор проверит его тип, а также значение флага IF. Если флаг сброшен (0), процессор проигнорирует прерывание; если же флаг установлен (1), то процессор сохранит текущий контекст и начнет выполнять первую инструкцию обработчика прерывания, загрузив ее из таблицы векторов прерываний.

К счастью, нам не придется записывать команды в машинном коде, поскольку ассемблер разрешает использовать их символические имена. Но перед тем как углубиться в команды, поговорим об их операндах.

3.2. Операнды

Данные, которые обрабатываются командами, называются операндами. Операнды в языке ассемблера записываются непосредственно после команды; если их несколько, то через запятую. Одни команды вообще не имеют никаких операндов, другие имеют один или два операнда.

В качестве операнда можно указать непосредственное значение (например, 0x123), имя регистра или ссылку на ячейку памяти (так называемые косвенные операнды).

Что же касается разрядности, имеются 32-разрядные, 16-разрядные, и 8-разрядные операнды. Почти каждая команда требует, чтобы операнды были одинакового размера (разрядности). Команда `MOV AX, 0x1234` имеет два операнда: операнд регистра и непосредственное значение, и оба они 16-битные.

Последний тип операнда — косвенный тип — адресует данные, находящиеся в памяти, получает их из памяти и использует в качестве значения. Узнать этот операнд очень просто — по наличию в записи квадратных скобок. Адресация памяти будет рассмотрена в следующем параграфе.

В документации по Ассемблеру различные форматы операндов представлены следующими аббревиатурами:

- **reg8-операнд** — любой 8-разрядный регистр общего назначения;
- **reg16-операнд** — любой 16-разрядный регистр общего назначения;
- **reg32-операнд** — любой 32-разрядный регистр общего назначения;
- **m** — операнд может находиться в памяти;
- **imm8** — непосредственное 8-разрядное значение;
- **imm16** — непосредственное 16-разрядное значение;
- **imm32** — непосредственное 32-разрядное значение;
- **segreg** — операнд должен быть сегментным регистром.

Допускаются неоднозначные типы операндов, например: **reg8/imm8-операнд** может быть любым 8-битным регистром общего назначения или любым 8-битным непосредственным значением.

Иногда размер операнда определяется только по последнему типу, например, следующая запись аналогична предыдущей: **R/imm8-операнд** может быть любым регистром (имеется в виду 8-битный регистр) общего назначения или 8-разрядным значением.

3.3. Адресация памяти

Мы уже знаем, что адрес, как и сама команда, — это число. Чтобы не запоминать адреса всех «переменных», используемых в программе, этим адресам присваивают символические обозначения, которые называются переменными (иногда их также называют указателями).

При использовании косвенного операнда адрес в памяти, по которому находится нужное значение, записывается в квадратных скобках: [адрес]. Если мы используем указатель, то есть символическое представление адреса, например, [ESI], то в листинге машинного кода мы увидим, что указатель был заменен реальным значением адреса. Можно также указать точный адрес памяти, например, [0x594F].

Чаще всего мы будем адресовать память по значению адреса, занесенному в регистр процессора. Чтобы записать такой косвенный операнд, нужно просто написать имя регистра в квадратных скобках. Например, если адрес загружен в регистр ESI, вы можете получить данные, расположенные по этому адресу, используя выражение [ESI].

Теперь рассмотрим фрагмент программы, в которой регистр ESI содержит адрес первого элемента (нумерация начинается с 0) в массиве байтов. Как получить доступ, например, ко второму элементу (элементу, адрес которого на 1 байт больше) массива? Процессор поддерживает сложные способы адресации, которые очень нам пригодятся в дальнейшем. В нашем случае, чтобы получить доступ ко второму элементу массива, нужно записать косвенный операнд [ESI + 1].

Имеются даже более сложные типы адресации: [адрес + EBX + 4]. В этом случае процессор складывает *адрес*, значение 4 и значение, содержащееся в регистре *EBX*. Результат этого выражения называется эффективным адресом (EA, Effective Address) и используется в качестве адреса, по которому фактически находится операнд (мы пока не рассматриваем сегментные регистры). При вычислении эффективного адреса процессор 80386 также позволяет умножать один член выражения на константу, являющуюся степенью двойки: [адрес + EBX * 4]. Корректным считается даже следующее «сумасшедшее» выражение:

[число - 6 + EBX * 8 + ESI]

На практике мы будем довольствоваться только одним регистром [ESI] или суммой регистра и константы, например, [ESI + 4]. В зависимости от режима процессора, мы можем использовать любой 16-разрядный или 32-разрядный регистр общего назначения [EAX], [EBX], ... [EBP].

Процессор предыдущего поколения 80286 позволял записывать адрес в виде суммы содержимого регистра и константы только для регистров BP, SI, DI, и BX.

Выше мы упомянули, что в адресации памяти участвуют сегментные регистры. Их функция зависит от режима процессора. Каждый способ адресации предполагает, что при вычислении реального (фактического) адреса используется сегментный регистр по умолчанию. Сменить регистр по умолчанию можно так:

```
ES:[ESI]
```

Некоторые ассемблеры требуют указания регистра внутри скобок:

```
[ES:ESI]
```

В наших примерах мы будем считать, что все сегментные регистры содержат одно и то же значение, поэтому мы не будем использовать их при адресации.

3.4. Команды языка ассемблера

Когда мы знаем, что такое операнд, давайте рассмотрим, как описываются команды языка ассемблера. Общий формат такой:

```
имя_команды [подсказка] операнды
```

В следующих главах мы поговорим об отдельных командах и выполняемых ими функциях. Операнды мы только что рассмотрели, осталась одна «темная лошадка» — подсказка. Необязательная подсказка указывает компилятору требуемый размер операнда. Ее значением может быть слово BYTE (8-битный операнд), WORD (16-битный) или DWORD (32-битный).

Представим инициализацию некоторой «переменной» нулем, то есть запись нулей по адресу переменной. Подсказка сообщит компилятору размер операнда, то есть сколько именно нулевых байтов должно быть записано по этому адресу. Пока мы не знаем правильной инструкции для записи значения, поэтому будем считать, что записать значение можно так:

```
mov dword [ 0x12345678 ],0      ;записывает 4 нулевых байта,  
                                ;начиная с адреса 0x12345678  
mov word  [ 0x12345678 ],0      ;записывает 2 нулевых байта,  
                                ;начиная с адреса 0x12345678  
mov byte  [ 0x12345678 ],0      ;записывает 1 нулевой байт  
                                ;по адресу 0x12345678
```

Примечание.

В языке ассемблера точка с запятой является символом начала комментария.

Первая инструкция последовательно запишет 4 нулевых байта, начиная с адреса 0x12345678. Вторая инструкция запишет только два нулевых байта, поскольку размер операнда — слово. Последняя инструкция запишет только один байт (в битах: 00000000) в ячейку с адресом 0x12345678.

Глава 4 Основные команды языка ассемблера

- * Команда MOV
- * «Остроконечники» и «тупоконечники»
- * Арифметические команды
- * Логические команды

В этой главе рассмотрены основные команды процессоров семейства x86, которые являются фундаментом и простых, и сложных программ, написанных на языке ассемблера. Мы не только опишем синтаксис команд, но и приведем несколько практических примеров, которые пригодятся вам при написании собственных программ.

4.1. Команда MOV

Прежде чем изменять каким-либо образом наши данные, давайте научимся их сохранять: копировать из регистра в память и обратно. Ведь прежде чем оперировать данными в регистрах, их сначала туда надо поместить.

Команда MOV, хоть название ее и происходит от слова «move» (перемещать), на самом деле не перемещает, а копирует значение из источника в приемник:

MOV приемник, источник

Рассмотрим несколько примеров применения команды MOV:

```
mov ax,[number]      ;заносим значение переменной number
                     ;в регистр AX
mov [number],bx       ;загрузить значение регистра BX
                     ;в переменную number
mov bx,cx             ;занести в регистр BX значение
                     ;регистра CX
mov al,1              ;занести в регистр AL значение 1
mov dh,cl             ;занести в регистр DH значение
                     ;регистра CL
mov esi,edi           ;копировать значение регистра EDI
                     ;в регистр ESI
mov word [number],1   ;сохранить 16-битное значение 1
                     ;в переменную "number"
```

Процессоры семейства x86 позволяют использовать в командах только один косвенный аргумент. Следующая команда копирования значения, находящегося по адресу number_one, в область памяти с адресом number_two, недопустима:

```
mov [number_two],[number_one] ;НЕПРАВИЛЬНО!!!
```

Чтобы скопировать значение из одной области памяти в другую, нужно использовать промежуточный регистр:

```
mov ax, [number_one]    ;загружаем в AX 16-битное
                        ;значение "number_one"
mov [number_two], ax    ;а затем копируем его в переменную
                        ;"number_two"
```

Оба операнда команды MOV должны быть одного размера:

```
mov ax, bl              ;НЕПРАВИЛЬНО! — Операнды разных
                        ;размеров.
```

Для копирования значения BL в регистр AX мы должны «расширить диапазон», то есть скопировать весь BX в AX, а затем загрузить 0 в AH:

```
mov ax, bx              ;загружаем BX в AX
mov ah, 0               ;"сбрасываем" верхнюю часть
                        ;AX — записываем в нее 0
```

Регистр AH является верхней 8-битной частью регистра AX. После выполнения команды **MOV ax, bx** регистр AH будет содержать значение верхней части регистра BX, то есть значение регистра BH. Но мы не можем быть уверены, что BH содержит 0, поэтому мы должны загрузить 0 в AH — команда **MOV ah, 0** «сбрасывает» значение регистра AH. В результате мы расширили 8-битное значение, ранее содержащееся в регистре BL, до 16 битов. Новое, 16-битное, значение будет находиться в регистре AX.

Можно поступить и наоборот: сначала сбросить весь AX, а затем загрузить BL в младшую часть AX (AL):

```
mov ax, 0               ;AH = 0, AL = 0
mov al, bl              ;заносим в AL значение BL
```

Точно так же можно скопировать 16-битное значение в 32-битный регистр.

Для полноты картины приведем список всех допустимых форматов команды MOV — как в официальной документации:

```
MOV r/m8, reg8
MOV r/m16, reg16
MOV r/m32, reg32
MOV reg8, r/m8
MOV reg16, r/m16
MOV reg32, r/m32
MOV reg8, imm8
MOV reg16, imm16
MOV reg32, imm32
MOV r/m8, imm8
MOV r/m16, imm16
MOV r/m32, imm32
```

4.2. «Остроконечники» и «тупоконечники»

Сейчас мы немного отклонимся от обсуждения команд. Предположим, что вы хотите проверить, было ли значение 0x12345678, которое содержалось в регистре ЕВР, корректно загружено в 32-разрядную переменную counter. Следующий фрагмент кода заносит значение 0x12345678 в переменную counter:

```
mov ebp, 0x12345678    ;загружаем в ЕВР значение 0x12345678
mov [counter], ebp     ;сохраняем значение ЕВР
                        ;в переменную "counter" (счетчик)
```

Для того чтобы проверить, загружено значение или нет, нужно воспользоваться отладчиком. Понимаю, что вы пока не знаете ни того, как откомпилировать программу, ни того, как загрузить ее в отладчик, но давайте представим, что это уже сделано за вас.

Как будет выглядеть откомпилированная программа в отладчике? Отладчик преобразует все команды из машинного кода назад в язык ассемблера. Все точно так же, как мы написали, только с небольшим отличием: везде, где мы использовали символьное имя переменной, оно будет заменено реальным адресом переменной, например:

```
0804808A      BD78563412      mov ebp, 0x12345678
0804808F      892DC0900408    mov dword [+0x80490c0], ebp
```

Первая колонка — это реальный адрес команды в памяти, вторая — машинный код, в который превратилась команда после компиляции. После этого мы видим символьное представление машинной команды в мнемокодах ассемблера. Символическое имя нашей переменной counter было заменено адресом этой переменной в памяти (0x80490c0).

Перед выполнением первой команды, `mov ebp, 0x12345678`, регистры процессора содержали следующие значения:

```
eax = 0x00000000 ebx = 0x00000000 ecx = 0x00000000 edx = 0x00000000
esp = 0xBFFFFFF90 ebp = 0x00000000 esi = 0x00000000 edi = 0x00000000
ds  = 0x0000002B es  = 0x0000002B fs  = 0x00000000 gs  = 0x00000000
ss  = 0x0000002B cs  = 0x00000023 eip = 0x0804808A eflags = 0x00200346
Flags: PF ZF TF IF ID
```

После выполнения первой команды значение регистра ЕВР было заменено значением 0x12345678. Если посмотреть дамп памяти по адресу нашей переменной (0x80490c0), то мы увидим следующее:

```
Dumping 64 bytes of memory starting at 0x080490C0 in hex
080490C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```


Когда будет выполнена вторая команда MOV, значение 0x12345678 будет записано в память по адресу 0x80490c0 и дамп памяти покажет другой результат:

```
Dumping 64 bytes of memory starting at 0x080490C0 in hex
080490C0: 78 56 34 12 00 00 00 00 00 00 00 00 00 00 00 00 xV4
```

Требуемое значение (0x12345678) действительно было записано по адресу 0x80490c0, но почему-то задом наперед. Дело в том, что все x86-процессоры относятся к классу LITTLE_ENDIAN, то есть хранят байты слова или двойного слова в порядке от наименее значимого к наиболее значимому («little-end-first», младшим байтом вперед). Процессоры BIG_ENDIAN (например, Motorola) поступают наоборот: размещают наиболее значимый байт по меньшему адресу («big-end-first», старшим байтом вперед).

Примечание.

Любопытно, что термины LITTLE_ENDIAN и BIG_ENDIAN — это не просто сокращения: они происходят от названий соперничавших в Лилипутии партий «остроконечников» и «тупоконечников» из «Путешествий Гулливера» Джонатана Свифта, отсюда и название этого пункта. «Остроконечники» и «тупоконечники» у Свифта кушали яйца с разных концов, одни — с острого, другие — с тупого. В результате спора по поводу того, как правильнее, развязалась война, а дальше ... если не помните, возьмите книжку и почитайте.

Порядок следования байтов в слове (двойном слове) учитывается не только при хранении, но и при передаче данных. Если у вас есть небольшой опыт программирования, возможно, вы сталкивались с «остроконечниками и тупоконечниками» при разработке сетевых приложений, когда некоторые структуры данных приходилось преобразовывать к «тупоконечному» виду при помощи специальных функций (htonl, htons, ntohl, ntohs).

Когда переменная counter будет прочитана обратно в регистр, там окажется оригинальное значение, то есть 0x12345678.

4.3. Арифметические команды

Рассмотренная выше команда MOV относится к группе команд перемещения значений, другие команды из которой мы рассмотрим позже. А сейчас перейдем к следующей группе чаще всего используемых команд — арифметическим операциям. Процессор 80386 не содержит математического сопроцессора, поэтому мы рассмотрим только целочисленную арифметику, которая полностью поддерживается процессором 80386. Каждая арифметическая команда изменяет регистр признаков.

4.3.1. Инструкции сложения ADD и вычитания SUB

Начнем с самого простого — сложения (ADD) и вычитания (SUB). Команда ADD требует двух операндов, как и команда MOV:

```
ADD o1, o2
```

Команда ADD складывает оба операнда и записывает результат в o1, предыдущее значение которого теряется. Точно так же работает команда вычитания — SUB:

```
SUB o1, o2
```

Результат, o1-o2, будет сохранен в o1, исходное значение o1 будет потеряно.

Теперь рассмотрим несколько примеров:

```
mov ax, 8           ; заносим в AX число 8
mov cx, 6           ; заносим в CX число 6
mov dx, cx          ; копируем CX в DX, DX = 6
add dx, ax          ; DX = DX + AX
```

Поскольку мы хотим избежать уничтожения (то есть перезаписи результатом) исходных значений и хотим сохранить оба значения — AX и CX, мы копируем оригинальное значение CX в DX, а затем добавляем AX к DX. Команда ADD сохранит результат DX + AX в регистре DX, а исходные значения AX и CX останутся нетронутыми.

Рассмотрим еще примеры использования инструкций ADD и SUB:

```
add eax, 8           ; EAX = EAX + 8
sub ecx, ebp         ; ECX = ECX - EBP
add byte [number], 4 ; добавляем значение 4
                     ; к переменной number
                     ; размером в 1 байт
                     ; (диапазон значений 0-255)
sub word [number], 4 ; number = number - 4
                     ; размером в 2 байта
                     ; (диапазон значений 0-65535)
add dword [number], 4 ; добавляем значение 00000004
                     ; к "number"
sub byte [number], al ; вычитаем значение регистра AL
                     ; из "number"
sub ah, al           ; вычитаем AL из AH, результат
                     ; помещаем в AH
```

Что произойдет, если сначала занести в AL (8-разрядный регистр) наибольшее допустимое значение (255), а затем добавить к нему 8?

```
mov al, 255          ; заносим в AL значение 255, то есть 0xFF
add al, 8             ; добавляем 8
```

В результате в регистре AL мы получим значение 7.

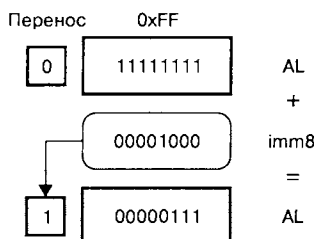


Рис. 4.1. Сложение 255 (0xFF) + 8

Но мы ведь ожидали 0x107 (263 в десятичном виде). Что случилось? В регистре AL может поместиться только 8-разрядное число (максимальное значение — 255). Девятый, «потерянный», бит скрыт в регистре признаков, а именно в флаге CF — признак переноса. Признак переноса используется в арифметических командах при работе с большими диапазонами чисел, чем могут поддерживать регистры. Полезны для этого команды ADC (Add With Carry — сложение с переносом) и SBB (Subtract With Borrow — вычитание с займом):

```
ADC o1, o2    ;o1 = o1 + o2 + CF
SBB o1, o2    ;o1 = o1 - o2 - CF
```

Эти команды работают так же, как ADD и SUB, но соответственно добавляють или вычитают флаг переноса CF.

В контексте арифметических операций очень часто используются так называемые пары регистров. Пара — это два регистра, использующихся для хранения одного числа. Часто используется пара EDX:EAX (или DX:AX) — обычно при умножении. Регистр AX хранит младшие 16 битов числа, а DX — старшие 16 битов. Таким способом даже древний 80286 может обрабатывать 32-разрядные числа, хотя у него нет ни одного 32-разрядного регистра.

Пример: пара DX:AX содержит значение 0xFFFF (AX = 0xFFFF, DX = 0). Добавим 8 к этой паре и запишем результат обратно в DX:AX:

```
mov ax, 0xffff    ;AX = 0xFFFF
mov dx, 0          ;DX = 0
add ax, 8          ;AX = AX + 8
adc dx, 0          ;добавляем 0 с переносом к DX
```

Первая команда ADD добавит 8 к регистру AX. Полностью результат не помещается в AX, поэтому его старший бит переходит в CF. Вторая команда добавит к DX значение 0 и значение CF.

После выполнения ADC флаг CF будет добавлен к DX (DX теперь равен 1). Результат сложения 0xFFFF и 8 (0x10007) будет помещен в пару DX:AX (DX=1, AX=0007).

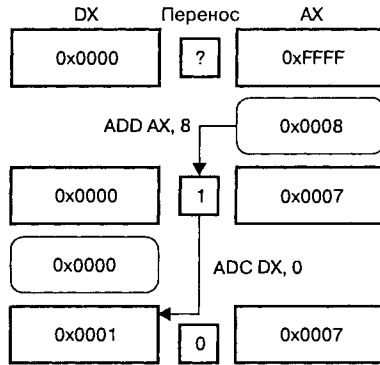


Рис. 4.2. Сложение чисел 0xFFFF и 0x0008 и их сохранение в регистрах

Процессор 80386 может работать с 32-разрядными числами напрямую — безо всяких переносов:

```
mov eax, 0xffff    ;EAX = 0xFFFF
add eax, 8          ;EAX = EAX + 8
```



Рис. 4.3. Использование 32-разрядных регистров процессора 80386

После выполнения этих инструкций в EAX мы получим 32-разрядное значение 0x10007. Для работы с 64-разрядными числами мы можем использовать пару EDX:EAX — точно так же, как мы использовали пару DX:AX.

4.3.2. Команды инкрементирования INC и декрементирования DEC

Эти команды предназначены для инкрементирования и декрементирования. Команда INC добавляет, а DEC вычитает единицу из единственного операнда. Допустимые типы операнда — такие же, как у команд ADD и SUB, а формат команд таков:

```
INC o1              ;o1 = o1 + 1
DEC o1              ;o1 = o1 - 1
```

ВНИМАНИЕ! Ни одна из этих инструкций не изменяет флаг CF. О значении этого факта и следствиях из него, а также о том, как безопасно (без потери

данных) использовать эти команды, будет сказано в главе, посвященной оптимизации.

Увеличение на единицу значения регистра AL выглядит следующим образом:

```
add al, 1      ;AL = AL + 1
inc al         ;AL = AL + 1
```

Увеличение на единицу значения 16-битной переменной number:

```
inc word [number] ;мы должны указать размер
                  ;переменной — word
```

4.3.3. Отрицательные числа — целые числа со знаком

Отрицательные целые числа в ПК представлены в так называемом дополнительном коде. Дополнительный код можно представить себе как отображение некоторого диапазона, включающего положительные и отрицательные целые числа, на другой диапазон, содержащий только положительные целые числа. Рассмотрим код дополнения одного байта.

Один байт может содержать числа в диапазоне от 0 до 255. Код дополнения заменяет этот диапазон другим — от -128 до 127. Диапазон от 0 до 127 отображается сам на себя, а отрицательным числам сопоставляется диапазон от 128 до 255: числу -1 соответствует число 255 (0xFF), -2 — 254 (0xFE) и т.д. Число -50 будет представлено как 206. Обратите внимание: самый старший бит отрицательного числа всегда устанавливается в 1 — так можно определить, что число отрицательное. Процесс отображения отрицательных чисел в дополнительный код иногда называют **малпингом** (mapping).

Дополнительный код может быть расширен до 2 байтов (от 0 до 65535). Он будет охватывать диапазон чисел от -32768 до 32767. Если дополнительный код расширить до 4 байтов, то получим диапазон от -2 147 483 648 до 2 147 483 647. Во многих языках программирования именно этот диапазон используется для целочисленного типа данных (integer).

Пример: преобразуем числа 4, -4, 386, -8000 и 45000 в дополнительный код, считая, что целевой диапазон — 16 бит (2 байта).

Прежде всего, выясним, сколько чисел может поместиться в 16 разрядов. Это очень просто: возведем 2 в степень 16. $2^{16} = 65\,536$, что соответствует диапазону от 0 до 65 535. Теперь определим границы: $65\,536 / 2 = 32\,768$. Итак, мы получили диапазон от -32 768 до 32 767 (0 — это положительное число!).

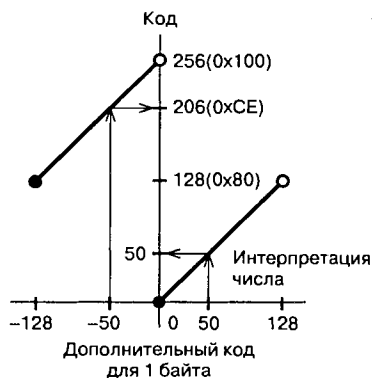


Рис. 4.4. Дополнительный код для 1 байта

Первое число, 4, попадает в диапазон $<0, 32\,767>$, поэтому оно отображается само на себя — в целевом диапазоне это будет тоже число 4. Число -4 — отрицательное, оно находится в диапазоне $<-32\,768, 0>$. В целевом диапазоне оно будет представлено как $65\,536 - 4 = 65\,532$. Число 386 останется само собой. Число $-8\,000$ — отрицательное, в результате отображения получается $65\,536 - 8\,000 = 57\,536$ — это и будет число $-8\,000$ в дополнительном коде. И, наконец, число 45 000 не может быть представлено в дополнительном коде, поскольку оно выходит за пределы диапазона.

Выполнять арифметические операции над отрицательными числами в дополнительном коде можно при помощи обычных команд ADD и SUB. Рассмотрим, как это происходит, на примере суммы чисел -6 и 7 в дополнительном коде из 2 байтов. Число 7 будет отображено само в себя, а число -6 будет представлено числом $65\,536 - 6 = 65\,530$ ($0xFFFFA$). Что получится, если мы сложим два эти числа (7 и $65\,530$)? Попробуем решить эту задачу на языке ассемблера:

```
mov ax,0xFFFFA      ;AX = -6, то есть 65530 или 0xFFFFA
mov dx,7             ;DX = 7
add ax,dx            ;AX = AX + DX
```

Мы получим результат $65\,530 + 7 = 65\,537 = 0x10001$, который не помещается в регистре AX, поэтому будет установлен флаг переноса. Но если мы его проигнорируем, то оставшееся в AX значение будет правильным результатом! Механизм дополнительного кода ввели именно для того, чтобы при сложении и вычитании отрицательных чисел не приходилось выполнять дополнительных действий.

Теперь давайте сложим два отрицательных числа. Ассемблер NASM позволяет указывать отрицательные числа непосредственно, поэтому нам не нужно преобразовывать их вручную в дополнительный код:

```
mov ax, -6           ;AX = -6
mov dx, -6           ;DX = -6
add ax,dx            ;AX = AX + DX
```

Результат: $0xFFFF4$ (установлен также флаг CF, но мы его игнорируем). В десятичной системе $0xFFFF4 = 65\,524$. В дополнительном коде мы получим правильный результат: -12 ($65\,536 - 65\,524 = 12$).

Отрицательные числа также могут использоваться при адресации памяти. Пусть регистр BX содержит адрес, а нам нужен адрес предыдущего байта, но мы не хотим изменять значение регистра BX (предполагается, что процессор находится в реальном режиме):

```
mov ax,[bx-1]        ;поместить в AX значение по адресу
                     ;на единицу меньшему, чем хранится в BX
```

Значение -1 будет преобразовано в $0xFFFF$, и инструкция будет выглядеть так: `MOV AX, [BX+0xFFFF]`. При вычислении адреса не учитывается флаг CF, поэтому мы получим адрес, на единицу меньший.

4.3.4. Команды для работы с отрицательными числами

Команда NEG

Система команд процессора 80386 включает в себя несколько команд, предназначенных для работы с целыми числами со знаком. Первая из них — команда NEG (negation, отрицание):

```
NEG r/m8
NEG r/m16
NEG r/m32
```

Используя NEG, вы можете преобразовывать положительное целое число в отрицательное и наоборот. Инструкция NEG имеет только один операнд, который может быть регистром или адресом памяти. Размер операнда — любой: 8, 16 или 32 бита.

```
neg eax          ;изменяет знак числа, сохраненного в EAX
neg bl           ;то же самое, но используется 8-битный
                  ;регистр bl
neg byte [number] ;изменяет знак 8-битной переменной number
```

Расширение диапазона целого беззнакового числа делалось просто: мы просто копировали число в больший регистр, а расширенное «место» заполняли нулями. При работе с целыми числами со знаком мы должны заполнить это место старшим битом преобразуемого числа. Так мы можем сохранять положительные и отрицательные числа при расширении их диапазона. Расширение диапазона числа со знаком называется знаковым расширением.

Процессор имеет несколько специальных команд, предназначенных для знакового расширения. Эти команды не имеют операндов, они выполняют действия над фиксированными регистрами.

Команда CBW

Команда CBW копирует седьмой (старший) бит регистра AL в регистр AH, расширяя таким образом оригинальное значение регистра AL в значение со знаком регистра AX (значение AH становится равно 0x00 или 0xFF = 1111111b, в зависимости от старшего бита AL). Сложно? Ничего, скоро рассмотрим пару примеров, и все станет на свои места.

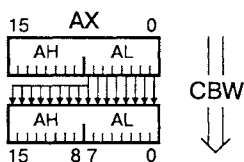


Рис. 4.5. Знаковое расширение с помощью инструкции CBW

Команда CWD

Команда CWD копирует старший бит AX в регистр DX, расширяя таким образом оригинальное значение AX в пару регистров со знаком DX:AX.

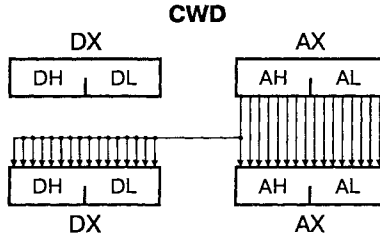


Рис. 4.6. Знаковое расширение с помощью инструкции CWD

Команда CDQ

Команда CDQ копирует старший бит EAX в регистр EDX, расширяя таким образом оригинальное значение EAX в пару регистров со знаком EDX:EAX.

Команда CWDE

Команда CWDE копирует старший бит AX в верхнюю часть (старшую часть) EAX, расширяя таким образом оригинальное значение AX в двойное слово со знаком, которое будет помещено в регистр EAX.

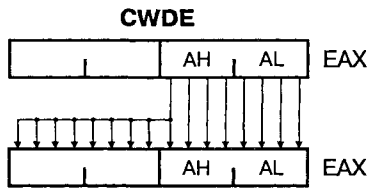


Рис. 4.7. Знаковое расширение с помощью инструкции CWDE

Рассмотрим пару примеров:

```
mov al, -1      ; AL = -1 (или 0xFF)
cbw             ; знаковое расширение на весь AX
```

После выполнения команды CBW AX будет содержать значение 0xFFFF, то есть -1. Единица (1) старшего разряда заполнила все биты AH, и мы получили знаковое расширение AL на весь регистр AX.


```
mov ax, 4      ; AX = 4
cwd           ; выполним знаковое расширение в DX
```

Первая команда заносит в регистр AX значение 4. Вторая команда, CWD, производит знаковое расширение AX в пару DX:AX. Оригинальное значение DX заменяется новым значением — старшим битом регистра AX, который в этом случае равен 0. В результате мы получили 0 в регистре DX.

Иногда команда CWD полезна для очищения регистра DX, когда AX содержит положительное значение, то есть значение, меньшее 0x8000.

4.3.5. Целочисленное умножение и деление

Давайте познакомимся с оставшимися целочисленными операциями: умножением и делением. Первое арифметическое действие выполняется командой MUL, а второе — командой DIV.

Дополнительный код делает возможным сложение и вычитание целых чисел со знаком и без знака с помощью одних и тех же команд ADD и SUB. Но к умножению и делению это не относится: для умножения и деления чисел со знаком служат отдельные команды — IMUL и IDIV. Операнды этих инструкций такие же, как у MUL и DIV.

Операции умножения и деления имеют свою специфику. В результате умножения двух чисел мы можем получить число, диапазон которого будет в два раза превышать диапазон операндов. Деление целых чисел — это операция целочисленная, поэтому в результате образуются два значения: частное и остаток.

С целью упрощения реализации команд умножения и деления эти команды спроектированы так, что один из операндов и результат находятся в фиксированном регистре, а второй операнд указывается программистом.

Подобно командам ADD и SUB, команды MUL, DIV, IMUL, IDIV изменяют регистр признаков.

Команды MUL и IMUL

Команда MUL может быть записана в трех различных форматах — в зависимости от операнда:

```
MUL r/m8
MUL r/m16
MUL r/m32
```

В 8-разрядной форме операнд может быть любым 8-битным регистром или адресом памяти. Второй операнд всегда хранится в AL. Результат (произведение) будет записан в регистр AX.

```
(r/m8) * AL -> AX
```

В 16-разрядной форме операнд может быть любым 16-битным регистром или адресом памяти. Второй операнд всегда хранится в AX. Результат сохраняется в паре DX:AX.

```
(r/m16) * AX -> DX:AX
```

В 32-разрядной форме второй операнд находится в регистре EAX, а результат записывается в пару EDX:EAX.

```
(r/m32) * EAX -> EDX:EAX
```

Рассмотрим несколько примеров.

Пример 1: умножить значения, сохраненные в регистрах BH и CL, результат сохранить в регистр AX:

```
mov al, bh      ;AL = BH — сначала заносим в AL второй операнд
mul cl          ;AX = AL * CL — умножаем его на CL
```

Результат будет сохранен в регистре AX.

Пример: вычислить 486^2 , результат сохранить в DX:AX:

```
mov ax, 486     ; AX = 486
mul ax          ; AX * AX -> DX:AX
```

Пример 2: вычислить диаметр по радиусу, сохраненному в 8-битной переменной radius1, результат записать в 16-битную переменную diameter1:

```
mov al, 2        ; AL = 2
mul byte [radius1] ; AX = radius * 2
mov [diameter1],ax ; diameter <- AX
```

Вы можете спросить, почему результат 16-разрядного умножения сохранен в паре DX:AX, а не в каком-то 32-разрядном регистре? Причина — совместимость с предыдущими 16-разрядными процессорами, у которых не было 32-разрядных регистров.

Команда IMUL умножает целые числа со знаком и может использовать один, два или три операнда. Когда указан один операнд, то поведение IMUL будет таким же, как и команды MUL, просто она будет работать с числами со знаком.

Если указано два операнда, то инструкция IMUL умножит первый операнд на второй и сохранит результат в первом операнде, поэтому первый операнд всегда должен быть регистром. Второй операнд может быть регистром, непосредственным значением или адресом памяти.

```
imul edx,ecx      ;EDX = EDX * ECX
imul ebx,[sthing] ;умножает 32-разрядную переменную
                  ;"sthing" на EBX, результат будет
                  ;сохранен в EBX
```

```
imul ecx,6           ;ECX = ECX * 6
```

Если указано три операнда, то команда IMUL перемножит второй и третий операнды, а результат сохранит в первый операнд. Первый операнд — только регистр, второй может быть любого типа, а третий должен быть только непосредственным значением:

```
imul edx,ecx,7       ;EDX = ECX * 7
imul ebx,[sthing],9   ;умножаем переменную "sthing" на 9,
                      ;результат будет сохранен EBX
imul ecx,edx,11       ;ECX = EDX * 11
```

Команды DIV и IDIV

Подобно команде MUL, команда DIV может быть представлена в трех различных форматах в зависимости от типа операнда:

```
DIV r/m8
DIV r/m16
DIV r/m32
```

Операнд служит делителем, а делимое находится в фиксированном месте (как в случае с MUL). В 8-битной форме переменный операнд (делитель) может быть любым 8-битным регистром или адресом памяти. Делимое содержится в AX. Результат сохраняется так: частное — в AL, остаток — в AH.

$AX / (r/m8) \rightarrow AL, \text{остаток} \rightarrow AH$

В 16-битной форме операнд может быть любым 16-битным регистром или адресом памяти. Второй операнд всегда находится в паре DX:AX. Результат сохраняется в паре DX:AX (DX — остаток, AX — частное).

$DX:AX / (r/m16) \rightarrow AX, \text{остаток} \rightarrow DX$

В 32-разрядной форме делимое находится в паре EDX:EAX, а результат записывается в пару EDX:EAX (частное в EAX, остаток в EDX).

$EDX:EAX / (r/m32) \rightarrow EAX, \text{остаток} \rightarrow EDX$

Команда IDIV используется для деления чисел со знаком, синтаксис ее такой же, как у команды DIV.

Рассмотрим несколько примеров.

Пример 1: разделить 13 на 2, частное сохранить в BL, а остаток в — BH:

```
mov ax,13           ; AX = 13
mov cl,2            ; CL = 2
div cl              ; делим на CL
mov bx,ax           ; ожидаемый результат находится в AX,
                    ; копируем в BX
```

Пример 2: вычислить радиус по диаметру, значение которого сохранено в 16-битной переменной `diameter1`, результат записать в `radius1`, а остаток проигнорировать.

```
mov ax,[diameter1]    ;AX = diameter1
mov bl,2              ;загружаем делитель 2
div bl                ;делим
mov [radius1],al      ;сохраняем результат
```

4.4. Логические команды

К логическим операциям относятся: логическое умножение (И, AND), логическое сложение (ИЛИ, OR), исключающее ИЛИ (XOR) и отрицание (NOT). Все эти инструкции изменяют регистр признаков.

Команда AND

Команда AND выполняет логическое умножение двух операндов — `o1` и `o2`. Результат сохраняется в операнде `o1`. Типы операндов такие же, как у команды ADD: операнды могут быть 8-, 16- или 32-битными регистрами, адресами памяти или непосредственными значениями.

AND `o1`, `o2`

Таблица истинности для оператора AND приведена ниже (табл. 4.1).

Таблица истинности для оператора AND

Таблица 4.1

A	b	a AND b
0	0	0
0	1	0
1	0	0
1	1	1

Следующий пример вычисляет логическое И логической единицы и логического нуля (`1 AND 0`).

```
mov al,1             ; AL = one
mov bl,0             ; BL = zero
and al,bl            ; AL = AL and BL = 0
```

Тот же самый пример, но записанный более компактно:

```
mov al,1             ; AL = one
and al,0             ; AL = AL and 0 = 1 and 0 = 0
```

Команда OR

Команда OR выполняет логическое сложение двух операндов — o1 и o2. Результат сохраняется в операнде o1. Типы операндов такие же, как у команды AND.

```
OR o1, o2
```

Таблица истинности для оператора OR приведена ниже (табл. 4.2).

Таблица истинности для оператора OR

Таблица 4.2

A	b	a OR b
0	0	0
0	1	1
1	0	1
1	1	1

Дополнительные примеры использования логических команд будут приведены в последнем пункте данной главы. А пока рассмотрим простой пример установки наименее значимого бита (первый справа) переменной mask в 1.

```
or byte [mask],1
```

Команда XOR

Вычисляет так называемое «исключающее ИЛИ» операндов o1 и o2. Результат сохраняется в o1. Типы операндов такие же, как у предыдущих инструкций. Формат команды:

```
XOR o1, o2
```

Таблица истинности для оператора XOR приведена ниже (табл. 4.3).

Таблица истинности для оператора XOR

Таблица 4.3

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

Исключающее ИЛИ обратимо: выражение ((x хог у) хог у) снова возвратит x.

```
mov al,0x55    ; AL = 0x55
xor al,0xAA     ;AL = AL xor 0xAA
xor al,0xAA     ;возвращаем в AL исходное значение — 0x55
```

Команда NOT

Используется для инверсии отдельных битов единственного операнда, который может быть регистром или памятью. Соответственно команда может быть записана в трех различных форматах:

```
NOT r/m8
NOT r/m16
NOT r/m32
```

Таблица истинности для оператора NOT приведена ниже (табл. 4.4).

Таблица истинности для оператора NOT

Таблица 4.4

A	NOT a
0	1
1	0

Следующий пример демонстрирует различие между операциями NOT и NEG:

```
mov al,00000010b    ;AL = 2
mov bl,al            ;BL = 2
not al               ;после этой операции мы получим
                    ;11111101b = 0xFD (-3)
neg bl              ;а после этой операции результат будет
                    ;другим: 11111110 = 0xFE (-2)
```

Массивы битов (разрядные матрицы)

Любое число можно записать в двоичной системе в виде последовательности нулей и единиц. Например, любое 16-разрядное число состоит из 16 двоичных цифр — 0 и 1. Мы можем использовать одно число для хранения шестнадцати различных состояний — флагов. Нам не нужно тратить место на хранение 16 различных переменных, ведь для описания состояния (включено/выключено) вполне достаточно 1 бита. **Переменная, используемая для хранения флагов, называется разрядной матрицей или массивом битов.**

Высокоуровневые языки программирования также используют разрядные матрицы, например, при хранении набора значений перечисления, для экономии памяти. Мы уже знакомы с одной разрядной матрицей, которая очень часто используется в программировании — это регистр признаков микропроцессора. Мы будем очень часто сталкиваться с разрядными матрицами при программировании различных устройств: видеоадаптера, звуковой платы и т.д. В этом случае изменение одного бита в разрядной матрице может изменить режим работы устройства.

Для изменения значения отдельных битов в матрице служат логические операции. Первый операнд задает разрядную матрицу, с которой мы будем работать, а второй операнд задает так называемую маску, используемую для выбора отдельных битов.

Для установки определенных битов массива в единицу (все остальные биты при этом должны остаться без изменения) применяется команда OR. В качестве маски возьмите двоичное число, в котором единицы стоят на месте тех битов, которые вы хотите установить в массиве. Например, если вы хотите установить первый и последний биты массива, вы должны использовать маску 10000001. Все остальные биты останутся нетронутыми, поскольку 0 OR X всегда возвращает X.

Чтобы сбросить некоторые биты (установить их значение в 0), возьмите в качестве маски число, в котором нули стоят на месте тех битов, которые вы хотите сбросить, а единицы — во всех остальных позициях, а потом используйте команду AND. Поскольку 1 AND X всегда возвращает X, мы сбросим только необходимые нам биты.

Рассмотрим несколько примеров.

Пример. В регистре AL загружен массив битов. Нужно установить все нечетные позиции в 1. Предыдущее состояние массива неизвестно.

`or al, 10101010b` ;маска устанавливает все нечетные биты в 1

Пример. В массиве битов, хранящемся в регистре AL, сбросить 0-й и 7-й биты, все остальные оставить без изменения. Исходное состояние массива также неизвестно.

`and al, 01111110b` ;каждая 1 в маске сохраняет бит
;в ее позиции

С помощью XOR также можно изменять значения битов, не зная предыдущего состояния. Для этого в маске установите 1 для каждого бита, который вы хотите инвертировать (0 станет 1, а 1 станет 0), а для всех оставшихся битов установите 0. Если мы выполним XOR дважды, то получим исходное значение. Такое поведение операции XOR позволяет использовать эту команду для простого шифрования: к каждому байту шифруемых данных применяется XOR с постоянной маской (ключом), а для дешифровки тот же ключ применяется (XOR) к зашифрованным данным.

Глава 5 Управляющие конструкции

- Последовательное выполнение команд
- Конструкция «IF THEN» — выбор пути
- Итерационные конструкции — циклы
- Команды обработки стека

Программа любой сложности на любом языке программирования может быть написана при помощи всего трех управляющих структур: линейной, условия и цикла. В этой главе мы рассмотрим эти три краеугольных камня программирования и реализацию их в языке ассемблера, а в конце главы вы узнаете о стеке, который нужен для использования подпрограмм.

5.1. Последовательное выполнение команд

Последовательная обработка знакома нам еще с концепции фон Неймана. Каждая программа состоит из одной или нескольких последовательностей отдельных элементарных команд. Последовательность здесь означает участок программы, где команды выполняются одна за другой, без любых переходов. В более широком контексте языка программирования высокого уровня можно рассматривать целую программу как последовательность, состоящую как из элементарных команд, так и из управляющих конструкций — условных и итерационных.

Если программа не содержит других конструкций, кроме последовательности элементарных команд, она называется линейной.

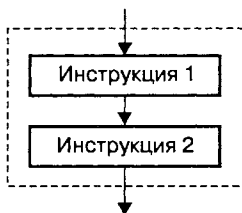


Рис. 5.1. Последовательная обработка команд

5.2. Конструкция «IF THEN» — выбор пути

В языках программирования высокого уровня конструкция выбора известна как оператор IF-THEN. Эта конструкция позволяет выбрать следующее действие из нескольких возможных вариантов в зависимости от выполнения определенного условия. В языке ассемблера механизм выбора реализован посредством команд сравнения, условного и безусловного переходов.

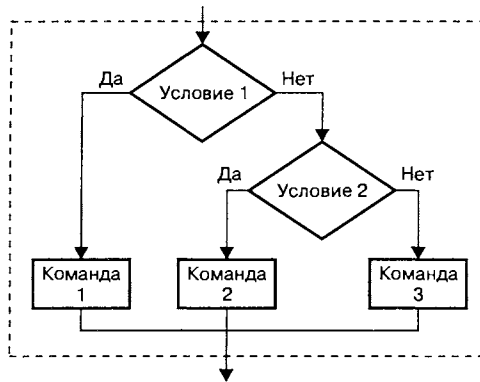


Рис. 5.2. Выбор следующей команды

5.2.1. Команды CMP и TEST

Команды CMP и TEST используются для сравнения двух операндов. Операндами могут быть как регистры, так и адреса памяти, размер операнда — 8, 16 или 32 бита.

`CMP o1, o2`

Команда CMP — это сокращение от «compare», «сравнить». Она работает подобно SUB: операнд o2 вычитается из o1. Результат нигде не сохраняется, команда просто изменяет регистр признаков. Команда CMP может использоваться как для сравнения целых беззнаковых чисел, так и для сравнения чисел со знаком.

Команда TEST работает подобно CMP, но вместо вычитания она вычисляет поразрядное И операндов. Результат инструкции — измененные флаги регистра признаков. Мы можем использовать TEST для проверки значений отдельных битов в массиве битов.

Проиллюстрируем эти команды несколькими примерами:

```

cmp ax,4           ;сравниваем AX со значением 4
cmp dl,ah          ;сравниваем DL с AH
cmp [diameter1],ax ;сравниваем переменную "diameter1" с AX
  
```

```
cmp ax,[diameter1] ;сравниваем AX с переменной "diameter1"  
cmp eax,ecx        ;сравниваем регистры EAX и ECX  
test ax,00000100b  ;проверяем значение второго  
                  ;(третьего справа) бита
```

5.2.2. Команда безусловного перехода — JMP

Самый простой способ изменить последовательность выполнения команд заключается в использовании команды **jmp** — так называемой команды безусловного перехода. Она перезаписывает указатель команд (регистр IP или CS), что заставляет процессор «переключиться» на выполнение команды по указанному адресу. Формат команды таков:

JMP [тип_перехода] операнд

Команда JMP — аналог конструкции GOTO, которая используется в высокоуровневых языках программирования. Название команды объясняет ее действие, а именно «jmp», «переход». Команде нужно передать один обязательный операнд — адрес в памяти, с которого процессор должен продолжить выполнение программы. Операнд может быть указан явно (непосредственное значение адреса) или быть регистром общего назначения, в который загружен требуемый адрес. Но новичкам я никогда не рекомендовал бы это делать: язык ассемблера, подобно языкам программирования высокого уровня, позволяет обозначить адрес назначения при помощи метки.

В зависимости от «расстояния» переходы бывают трех типов: короткие (short), ближние (near) и дальние (far). Тип перехода задается необязательным параметром инструкции jmp. Если тип не задан, по умолчанию используется тип near.

Максимальная «длина» короткого перехода (то есть максимальное расстояние между текущим и целевым адресом) ограничена. Второй байт инструкции (операнд) содержит только одно 8-разрядное значение, поэтому целевой адрес может быть в пределах от -128 до 127 байтов. При переходе выполняется знаковое расширение 8-разрядного значения и его добавление к текущему значению E(IP).

«Длина» ближнего перехода (near) зависит только от режима процессора. В реальном режиме меняется только значение IP, поэтому мы можем «путешествовать» только в пределах одного сегмента (то есть в пределах 64 Кб); в защищенном режиме используется EIP, поэтому целевой адрес может быть где угодно в пределах 4 Гб адресного пространства.

Переход типа far модифицирует кроме IP еще и сегментный регистр CS, который используется при вычислении фактического адреса памяти. Поэтому команда перехода должна содержать новое значение CS.

Сейчас мы совершим «дальний переход» от предмета нашего рассмотрения и поговорим о метках в языке ассемблера. Вкратце, метка — это идентификатор, заканчивающийся двоеточием. Во время компиляции он будет заменен точным адресом согласно его позиции в программе. Рассмотрим следующий фрагмент кода:

```
mov ax,4           ;AX = 4
new_loop:         ;метка new_loop
mov bx, ax         ;копируем AX в BX
```

Чтобы перейти к метке `new_loop` из другого места программы, используйте команду:

```
jmp new_loop      ;переходим к new_loop
```

После выполнения этой команды выполнение программы продолжится с метки `new_loop`.

Если вам нужно сначала написать инструкцию перехода и только потом определить метку, нет проблем: компилятор обрабатывает текст программы в несколько проходов и понимает такие «забегания вперед»:

```
jmp start          ;переход на start
finish:           ;метка "finish"
...
...               ;какие-то команды
...
start:            ;метка "start"
jmp finish         ;переход на "finish"
```

Теперь давайте вернемся к различным типам переходов: даже если мы новички, мы все равно будем изредка их использовать. Короткий переход полезен в ситуации, где метка назначения находится в пределах 128 байтов. Поскольку команда короткого перехода занимает 2 байта, команда ближнего перехода занимает 3 байта, а дальнего — 5 байтов, мы можем сэкономить байт или три. Если вы не можете оценить правильное «расстояние», все равно можете попробовать указать `short` — в крайнем случае, компилятор выдаст ошибку:

```
near_label:       ;метка "near_label"
...               ;несколько команд
...
jmp short near_label ;переходим к "near_label"
```

5.2.3. Условные переходы — Jx

Другой способ изменения последовательности выполнения команд заключается в использовании команды условного перехода.

В языке ассемблера имеется множество команд условного перехода, и большинство из них вам нужно знать — иначе вы не сможете написать даже

«средненькую» программку. Имена этих команд различаются в зависимости от условия перехода. Условие состоит из значений одного или нескольких флагов в регистре признаков. Работают эти команды одинаково: если условие истинно, выполняется переход на указанную метку, если нет, то процессор продолжит выполнять программу со следующей команды.

Общий формат команд условного перехода следующий:

```
Жх метка_назначения
```

Рассмотрим наиболее часто встречающиеся команды:

```

jz is_true      ;переходит к is_true, если флаг ZF = 1
jc is_true      ;переходит к is_true, если флаг CF = 1
js is_true      ;переходит к is_true, если флаг SF = 1
jo is_true      ;переходит к is_true, если флаг переполнения
                  ;OF = 1
```

Любое условие может быть инвертировано, например:

```
jnz is_true ;переходит к is_true, если флаг ZF = 0
```

Так же образованы имена команд JNC, JNS и JNO.

Рассмотрим сводную таблицу команд условного перехода в зависимости от условия, которое проверяет процессор (чтобы не писать «для перехода», будем использовать сокращение jump) (см. табл. 5.1).

Сводная таблица команд условного перехода

Таблица 5.1

	o1==o2 o1=o2	o1!=o2 o1<>o2	o1>o2	o1<o2	o1=<o2	o1>=o2
Инструкции для беззнаковых чисел	JE(JZ)	JNE(JNZ)	JA(JNBE)	JB(JNAE)	JNA(JBE)	JNB(JAE)
	Jump, если равно Jump, если 0	Jump, если не равно Jump, если не 0	Jump, если больше Jump, если не меньше или равно	Jump, если меньше Jump, если не больше или равно	Jump, если не больше Jump, если меньше или равно	Jump, если не меньше Jump, если больше или равно
Инструкции для чисел со знаком	JE(JZ)	JNE(JNZ)	JG(JNLE)	JL(JNGE)	JNG(JLE)	JNL(JGE)
	Jump, если равно Jump, если 0	Jump, если не равно Jump, если не 0	Jump, если больше Jump, если не меньше или равно	Jump, если меньше Jump, если не больше или равно	Jump, если не больше Jump, если меньше или равно	Jump, если не меньше Jump, если больше или равно

В первой строке таблицы указано условие перехода. Во второй строке показаны соответствующие команды условного перехода (в скобках — их дополнительные названия). Чтобы лучше запомнить имена команд, запомните несколько английских слов: equal — равно, above — больше, below — ниже, zero — ноль, greater — больше, less — меньше. Таким образом, JE — Jump if Equal (Переход, если Равно), JNE — Jump if Not Equal (Переход, если Не Равно), JA — Jump if Above (Переход, если больше) и т.д.

Подобно командам MUL и DIV, для работы с числами со знаком служит другой набор команд условного перехода. Причина этого в том, что проверяемое условие состоит из значений других флагов.

Адрес назначения команды условного перехода должен лежать в пределах 128 байтов: все они реализуют переходы короткого типа. Если вам нужно «прогуляться» за пределы 128 байтов, то вы должны в инструкции условного перехода указать адрес, по которому будет находиться команда **jmp**, которая и выполнит дальнейший переход:

```
jz far_jump      ; если ZF = 1, перейти к far_jump
...              ; несколько команд
far_jump:
jmp far finish   ; "дальний" переход
```

Теперь рассмотрим, как реализовать конструкцию IF-THEN на языке ассемблера. В нашем простом случае мы перейдем к метке `if_three`, если регистр AX содержит значение 3.

Прежде всего мы должны проверить, есть ли в регистре AX тройка. Для этого используем команду CMP:

```
cmp ax,3          ;сравниваем AX с 3
```

Для проверки равенства применим команду JZ, как показано в таблице команд условного перехода:

```
jz is_three      ;переходит к "is_three", если AX = 3
```

Обратите внимание, что для проверки на равенство используются одинаковые команды (JZ — равно и JNZ — не равно) для чисел со знаком и для беззнаковых чисел. Если AX = 3, то команда **jz** выполнит переход к метке `is_three`, в противном случае будет продолжено выполнение программы со следующей за **jz** команды.

Следующий пример показывает беззнаковое сравнение CL и AL. Если оба значения равны, то в регистр BL помещается значение 1, если AL больше, чем CL, то BL=2, а если AL меньше CL, то BL=3.

```
cmp al,cl          ;сравниваем AL и CL
jz write_1         ;переходим к write_1, если AL = CL
cmp al,cl          ;сравниваем AL и CL
ja write_2         ;переходим к write_2, если AL > CL
mov bl,3           ;последний случай — сразу загружаем 3 в BL
end_if:           ;просто метка, символизирующая конец IF
...
write_1:           ;метка write_1
mov bl,1           ;BL = 1
jmp end_if         ;переходим к end_if
write_2:           ;метка write_2
mov bl,2           ;BL = 2
jmp end_if         ;переходим к end_if
```

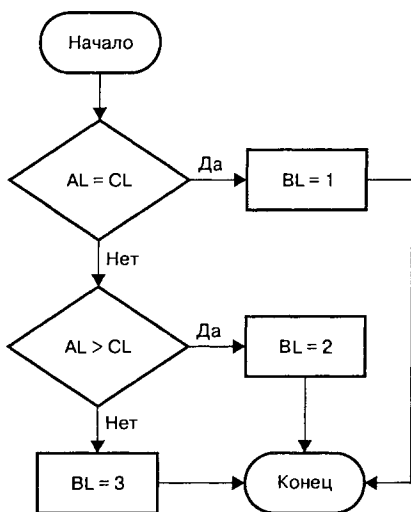


Рис. 5.3. Структурная схема нашей программы

В нашей программе мы использовали безусловный переход (`jmp end_if`), чтобы вернуть управление на исходную позицию. Это не лучшее решение: нам придется выполнить еще один безусловный переход перед меткой `write_1`, а то наша программа «зациклится». Адрес назначения понятен — следующая после последнего `jmp end_if` команда. Вот так выглядит улучшенный фрагмент кода:

```

mov bl,1           ;сразу устанавливаем BL = 1
cmp al,cl          ;сравниваем AL и CL
je end_if          ;переходим в конец программы, если AL = CL
mov bl,2           ;BL = 2
cmp al,cl          ;сравниваем AL и CL
ja end_if          ;переходим в конец программы, если AL > CL
mov bl,3           ;BL = 3
end_if:            ;конец программы
    
```

Новый пример короче, но нет предела совершенству, и мы можем его еще улучшить. Инструкция `MOV` не изменяет регистр флагов, поэтому в дальнейшем сравнении нет надобности:

```

mov bl,1           ;BL = 1
cmp al,cl          ;сравниваем AL и CL
je end_if          ;переходим в конец программы, если AL = CL
mov bl,2           ;BL = 2
ja end_if          ;переходим в конец программы, если AL > CL
mov bl,3           ;BL = 3
end_if:            ;конец программы
    
```

Если подытожить, то мы только что записали на Ассемблере следующую конструкцию языка С:

```
if (a1 == c1) b1 = 1 else if (a1 > c1) b1 = 2 else b1 = 3;
```

5.3. Итерационные конструкции — циклы

Последней управляющей конструкцией, которую мы рассмотрим, будет итерация, или цикл. **Циклом называется многократное повторение последовательности команд до наступления указанного условия.**

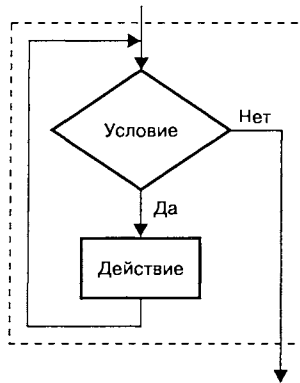


Рис. 5.4. Цикл в программе

В языках программирования высокого уровня известно много разновидностей циклов, в том числе:

- ♦ **цикл со счетчиком** (цикл FOR), повторяющийся заранее заданное количество раз;
- ♦ **цикл с условием** (цикл WHILE), повторяющийся до тех пор, пока условие истинно;
- ♦ **цикл с инверсным условием** (цикл UNTIL), повторяющийся до тех пор, пока условие *не станет* истинным.

Цикл со счетчиком с помощью конструкций IF и GOTO

Давайте попробуем написать цикл с пустым телом (то есть внутри цикла не будут выполняться никакие команды). Первое, с чем нужно разобраться — это где разместить переменную управления циклом, то есть счетчик. Счетчик нужен для того, чтобы цикл выполнялся не бесконечно, а определенное количество раз. Команда сравнения позволяет хранить счетчик либо в памяти, либо в каком-то регистре общего назначения.

Рассмотрим символическую структуру пустого цикла FOR на псевдоязыке:

```
FOR_START:           ;начало
I = 0                ;инициализация счетчика
FOR_LOOP:            ;метка цикла
...                  ;тело цикла (пустое)
I=I+1                ;увеличиваем счетчик
IF I < 10 THEN        ;проверяем счетчик
GOTO FOR_LOOP         ;переходим на начало цикла или выходим
                     ;из цикла
FOR_FINISH:          ;конец цикла
```

В нашем примере тело цикла должно повторяться 10 раз. Сначала мы инициализируем счетчик. Затем выполняем тело цикла (в нашем случае пустое), после этого увеличиваем счетчик на 1. Проверяем: если счетчик меньше 10, то начинаем опять выполнять тело цикла, если же счетчик равен 10, то мы выходим из цикла.

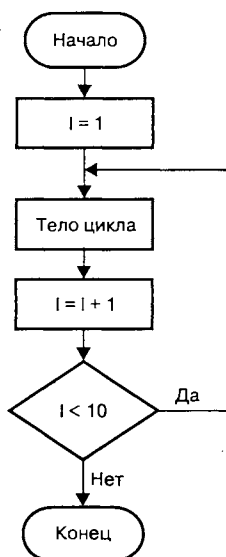


Рис. 5.5. Структурная схема цикла FOR

А теперь запишем нашу схему на языке ассемблера. Мы уже знаем, как реализовать на языке ассемблера конструкции IF и GOTO, из которых можно построить цикл FOR. В качестве счетчика (псевдопеременной I) мы будем использовать регистр ECX:

```
for_start:
mov ecx,0           ;инициализируем счетчик ECX = 0
for_loop:           ;метка для перехода назад
```

```

...                ;тело цикла
inc ecx            ;увеличиваем ECX на 1
cmp ecx,10         ;сравниваем ECX с 10
jnz for_loop       ;если не равно, переход на for_loop
for_finish:        ;если ECX = 10, выходим

```

Рассмотрим другую версию цикла FOR. Она работает так же, как предыдущая, но счетчик мы будем хранить не в регистре, а в памяти, в переменной I.

```

for_start:
mov dword [i],0    ;переменная типа dword I = 0
for_loop:          ;метка для перехода назад
...                ;тело цикла
inc dword [i]       ;увеличиваем i на 1
cmp dword [i],10    ;сравниваем i с 10
jnz for_loop        ;если не равно, переход на for_loop
for_finish:         ;если равно, выходим

```

Вторая версия будет работать медленнее, поскольку счетчик хранится в памяти, время доступа к которой существенно больше, чем время доступа к регистрам.

В заключение давайте рассмотрим еще одну версию цикла, использующую команду DEC и команду проверки флага ZF вместо команды сравнения CMP. Принцип работы следующий: устанавливаем счетчик (ECX=10), выполняем тело цикла, уменьшаем счетчик на 1. Если ZF установлен, значит, в ECX находится 0 и нам нужно прекратить выполнение цикла:

```

for_start:
mov ecx,10         ;ECX = 10
for_loop:          ;метка для перехода назад
...                ;тело цикла
dec ecx            ;уменьшаем ECX на 1
jnz for_loop       ;если не 0, переходим на for_loop
for_finish:        ;если 0, выходим из цикла

```

Мы только что записали на языке ассемблера следующую конструкцию языка C:

```
for (i=0; i < 10; i++) {}
```

LOOP — сложная команда, простая запись цикла

В главе, посвященной процессору 80386, мы упомянули, что x86-совместимые чипы используют архитектуру CISC (Компьютер со сложным набором команд), то есть имеют полную систему команд. Другими словами, в составе системы команд имеются сложные команды, которые могут заменить ряд простых. При чем здесь циклы? Если у вас CISC-процессор, то вам не нужно

реализовать цикл самостоятельно — для организации цикла можно использовать команду LOOP:

LOOP метка

Подобно команде MUL, команда LOOP работает с двумя операндами. Первый операнд фиксирован, и мы не можем его указать явно. Это значение регистра ECX (или CX). Второй — это адрес целевой метки цикла. Инструкция LOOP уменьшает значение регистра ECX (CX) на единицу и, если результат не равен 0, то она переходит на указанную метку. Метка должна быть в пределах 128 байтов (короткий тип перехода).

Перепишем наш простой цикл FOR с использованием команды LOOP:

```
for_start:
mov cx,10          ;CX = 10 — 10 итераций
for_loop:          ;метка для возврата назад
...                ;тело цикла
loop for_loop      ;уменьшаем CX, если не 0, переходим
                  ;к for_loop
for_finish:        ;выход из цикла
```

Как видите, код стал еще более компактным.

Цикл со счетчиком и дополнительным условием. Команды LOOPZ и LOOPNZ

Команда LOOPZ позволяет организовать цикл с проверкой дополнительного условия. Например, мы можем уточнить условие из предыдущего примера: цикл нужно выполнить, как и раньше, не более 10 раз, но только при условии, что регистр BX содержит значение 3. Как только значение в регистре BX изменится, цикл нужно прервать.

LOOPZ метка
LOOPNZ метка

Команда LOOPZ уточняет условие перехода следующим образом: переход на указанную метку произойдет, если CX не содержит нуля и в то же время флаг ZF равен единице. Другое имя этой команды — LOOPE.

Следующий фрагмент кода показывает пример цикла с дополнительным условием:

```
for_start:
mov cx,10          ;CX = 10
for_loop:          ;метка для возврата назад
...                ;тело цикла FOR
...                ;
...                ;где-то здесь изменяется регистр BX
...                ;
```

```

cmp bx,3          ;BX равен 3?
loopz for_loop    ;CX=CX-1; если CX<>0, и если BX=3,
                  ;переход к for_loop
for_finish:       ;если CX = 0 или если BX <> 3, выходим

```

Команда LOOPNZ работает аналогично, но дополнительное условие противоположно: переход будет выполнен только если CX (ECX) не равен 0 и в то же время ZF равен 0. Другое имя этой команды — LOOPNE.

5.4. Команды обработки стека

При программировании очень часто возникает потребность временно сохранять содержимое регистров процессора или какого-то адреса памяти, чтобы через некоторое время восстановить исходные значения. Язык ассемблера удовлетворяет эту потребность набором команд для работы со специальной областью памяти, которая называется стеком.

Что такое стек и как он работает?

Давайте разберемся, как работает стек. Все мы знаем, что такое очередь. Приходит первый клиент, пока его обслуживают, подходят еще два клиента. Когда первый клиент обслужен, начнут обслуживать второго клиента, затем третьего — и так далее. Принцип заключается в том, что первым будет обслужен тот, кто пришел первым. Такой тип очереди называется FIFO (First In — First Out) — первым пришел, первым вышел.

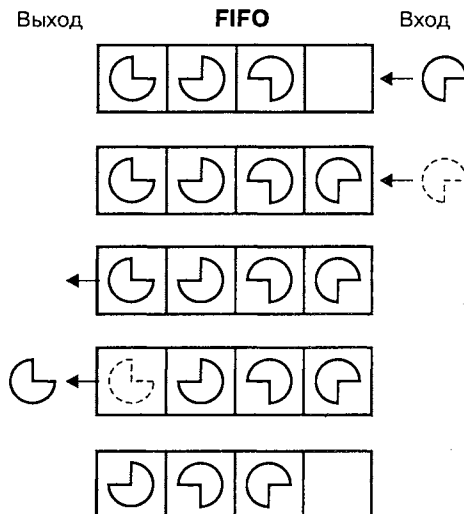


Рис. 5.6. Очередь FIFO

Пример использования:

```
push eax           ;поместить EAX в стек
```

Мы можем сами реализовать команду PUSH с помощью следующей пары команд:

```
sub esp,4          ;уменьшаем ESP на 4 (EAX — 4-байтный
                  ;регистр)
mov [ss:esp],eax   ;сохраняем EAX в стеке
```

В общем виде (с использованием оператора `sizeof`, «позаимствованного» из языков высокого уровня) команда `push o1` может быть записана на псевдо-языке так:

```
(E)SP=(E)SP-sizeof(o1)
o1 -> SS:[(E)SP]
```

Другая команда, **POP**, записывает в свой операнд значение вершины стека (последнее сохраненное в стеке значение). Тип операнда должен быть таким же, как у инструкции PUSH (другими словами, если вы поместили в стек 32-разрядный регистр, извлечение из стека должно происходить тоже в 32-разрядный регистр).

Команду POP можно реализовать с помощью команд MOV и ADD:

```
mov eax,[ss:esp]   ;помещаем в EAX вершину стека
add esp,4          ;"удаляем" последнее значение
                  ;типа dword в стеке
```

Рассмотрим несколько примеров:

```
push eax           ;сохранить значение регистра EAX в стеке
push esi           ;сохранить значение регистра ESI в стеке
...
pop eax            ;извлечь данные из стека в EAX
pop esi            ;извлечь данные из стека в ESI
```

В результате выполнения этих команд мы поменяем местами значение регистров EAX и ESI: сначала помещаем в стек значение EAX, затем — ESI, после этого извлекаем из стека последнее сохраненное значение (бывшее значение регистра ESI) в регистр EAX, после этого в стеке останется бывшее значение EAX, которое мы записываем в ESI.

Для обеспечения обратной совместимости с процессорами предыдущих поколений 16-битные регистры тоже можно поместить в стек.

```
mov ax,0x1234      ;AX = 0x1234
mov bx,0x5678      ;BX = 0x5678
push ax            ;сохранить значение регистра AX в стеке
push bx            ;сохранить значение регистра BX в стеке
...               ;изменяем значения регистров
pop bx             ;извлекаем вершину стека в BX
```

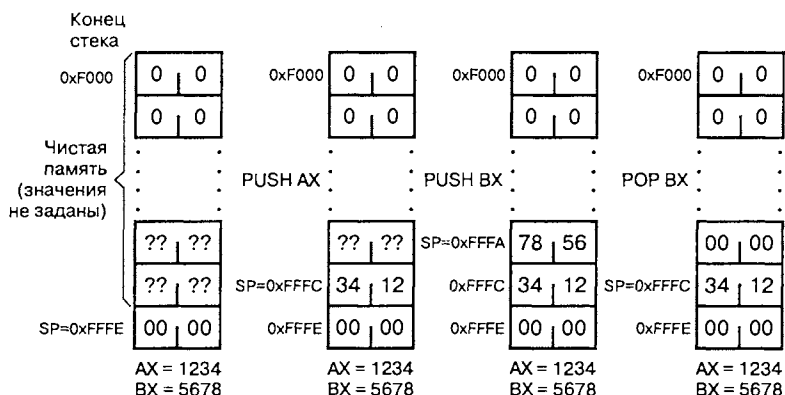


Рис. 5.8. Изменение содержимого стека

До выполнения первой команды PUSH вершина стека содержала значение 0x0000. На вершину стека указывает пара SS:SP. Допустим, что SP содержит адрес 0xFFFFE. После выполнения PUSH AX указатель стека был уменьшен на 2 и принял значение 0xFFFC, и по этому адресу (в новую вершину стека) было записано значение 0x1234. Вторая команда, PUSH BX, также уменьшила значение SP на 2 (0xFFFA) и записала в новую вершину стека значение 0x5678. Команда POP BX удалила значение 0x5678 из стека и сохранила его в регистре BX, а указатель стека увеличила на 2. Он стал равен 0xFFFC, и в вершине стека оказалось значение 0x1234.

Помните, что 8-битные регистры сохранять в стеке нельзя. Нельзя и поместить в стек регистр IP (EIP) непосредственно, при помощи команд PUSH/POP: это делается по-другому, и чуть позже вы узнаете, как именно.

Команды PUSHA/POPA и PUSHAD/POPAD: «толкаем» все регистры общего назначения

Иногда полезно сохранить в стеке значения сразу всех регистров общего назначения. Для этого используется команда PUSHA, а для извлечения из стека значений всех регистров служит команда POPA. Команды PUSHA и POPA помещают в стек и извлекают из него все 16-разрядные регистры. Операндов у этих команд нет.

Поскольку команды PUSHA и POPA разрабатывались для предшественника процессора 80386, они не могут сохранять значений 32-битных регистров (они просто не подозревают об их существовании). Для сохранения и восстановления значений расширенных регистров служат команды PUSHAD и POPAD.

Регистры помещаются в стек в следующем порядке (сверху вниз):

(E)AX, (E)CX, (E)DX, (E)BX, (E)SP, (E)BP, (E)SI, (E)DI

Рассмотрим небольшой пример:

```
pusha      ;поместить в стек все регистры общего назначения
           ;некоторые действия, модифицирующие
           ;значения регистров
```

...

```
popa      ;восстанавливаем все регистры
```

Команды PUSHF/POPF и PUSHFD/POPFD: «толкаем» регистр признаков

Рассмотренные четыре команды не заботились о помещении в стек регистра признаков. В 16-битных процессорах и регистр признаков был 16-битным, поэтому для помещения в стек флагов и восстановления из него использовались команды PUSHF и POPF. Для новых процессоров, где регистр признаков 32-битный, нужно использовать 32-битные версии этих команд — PUSHFD и POPFD.

Ни одна из рассмотренных до сих пор операций не изменяет старшие 16 битов регистра флагов, поэтому для практических целей будет достаточно команд PUSHF и POPF.

```
cmp ax,bx      ;сравниваем AX и BX
pushf          ;помещаем результат сравнения в стек
...            ;выполняем операции, изменяющие флаги
add di,4        ;например, сложение
popf           ;восстанавливаем флаги
jz equal        ;если AX = BX, переходим на "equal"
```

Команды CALL и RET: организуем подпрограмму

Ни одна серьезная программа не обходится без подпрограмм. Основное назначение подпрограмм — сокращение кода основной программы: одни и те же инструкции не нужно писать несколько раз — их можно объединить в подпрограммы и вызывать по мере необходимости.

Для вызова подпрограммы используется команда CALL, а для возврата из подпрограммы в основную программу — RET. Формат обеих команд таков:

```
CALL тип_вызова операнды
RET
```

Команде CALL нужно передать всего один операнд — адрес начала подпрограммы. Это может быть непосредственное значение, содержимое регистра, памяти или метка. В отличие от JMP, при выполнении команды CALL первым

делом сохраняется в стеке значение регистра IP (EIP). Передача управления на указанный адрес называется вызовом подпрограммы.

Как и команде JMP, команде CALL можно указать «размер шага». По умолчанию используется *near*. Когда происходит вызов типа *far*, сегментный регистр CS также сохраняется в стеке вместе с IP (EIP).

Возврат из подпрограммы выполняется с помощью команды RET, которая выталкивает из стека его вершину в IP (EIP). После этого процессор продолжит выполнение инструкций, находящихся в основной программе после команды CALL.

Если подпрограмма вызывалась по команде **CALL far**, то для возврата из нее нужно восстановить не только IP (EIP), но и CS: следует использовать команду RETF, а не RET.

Существует еще более сложный способ возврата из подпрограммы: команда RETF или RET может принимать непосредственный операнд, указывающий, сколько порций данных нужно вытолкнуть из стека вслед за IP (EIP) и CS. Этот вариант мы рассмотрим в 13 главе, когда будем говорить об объединении в одной программе фрагментов кода на ассемблере и на языках высокого уровня, а сейчас перейдем к практике.

Напишем подпрограмму, которая складывает значения EAX и EBX, а результат помещает в ECX, не обращая внимания на переполнение. Значения EAX и EBX после возвращения из подпрограммы должны сохраниться неизменными.

Назовем нашу подпрограмму `add_it`. Прежде всего мы должны получить аргументы из основной программы. В высокоуровневых языках для передачи аргументов используется стек, но мы упростим себе задачу и используем регистры. Операция ADD изменяет значение своего операнда, поэтому первым делом сохраним его в стеке:

```
add_it:
push eax           ;сохраняем значение EAX в стек
add eax,ebx        ;EAX = EAX + EBX
mov ecx,eax        ;копируем значение из EAX в ECX
pop eax            ;восстанавливаем оригинальное значение EAX
ret                ;возвращаемся
```

Теперь вызовем нашу подпрограмму `add_it` с аргументами 4 и 8:

```
mov eax,4          ;EAX = 4
mov ebx,8          ;EBX = 8
call add_it        ;вызываем add_it
                   ;Результат будет в регистре ECX
```

Что если мы забудем восстановить оригинальное значение EAX (забудем написать команду `pop eax`)? Команда RET попыталась бы передать управле-

ние адресу, заданному оригинальным значением EAX, что могло бы вызвать сбой нашей программы, а то и всей операционной системы. То же самое произойдет, если мы забудем написать команду RET: процессор продолжит выполнение с того места, где заканчивается наша подпрограмма, и рано или поздно совершит недопустимое действие.

Мы можем упростить нашу подпрограмму `add_it`, полностью отказавшись от инструкций POP и PUSH:

```
add_it:
mov ecx, eax      ; копируем значение EAX (первый параметр) в ECX
add ecx, ebx      ; добавляем к нему EBX (второй параметр),
                  ; результат будет сохранен в регистре ECX
ret               ; возвращаемся
```

Команды INT и IRET: вызываем прерывание

Вернемся к теме прерываний. Прерыванием называется такое событие, когда процессор приостанавливает нормальное выполнение программы и начинает выполнять другую программу, предназначенную для обработки прерывания. Закончив обработку прерывания, он возвращается к выполнению приостановленной программы.

Во второй главе было сказано, что все прерывания делятся на две группы: программные и аппаратные. **Программные прерывания порождаются по команде INT.** Программные прерывания можно рассматривать как «прерывания по требованию», например, когда вы вызываете подпрограмму операционной системы для вывода строки символов. В случае с программным прерыванием вы сами определяете, какое прерывание будет вызвано в тот или иной момент. Команде INT нужно передать всего один 8-битный операнд, который задает номер нужного прерывания.

INT `op`

Аппаратные прерывания вызываются аппаратными средствами компьютера, подключенными к общей шине (ISA или PCI). Устройство, запрашивающее прерывание, генерирует так называемый запрос на прерывание (IRQ, interrupt requests). Всего существует 16 аппаратных запросов на прерывание, поскольку только 16 проводников в шине ISA выделено для этой цели. Запрос на прерывание направляется контроллеру прерываний, который, в свою очередь, запрашивает микропроцессор. Вместе с запросом он передает процессору номер прерывания. После запуска компьютера и загрузки операционной системы DOS, IRQ 0 (системный таймер) соответствует прерыванию 8 (часы).

Когда процессор получает номер прерывания, он помещает в стек контекст выполняемой в данный момент программы, подобно тому, как человек кладет в книгу закладку, чтобы не забыть, с какого места продолжить чтение книги. Роль закладки играют значения CS, (E)IP и регистр флагов.

Теперь процессор будет выполнять другую программу — обработчик прерывания. Адрес этой программы называется вектором прерывания. Векторы прерывания хранятся в таблице векторов прерываний, находящейся в памяти. Таблицу прерываний можно представить себе как массив адресов подпрограмм, в котором индекс массива соответствует номеру прерывания.

После того, как процессор определит адрес обработчика прерывания, он запишет его в пару CS и (E)IP. Следующая выполненная команда будет первой командой обработчика прерывания.

В десятой главе, посвященной программированию в DOS, мы опишем функции 21-го (0x21) прерывания, которое генерируется следующей инструкцией:

```
int 0x21 ; вызов DOS
```

Возврат из обработчика прерывания осуществляется с помощью команды IRET, которая восстанавливает исходные значения (E)IP, CS и флагов из стека. Формат команды:

IRET

Давайте разберемся, как вызывается прерывание на примере 21-го прерывания (рис. 5.9). Мы считаем, что процессор работает в реальном режиме с 16-битной адресацией.



Прежде всего процессор находит номер прерывания. Программные прерывания вызываются инструкцией INT, которая содержит номер прерывания в своем операнде. Следующий шаг — сохранение контекста программы в стеке. Продемонстрируем этот процесс с помощью команд ассемблера:

```
pushf      ;сохраняем значения регистра признаков в стеке
push cs    ;сохраняем регистр CS в стеке
```

```

push ip      ;эта команда недопустима. Мы написали ее с
              ;демонстрационной целью. Реализовать ее можно так:
              ;call here
              ;here:

```

Последний шаг — это безусловный переход на адрес, прочитанный из таблицы прерываний по номеру запрошенного прерывания: `JMP far`.

Здесь мы слегка забегаем вперед: пока просто примите, что таблица векторов прерываний находится в самом начале адресуемой памяти, то есть по адресу `0x0000:0x0000`. Каждый из векторов прерывания занимает четыре байта. Первые два байта определяют новое значение IP (то есть смещение), а оставшиеся два — новое значение сегментного регистра CS. Вектор прерывания номер `0x21` хранится по адресу `0x0000:(0x21*4)`, поэтому его вызов можно записать так:

```

jmp far [0x21*4] ;эта инструкция переходит на указанный
                  ;адрес, предполагаем, что DS=0, поэтому
                  ;мы можем адресовать память с адреса
                  ;0x0000: 0x0000)

```

Команду `INT` можно реализовать самостоятельно с помощью команд `PUSHF` и `CALL far`:


```

pushf          ;сохраняем флаги в стеке
call far [0x21*4] ;теперь сохраняем CS и IP
                  ;и выполняем "jump"

```

Программные прерывания часто используются как шлюз для вызова функций операционной системы. Подробнее о них мы поговорим в главах, посвященных отдельным операционным системам.

Глава 6 Прочие команды

- 
- Изменение регистра признаков напрямую
 - Команда XCHG — меняем местами операнды
 - Команда LEA — не только вычисление адреса
 - Команды для работы со строками
 - Команды ввода/вывода (I/O)
 - Сдвиг и ротация
 - Псевдокоманды
 - Советы по использованию команд

Ассемблер на примерах.
Базовый курс

В этой главе мы рассмотрим наиболее часто используемые команды системы команд процессора x86. С помощью этих команд вы сможете написать достаточно сложные программы для решения самых разнообразных задач.

6.1. Изменение регистра признаков напрямую

Несколько команд позволяют непосредственно модифицировать флаги регистра признаков. Мы рассмотрим четыре команды, которые изменяют флаги IF и ID, то есть флаг прерывания и флаг направления.

Команды CLI и STI

Команды CLI (Clear Interrupt) и STI (Set Interrupt) сбрасывают и устанавливают флаг прерывания IF. Иными словами, при их помощи вы можете запретить или разрешить аппаратные прерывания. Если этот флаг установлен (1), аппаратные прерывания разрешены. Команда CLI сбрасывает флаг (0) — запрещает аппаратные прерывания. Потом вы должны снова разрешить их, выполнив команду STI:

```
cli      ;отключить прерывания — использовать только в DOS!  
...      ;теперь мы никем не будем потревожены и можем  
          ;выполнять какие-то действия, например, изменять  
          ;таблицу прерываний  
sti      ;включаем прерывания снова
```

Команды STD и CLD

Команды STD и CLD модифицируют значение флага DF. Этим флагом пользуется группа команд для обработки строк, поэтому подробнее о нем мы поговорим в следующей главе. Команда CLD сбрасывает этот флаг (что означает отсчет вверх), а STD устанавливает его (отсчет в обратном порядке).

Формат команд простой:

STD

CLD

6.2. Команда XCHG — меняем местами операнды

Очень часто нам нужно поменять местами значения двух регистров. Конечно, можно это сделать, используя третий, временный, регистр, но намного проще использовать инструкцию XCHG (exchange — обмен), которая как раз для этого и предназначена.

XCHG o1, o2

Подобно инструкции MOV, она имеет два операнда — o1 и o2. Каждый из них может быть 8-, 16- и 32-разрядным, и только один из них может находиться в памяти — требования такие же, как у команды MOV.

xchg eax, eax	;меняем местами значения EAX и EAX.
	;Так реализован NOP — пустой оператор
xchg ebx, ecx	;меняем местами значения EBX и ECX
xchg al, ah	;меняем местами значения AL и AH
xchg dl, ah	;меняем местами значения DL и AH
xchg byte [variable], cl	;меняем местами один байт памяти и CL

6.3. Команда LEA — не только вычисление адреса

Имя этой команды — это сокращение от «Load Effective Address», то есть «загрузить эффективный адрес». Она вычисляет эффективный адрес второго операнда и сохраняет его в первом операнде (который может быть только регистром). Синтаксис этой команды требует, чтобы второй операнд был заключен в квадратные скобки, но фактически она не адресует память.

LEA o1, [o2]

LEA полезна в тех случаях, когда мы не собираемся обращаться к памяти, а адрес нужен нам для другой цели:

lea edi, [ebx*4+ecx]	;загружает в регистр EDI адрес,
	;вычисленный как EDI = EBX*4+ECX

Значение, вычисленное командой LEA, не обязательно рассматривать как адрес: ее можно использовать для целочисленных арифметических вычислений. Несколько примеров «экзотического» использования LEA будет приведено в главе, посвященной оптимизации.

6.4. Команды для работы со строками

Строкой в языке ассемблера называется последовательность символов (байтов), заканчивающаяся нулевым байтом (точно так же, как в языке C).

0x43	0x6F	0x6D	0x70	0x75	0x74	0x65	0x72	0x00
с	о	м	п	у	т	е	р	/0

Рис. 6.1. Строка заканчивается нулевым байтом

Система команд x86-совместимых процессоров содержит несколько команд, облегчающих манипуляции со строками. Каждую из них можно заменить несколькими элементарными командами, но, как и в случае с командой LOOP, эти команды сделают вашу программу короче и понятнее.

В зависимости от размера операнда команды для обработки строк делятся на три группы. Первая группа предназначена для работы с 8-битными операндами, то есть с отдельными символами. Имена этих команд заканчиваются на «B» (byte). Имена команд второй группы, предназначенных для работы с 16-битными операндами, заканчиваются символом «W» (word). Третья группа команд работает с 32-битными операндами, и имена их заканчиваются на «D» (double word).

Все команды, обсуждаемые в этом параграфе, работают с фиксированными операндами и не имеют аргументов. Порядок отсчета байтов в строке во время работы этих команд зависит от флага направления (DF).

Команды STOSx — запись строки в память

Под единым обозначением STOSx (STOre String) скрываются три команды:

- STOSB
- STOSW
- STOSD

Команда STOSB копирует содержимое регистра AL в ячейку памяти, адрес которой находится в паре регистров ES:(E)DI, и уменьшает или увеличивает (в зависимости от флага DF) на единицу значение регистра (E)DI, чтобы подготовиться к копированию AL в следующую ячейку. Если DF=0, то (E)DI будет увеличен на 1, в противном случае — уменьшен на 1. Какой регистр будет использоваться — DI или EDI — зависит от режима процессора.

Вторая инструкция, STOSW, работает аналогично, но данные берутся из регистра AX, а (E)DI уменьшается/увеличивается на 2. STOSD копирует содержимое EAX, а E(DI) уменьшает/увеличивает на 4.

cld	; сбрасываем DF, направление будет вверх
stosw	; сохраняем AX в ES:[DI] или ES:[EDI] (зависит от режима процессора) и увеличиваем (E)DI на 2

Команды LODSx — чтение строки из памяти

Под единым обозначением LODSx (LOaD String) скрываются три команды:

- ♦ LODSB
- ♦ LODSW
- ♦ LODSD

Действие этих команд противоположно командам предыдущей группы: они копируют порцию данных из памяти в регистр AL, AX и EAX. Адрес нужной ячейки памяти берется из пары DS:(E)SI. Если флаг DF равен нулю, то регистр SI будет увеличен на 1/2/4 (B, W, D), в противном случае — уменьшен на 1/2/4.

Команды CMPSx — сравнение строк

Под единым обозначением CMPSx (CoMPare String) скрываются три команды:

- ♦ CMPSB
- ♦ CMPSW
- ♦ CMPSD

Команда CMPSB сравнивает байт, находящийся по адресу ES:(E)DI, с байтом по адресу DS:(E)SI и изменяет значения регистров SI и DI в зависимости от флага DF. Команды CMPSB и CMPSD сравнивают не байты, а слова и двойные слова соответственно, увеличивая или уменьшая регистры SI и DI на размер порции данных (2 или 4).

Команды SCASx — поиск символа в строке

Под единым обозначением SCASx (SCAn String) скрываются три команды:

- ♦ SCASB
- ♦ SCASW
- ♦ SCASD

Команды SCASB/W/D сравнивают значения регистра AL/AX/EAX со значением в памяти по адресу [ES:(E)DI]. Регистр (E)DI изменяется в зависимости от флага DF.

Команды REP и REPZ — повторение следующей команды

Команда REP (Repeat) облегчает обработку строк произвольной длины. Это так называемая префиксная команда: ее нельзя использовать отдельно,

а только в паре с какой-нибудь другой командой. Она работает подобно команде LOOP: повторяет следующую за ней команду до тех пор, пока значение в регистре (E)CX не станет равно нулю. Регистр (E)CX уменьшается на единицу при каждой итерации. Чаще всего команда REP применяется в паре с MOVS или STOS:

```
rep
movsb    ;скопировать первые (E)CX байтов из DS:(E)SI в
          ;ES:(E)DI. Это ассемблерный аналог
          ;C-функции memcpy()
```

Или:

```
rep
stosb    ;скопировать (E)CX раз значение AL в ES:(E)DI.
          ;Это ассемблерный аналог C-функции memset()
```

Команда REPZ (синоним REPE), подобно команде LOOPZ, позволяет уточнить условие. Следующая итерация выполняется тогда, когда не только (E)CX не равен нулю, но и флаг ZF не установлен. Второе условие может быть инвертировано командой REPNZ (синоним REPNE).

Эти команды часто используются вместе с SCAS или CMPS:

```
repz          ;повторяем SCASB
scasb
Или:
repz          ;повторяем CMPSB
cmpsb
```

Теперь вы уже знаете достаточно, чтобы реализовать на языке ассемблера функцию подсчета символов в строке, которая в языке C называется `strlen()`.

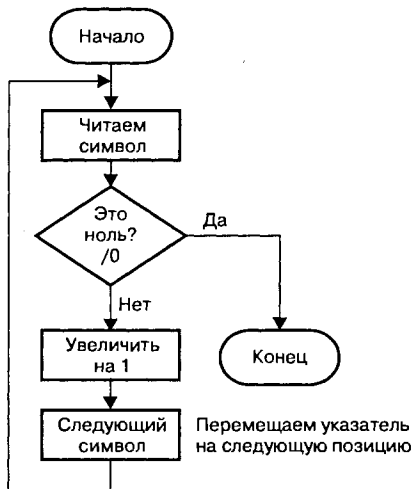


Рис. 6.2. Блок-схема функции `strlen()`

Функции нужен только один аргумент — адрес начала строки, который нужно сохранить в ES:(E)DI. Результат функции (количество символов строки + нулевой символ) будет записан в ECX. Указатель ES:(E)DI будет указывать на байт, следующий за последним (нулевым) символом строки. Код функции приведен в листинге 6.1.

Листинг 6.1. Код функции strlen()

```
strlen:
push eax          ;помещаем регистр EAX в стек
xor ecx,ecx       ;сбрасываем ECX (ECX=0), то же самое можно
                  ;сделать так: mov ecx,0
xor eax,eax       ;EAX = 0
dec ecx           ;ECX = ECX - 1. Получаем 0xFFFFFFFF -
                  ;максимальная длина строки
cld               ;DF = 0, будем двигаться вперед, а не назад
repne scasb       ;ищем нулевой байт
neg ecx           ;отрицание ECX (дополнительный код)
                  ;даст количество выполненных итераций
pop eax           ;восстанавливаем исходный EAX
ret               ;выходим из подпрограммы
```

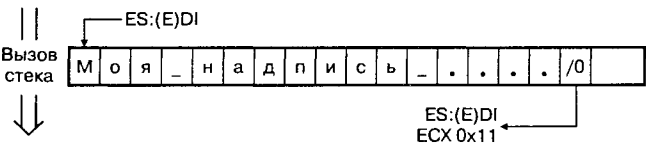


Рис. 6.3. Состояние регистров ES:(E)DI перед вызовом `strlen` и после ее выполнения

Вы можете легко адаптировать программу для работы только с 16-битными регистрами: просто удалите «Е» в имени каждого регистра.

Мы уже написали функцию для подсчета символов в строке, но до сих пор не знаем, как разместить строку в памяти. Пока будем считать, что в ES:(E)DI уже загружен правильный адрес начала строки. Мы вызываем нашу подпрограмму с помощью `CALL`, а после ее выполнения в регистре ECX получим количество символов в строке.

```
call strlen      ;вызываем strlen
```

Еще одна очень нужная функция — это функция сравнения двух строк, которая в языке С называется `strcmp()`.

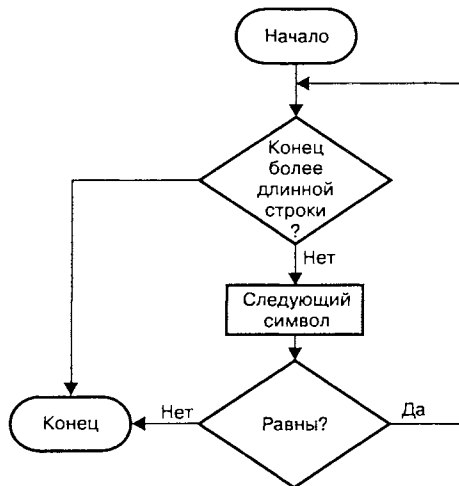


Рис. 6.4. Блок-схема функции `strcmp`

Подпрограмма сравнивает две строки: адрес первой сохранен в `ES:(E)DI`, а адрес второй — в `DS:(E)SI`. Если строки одинаковы, то после завершения подпрограммы `ECX` будет содержать 0, а если нет, то в `ECX` будет количество первых одинаковых символов. Код нашей `strcmp()` приведен в листинге 6.2.

Листинг 6.2. Код функции `strcmp()`

```

strcmp:
push edx           ; помещаем EDX в стек
push edi           ; помещаем EDI в стек
call strlen        ; вычисляем длину первой строки
mov edx,ecx        ; сохраняем длину первой строки в EDX
mov edi,esi        ; EDI = ESI
; push ds          ; сохраняем DS в стек
; push ds          ; еще раз
; pop es           ; а теперь загружаем его в ES (ES = DS)
;
call strlen        ; теперь вычисляем длину второй строки
; pop ds           ; восстанавливаем исходный DS
cmp ecx,edx        ; какая строка длиннее?
jae .length_ok     ; переходим, если длина первой строки (ECX)
                   ; больше или равна длине второй

```

```

mov ecx,edx      ;будем использовать более длинную строку
.length_ok:
pop edi          ;восстанавливаем исходный EDI
cld              ;DF = 0
repe cmpsb       ;повторяем сравнение, пока не встретим два
                  ;разных символа или пока ECX не станет
                  ;равен 0
pop edx          ;восстанавливаем исходное значение EDX
ret              ;конец
    
```

Наша функция **strcmp** будет работать только тогда, когда значения сегментных регистров DS и ES равны. Корректнее было бы раскомментировать закоментированные команды — наша функция смогла бы работать со строками, расположенными в разных сегментах. Однако в большинстве программ сегментные регистры равны, поэтому нам не нужно учитывать их разницу.

На рис. 6.5 вы можете видеть содержимое регистров после сравнения неодинаковых строк.

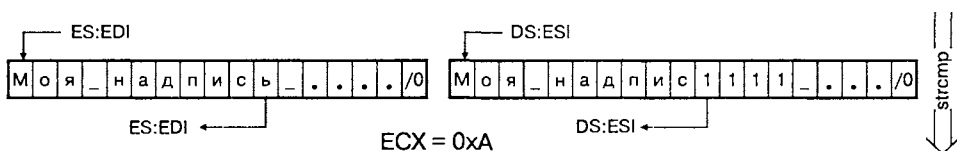


Рис. 6.5. Содержимое регистров до и после вызова `strcmp`

6.5. Команды ввода/вывода (I/O)

Периферийные устройства используют так называемые шлюзы ввода/вывода — обычно их называют портами ввода/вывода. С помощью портов ваша программа (или сам процессор) может «общаться» с тем или иным периферийным устройством. Для «общения» с периферийным или другим устройством на аппаратном уровне используются команды **IN** и **OUT**.

Команды **IN** и **OUT** — обмен данными с периферией

Команда **IN** позволяет получить от устройства, а **OUT** — передать устройству порцию данных в размере байта, слова или двойного слова.

<code>IN al, dx</code>	<code>OUT dx, al</code>
<code>IN ax, dx</code>	<code>OUT dx, ax</code>
<code>IN eax, dx</code>	<code>OUT dx, eax</code>
<code>IN al, imm8</code>	<code>OUT imm8, al</code>
<code>IN ax, imm8</code>	<code>OUT imm8, ax</code>

Команда IN читает данные из порта ввода/вывода, номер которого содержится в регистре DX, и помещает результат в регистр AL/AX/EAX. Другие регистры, кроме AL/AX/EAX и DX, использовать нельзя.

Команда OUT отправляет данные в порт. Типы ее операндов такие же, как у IN, но обратите внимание: операнды указываются в обратном порядке.

Диапазоны портов ввода/вывода, которые связаны с различными устройствами, перечислены в табл. 6.1.

Диапазоны портов ввода/вывода

Таблица 6.1

0000–001f : dma1	Первый контроллер DMA (Direct Memory Access)
0020–003f : pic1	Первый аппаратный контроллер прерываний
0040–005f : timer	Системный таймер
0060–006f : keyboard	Клавиатура
0070–007f : rtc	Часы реального времени (RTC, real time clock)
0080–008f : dma page reg	DMA page register
00a0–00bf : pic2	Второй аппаратный контроллер прерываний
00c0–00df : dma2	Второй DMA-контроллер
00f0–00ff : fpu	Математический сопроцессор
0170–0177 : ide1	Второй IDE-контроллер (Secondary)
01f0–01f7 : ide0	Первый IDE-контроллер (Primary)
0213–0213 : isapnp read	Интерфейс PnP (plug-and-play) шины ISA
0220–022f : soundblaster	Звуковая плата
0290–0297 : w83781d	Аппаратный мониторинг температуры и напряжения
0376–0376 : ide1	Второй IDE-контроллер (продолжение)
03c0–03df : vga+	Видеоадаптер
03f2–03f5 : floppy	Дискковод для гибких дисков
03f6–03f6 : ide0	Первый IDE-контроллер (продолжение)
03f7–03f7 : floppy DIR	Дискковод для гибких дисков (продолжение)
03f8–03ff : lirc_serial	Последовательный порт
0a79–0a79 : isapnp write	Интерфейс PnP (plug-and-play) шины ISA (продолжение)
0cf8–0cff : PCI conf1	Первый конфигурационный регистр шины PCI
4000–403f : Intel Corp. 82371AB/EB/MB PIIX4 ACPI	Чипсет ACPI
5000–501f : Intel Corp. 82371AB/EB/MB PIIX4 ACPI	Чипсет ACPI
e000–e01f : Intel Corp. 82371AB/EB/MB PIIX4 USB	Чипсет USB
f000–f00f : Intel Corp. 82371AB/EB/MB PIIX4 IDE	Чипсет контроллера дисков

Рассмотрение функций отдельных регистров периферийных устройств выходит далеко за рамки этой книги. Вот просто пример:

```
in al,0x60      ;чтение значения из порта с номером 0x60
                ; (это скан-код последней нажатой клавиши)
```

Организация задержки. Команда NOP

Название этой команды звучит как «No Operation», то есть это ничего не делающая, «пустая» инструкция.

```
nop            ;ничего не делать
```

Для чего же нужна такая команда? Когда программа работает с портами ввода/вывода, между операциями чтения и записи нужно делать паузы. Причина ясна: периферийные устройства работают медленнее процессора, и им нужно время, чтобы обработать запрос. Для организации задержки и используют NOP. Выполнение этой команды занимает один цикл процессора — маловато, поэтому чаще задержку организуют при помощи безусловного перехода на следующую команду:

```
jmp short delay1
delay1:
```

Поскольку команда короткого перехода занимает 2 байта, тот же переход на следующую команду можно записать так:

```
jmp short $+2      ;перейти на 2 байта вперед
```

Знак доллара означает текущий адрес.

Новые процессоры снабжены схемой предсказания переходов, то есть еще до самого перехода процессор будет знать, что ему нужно сделать. Предсказанный переход будет выполнен за меньшее число циклов процессора, чем непредсказанный, поэтому на новых процессорах задержку с помощью команды JMP организовать нельзя. Лучшим способом «подождать» периферийное устройство будет запись произвольного значения в «безопасный» порт — это диагностический порт с номером 0x80:

```
out 0x80,al      ; это просто задержка
```

6.6. Сдвиг и ротация

Операции поразрядного сдвига перемещают отдельные биты в байте, слове или двойном слове. Сдвиг может быть влево или вправо. При сдвиге вправо на одну позицию крайний правый (младший) бит теряется, а крайний левый замещается нулем. При сдвиге влево теряется крайний левый (старший) бит, а крайний правый замещается нулем. При сдвиге на несколько позиций «выталкивается» указанное количество битов, а «освободившиеся» биты замещаются нулями.

Ротация (также называемая циклическим сдвигом) не выталкивает биты, а возвращает их на место «освободившихся». В результате ротации вправо крайний правый бит встанет на место крайнего левого, а остальные биты будут сдвинуты на одну позицию вправо.

Где это может использоваться, вы поймете из описания самих команд.

Команды SHR и SHL — сдвиг беззнаковых чисел

Команды SHR и SHL поразрядно сдвигают беззнаковые целые числа вправо и влево соответственно. Это самый быстрый способ умножить или разделить целое число на степень двойки.

Представим число 5 в двоичной системе — 0101b. После умножения на 2 мы получим 10, в двоичной системе это число 01010b. Сравнивая два двоичных числа, вы, наверное, заметили, как можно быстро получить из числа 5 число 10: с помощью сдвига на один бит влево, добавив один ноль справа.

Точно так же можно умножить число на любую степень двух. Например, вместо умножения на 16 (2 в степени 4) можно сдвинуть исходное число на 4 бита влево.

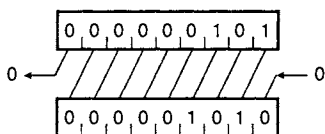


Рис. 6.6. Умножение 5 на 2 методом поразрядного сдвига влево

Деление на степень двойки выполняется по такому же принципу, только вместо поразрядного сдвига влево нужно использовать сдвиг вправо.

Команде SHL нужно передать два операнда:

SHL o1, o2

Первый должен быть регистром или адресом памяти, который нужно сдвинуть. Второй операнд определяет число позиций, на которое нужно сдвинуть. Чаще всего это непосредственное значение. Можно использовать в качестве второго операнда и регистр, но только CL — это касается всех операций сдвига и ротации. Сдвиг возможен не более чем на 32 позиции, поэтому принимается в расчет не весь второй операнд, а остаток от его деления на 32.

Старший «вытолкнутый» бит сохраняется в флаге переноса CF, а младший бит заменяется нулем. Кроме флага CF используется флаг знака (SF) и флаг

переполнения (OF). За этими флагами нужно следить, выполняя действия над числами со знаком, чтобы избежать превращения положительного числа в отрицательное и наоборот (в этом случае флаги SF и OF устанавливаются в 1).

То же самое, что и SHL, только биты сдвигаются вправо:

```
SHR 01, 02
```

Младший бит перемещается в CF, а старший заменяется нулем. Принцип работы SHR показан на рис. 6.7.



Рис. 6.7. Как работает SHR

А теперь рассмотрим несколько примеров, которые помогут вам лучше усвоить прочитанное.

Пример: используя SHR, разделим AX на 16, не обращая внимания на переполнение:

```
shr ax,4           ;сдвигаем AX на 4 бита вправо
```

Пример: подсчитаем количество двоичных единиц в AX и сохраним результат в BL.

AX — 16-разрядный регистр, поэтому мы должны сдвинуть его влево или вправо 16 раз. После каждого сдвига мы анализируем флаг переноса CF, в который попал вытолкнутый бит, что очень легко сделать с помощью инструкции JC. Если CF равен единице, то увеличиваем BL.

```
mov bl,0           ;инициализируем счетчик единиц BL=0
mov cx,16          ;CX = 16
repeat:
shr ax,1           ;сдвигаем на 1 бит вправо, младший бит
                  ;попадает в CF
jnc not_one        ;если это 0, пропускаем следующую команду
inc bl             ;иначе — увеличиваем BL на 1
not_one:
loop repeat        ;повторить 16 раз
```

После выполнения этого фрагмента кода регистр BL будет содержать количество единиц регистра AX, а сам регистр AX будет содержать 0.

Команды SAL и SAR — сдвиг чисел со знаком

Команды SAL и SAR используются для поразрядного сдвига целых чисел со знаком (арифметического сдвига). Команда SAL — это сдвиг влево, а команда SAR — вправо.

Формат команд таков:

SAL o1, o2

SAR o1, o2

Команда SAR сдвигает все биты, кроме старшего, означающего знак числа — этот бит сохраняется. Младший бит, как обычно, вытесняется в CF. Операнды обеих инструкций такие же, как у SHL и SHR.



Рис. 6.8. Как работает SAR

Команды RCR и RCL — ротация через флаг переноса

Эти команды выполняют циклический поразрядный сдвиг (ротацию). RCR действует точно так же, как SHR, но вместо нуля в старший бит первого операнда заносится предыдущее содержимое CF. Исходный младший бит вытесняется в CF. Команда RCL работает подобно RCR, только в обратном направлении.

Формат команд таков:

RCR o1, o2

RCL o1, o2

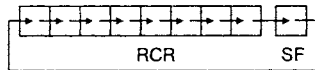


Рис. 6.9. Как работает RCR

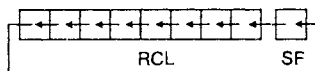


Рис. 6.10. Как работает RCL

Команды ROR и ROL — ротация с выносом во флаг переноса

Эти команды выполняют другой вариант циклического сдвига: ROR сначала копирует младший бит первого операнда в его старший бит, а потом заносит его в CF; ROL работает в обратном направлении.

```
ROR 01, 02
ROL 01, 02
```

Операнды этих команд аналогичны операндам команд RCR и RCL.

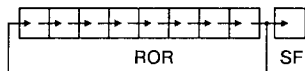


Рис. 6.11. Как работает ROR

6.7. Псевдокоманды

Некоторые из команд, рассмотренных нами, могут работать с операндом, расположенным в памяти. Классический пример — команда `MOV AX, [number]`, загружающая в регистр AX значение из области памяти, адрес которой представлен символическим обозначением «number». Но мы до сих пор не знаем, как связать символическое обозначение и реальный адрес в памяти. Как раз для этого и служат псевдокоманды.

Предложения языка ассемблера делятся на команды и псевдокоманды (директивы). Команды ассемблера — это символические имена машинных команд, обработка их компилятором приводит к генерации машинного кода. **Псевдокоманды же управляют работой самого компилятора.** На одной и той же аппаратной архитектуре могут работать различные ассемблеры: их команды обязательно будут одинаковыми, но псевдокоманды могут быть разными.

Псевдокоманды DB, DW и DD — определение констант

Чаще всего используется псевдокоманда `DB` (define byte), позволяющая определить числовые константы и строки. Рассмотрим несколько примеров:

```
db 0x55                ;один байт в шестнадцатеричном виде
db 0x55,0x56,0x57      ;три последовательных байта: 0x55,
                        ;0x56, 0x57
db 'a',0x55            ;можно записать символ в одинарных
                        ;кавычках.
                        ;Получится последовательность 0x61, 0x55
db 'Hello',13,10,'$'   ;можно записать целую строку.
                        ;Получится 0x48, 0x65, 0x6C, 0x6C,
                        ;0x6F, 0xD, 0xA, 0x24
```

Для определения порции данных размера, кратного слову, служит директива DW (define word):

```
dw 0x1234      ;0x34, 0x12
dw 'a'         ;0x61, 0x00: второй байт заполняется нулями
```

Директива DD (define double word) задает значение порции данных размера, кратного двойному слову:

```
dd 0x12345678   ;0x78 0x56 0x34 0x12
dd 1.234567e20  ;так определяются числа с плавающей точкой
```

А вот так определяется переменная, тот самый «number»:

```
number dd 0x1    ;переменная number инициализирована
               ;значением 1
```

Переменная «number» теперь представляет адрес в памяти, по которому записано значение 0x00000001 длиной в двойное слово.

Псевдокоманды RESB, RESW и RESD — объявление переменных

Вторая группа псевдокоманд позволяет определить неинициализированные данные. Если в первом случае мы заносим данные в память, то сейчас мы просто укажем, сколько байтов нужно зарезервировать для будущих данных. Память для этих неинициализированных переменных распределяется динамически, а сами данные не содержатся в исполнимом файле. Грубо говоря, предыдущую группу инструкций удобно использовать для определения констант, а эту группу — для переменных.

Для резервирования памяти служат три директивы: RESB (резервирует байт), RESW (резервирует слово) и RESD (резервирует двойное слово). Аргументом этих псевдокоманд является количество резервируемых позиций:

```
resb 1          ;резервирует 1 байт
resb 2          ;резервирует 2 байта
resw 2          ;резервирует 4 байта (2 слова)
resd 1          ;резервирует 4 байта
number resd 1   ;резервирует 4 байта для переменной
               ;"number"
buffer resb 64  ;резервирует 64 байта для
               ;переменной buffer
```

Некоторые ассемблеры поддерживают дополнительные директивы выделения памяти, но мы будем использовать ассемблер NASM, который их не поддерживает. Поэтому мы ограничимся директивами RESx.

Псевдокоманда TIMES — повторение следующей псевдокоманды

Директива TIMES — это псевдокоманда префиксного типа, то есть она используется только в паре с другой командой. Она повторяет последующую

псевдокоманду указанное количество раз, подобно директиве DUP из ассемблера Borland TASM. Применяется эта директива в тех случаях, когда нужно «забить» некоторую область памяти повторяющимся образцом.

Следующий код определяет строку, состоящую из 64 повторяющихся «Hello»:

```
many_hello: times 64 db 'Hello'
```

Первый аргумент, указывающий количество повторений, может быть и выражением. Например, задачу «разместить строку в области памяти размером в 32 байта и заполнить пробелами оставшееся место» легко решить с помощью директивы TIMES:

```
buffer db "Hello"           ;определяем строку
times 32-($-buffer) db ' '   ;определяем нужное
                               ;количество пробелов
```

Выражение 32-(\$-buffer) возвратит значение 27, потому что \$-buffer равно текущей позиции минус позиция начала строки, то есть 5.

Вместе с TIMES можно использовать не только псевдокоманды, но и команды процессора:

```
times 5 inc eax      ;5 раз выполнить INC EAX
```

В результате будет сгенерирован код:

```
inc eax
inc eax
inc eax
inc eax
inc eax
```

Псевдокоманда INCBIN — подключение двоичного файла

Эта директива будет полезна более опытным разработчикам. Она упаковывает графические или звуковые данные вместе с исполняемым файлом:

```
incbin "sound.wav"           ;упаковываем весь файл
incbin "sound.wav",512       ;пропускаем первые 512 байтов
incbin "sound.wav",512,80    ;пропускаем первые 512 байтов и
                               ;последние 80
```

Псевдокоманда EQU — вычисление константных выражений

Эта директива определяет константу, известную во время компиляции. В качестве значения константы можно указывать также константное выражение. Директиве EQU должно предшествовать символическое имя:

```
four EQU 4                  ;тривиальный пример.
                             ;Позже я покажу и нетривиальные
```

Оператор SEG — смена сегмента

При создании больших программ для реального режима процессора нам нужно использовать для кода и данных несколько сегментов, чтобы обойти проблему 16-битной адресации. Пока будем считать, что сегмент — это часть адреса переменной.

С помощью оператора SEG в сегментный регистр может быть загружен адрес сегмента, где физически расположена переменная:

```
mov ax,seg counter ;поместить в AX адрес сегмента, где
                   ;размещена переменная counter
mov es,ax          ;поместить этот адрес в сегментный
                   ;регистр.
                   ;это можно сделать только косвенно
mov bx,counter     ;загрузить в BX адрес (смещение)
                   ;переменной counter. Теперь пара ES:BX
                   ;содержит полный адрес переменной counter
mov cx,es:[bx]     ;копировать значение переменной
                   ;в регистр CX
```

В наших учебных программах на сегментные регистры можно не обращать внимания, потому что они содержат одно и то же значение. Поэтому оператор SEG нам пока не понадобится.

6.8. Советы по использованию команд

Теперь, когда вы уже познакомились с основными командами, самое время поговорить об оптимизации кода программы.

Сегодня, когда компьютеры очень быстры, а память стоит очень дешево, почти никто не считается ни со скоростью работы программы, ни с объемом занимаемой памяти. Но язык ассемблера часто применяется в особых ситуациях — в «узких местах», где быстроедействие или малый размер программы особенно важны. Удачным подбором команд можно существенно сократить размер программы или увеличить ее быстроедействие — правда, при этом страдает удобочитаемость. Ускорить работу программы иногда можно также путем правильного выделения памяти.

Директива ALIGN — выравнивание данных в памяти

Мы знаем, что процессоры работают с регистрами на порядок быстрее, чем с операндами, расположенными в памяти. Поэтому желательно выполнять все вычисления в регистрах, сохраняя в памяти только результат.

При обработке больших массивов частые обращения к памяти неизбежны. Скорость доступа к памяти можно увеличить, если размещать данные по адресам, кратным степени двойки. Дело в том, что между памятью и про-

цессором имеется система буферов памяти, недоступная программисту. Эти буферы содержат блоки чаще всего используемых данных из основной памяти или несохраненные данные. «Выравнивание» данных по некоторым адресам «освобождает» буферы, поэтому данные обрабатываются быстрее.

К сожалению, каждый тип процессора предпочитает свой тип выравнивания. Мы можем сообщить компилятору требуемый тип директивой `ALIGN`. Ее аргумент — то число, по адресам, кратным которому, требуется размещать данные:

```
align 4      ;размещает данные по адресам, кратным 4
align 16     ;размещает данные по адресам, кратным 16
```

Загрузка значения в регистр

Много обращений к памяти можно исключить, если отказаться от непосредственных операндов, что положительно скажется на быстродействии программы. Например, как мы привыкли инициализировать счетчик? Командой `MOV`, которая копирует в регистр значение 0? Намного рациональнее для этого использовать логическую функцию `XOR` (если ее аргументы одинаковы, результат равен 0):

```
xor eax,eax      ;в машинном коде 0x33,0xC0
```

Эта команда будет выполнена быстрее и займет меньше памяти, чем

```
mov eax,0        ;в машинном коде 0xB8,0,0,0,0
```

Но нужно помнить, что `XOR` изменяет регистр признаков, поэтому ее нужно использовать только в тех случаях, когда его изменение не имеет значения.

Другой часто используемой парой инструкций являются:

```
xor eax,eax      ;EAX = 0
inc eax          ;увеличиваем на 1
```

Этот фрагмент кода загружает в `EAX` значение 1. Если использовать `DEC` вместо `INC`, в `EAX` будет значение -1.

Оптимизируем арифметику

Если вам нужно добавить к значению константу, то имейте в виду, что несколько команд `INC` будут выполнены быстрее, чем одна `ADD`. Таким образом, вместо команды

```
add eax,4        ;добавляем 4 к EAX
```

следует использовать четыре команды:

```
inc eax          ;добавляем 1 к EAX
inc eax
inc eax
inc eax
```

Помните, что ни INC, ни DEC не устанавливают флаг переноса, поэтому не используйте их в 64-разрядной арифметике, где используется перенос. Зато при вычислении значения адреса вы можете смело использовать INC и DEC, поскольку арифметические операции над адресами никогда не требуют переноса.

Вы уже знаете, что для быстрого умножения и деления можно использовать поразрядные сдвиги влево и право. Умножение можно выполнять также с помощью команды LEA, которая вычисляет эффективный адрес второго операнда. Вот несколько примеров:

```
lea ebx,[ecx+edx*4+0x500] ;загружаем в EBX результат
                           ;выражения ECX + EDX*4 + 0x500
lea ebx,[eax+eax*4-1]     ;EBX = EAX*5 - 1
lea ebx,[eax+eax*8]       ;EBX = EAX*9
lea ecx,[eax+ebx]         ;вычисляем ECX = EAX + EBX
```

Операции сравнения

Очень часто нам нужно проверить какой-то регистр на наличие в нем 0. Совершенно не обязательно использовать для этого CMP, можно использовать OR или TEST. Таким образом, вместо команд

```
cmp eax,0                 ;EAX равен 0?
jz is_zero                ;да? Переходим к is_zero
```

можно написать следующее:

```
or eax,eax                ;этот OR устанавливает тот же
                           ;флаг (ZF), если результат равен 0
jz is_zero                ;да? Переходим к is_zero
```

Если оба операнда команды OR имеют одно и то же значение, первый операнд сохраняется. Кроме того, команда изменяет регистр признаков: если результат операции OR нулевой, то нулевой признак (ZF) будет установлен в 1.

Любая арифметическая команда устанавливает флаг нуля, поэтому совсем не обязательно использовать CMP для проверки на 0. Можно сразу использовать jz:

```
dec eax
jz now_zero ;переход, если EAX после декремента равен 0
```

Иногда нужно узнать, содержит ли регистр отрицательное число. Для этого можно использовать команду TEST, которая вычисляет логическое И, но не сохраняет результат, а только устанавливает флаги регистра признаков. Флаг знака SF будет установлен в 1, если результат отрицательный, то есть если старший бит результата равен 1. Значит, мы можем выполнить TEST с двумя одинаковыми операндами: если число отрицательное, то флаг SF будет установлен (логическое И для двух отрицательных чисел даст 1 ($1 \text{ AND } 1 = 1$)) в старшем бите, то есть $SF=1$):


```
test eax,eax      ;вызываем TEST для двух одинаковых  
                  ;операндов  
js is_negative    ;переходим, если SF=1
```

Разное

Обычно вместо нескольких простых команд проще использовать одну сложную (например, LOOP или команды для манипуляций со строками). Очень часто такой подход и правильнее: сложные команды оптимизированы и могут выполнить ту же самую функцию быстрее, чем множество простых команд.

Размер исполняемого файла может быть уменьшен, если оптимизировать переходы. По умолчанию используется «средний шаг» — near, но если все цели находятся в пределах 128 байтов, используйте короткий тип перехода (short). Это позволит сэкономить один-три байта на каждом переходе.

Глава 7 Полезные фрагменты кода

- Простые примеры
- Преобразование числа в строку
- Преобразование строки в число

В этой главе мы представим несколько подпрограмм, которые можно использовать в большинстве ваших ассемблерных программ. Мы будем рассматривать только системно-независимый код, одинаково пригодный для любой операционной системы. Программы, использующие особенности операционных систем, будут рассмотрены в следующих главах.

7.1. Простые примеры

Начнем с нескольких простых примеров, которые помогут уменьшить «пропасть» между языком высокого уровня и ассемблером.

Сложение двух переменных

Задача: сложить два 32-разрядных числа, содержащихся в переменных `number1` и `number2`, а результат сохранить в переменной `result`.

Сначала нужно выбрать регистры, куда будут загружены нужные значения. Затем нужно сложить эти регистры, а результат записать в переменную `result`:

```
mov eax,[number1] ;квадратные скобки означают доступ к памяти
mov ebx,[number2] ;EBX = number2
add eax,ebx       ;EAX = EAX + EBX
mov [result],eax  ;сохраняем результат в переменной result
...
number1 dd 8      ;определяем переменную number1 и
                  ;инициализируем 8
number2 dd 2      ;переменную number2 инициализируем
                  ;значением 2
result dd 0       ;определяем переменную result
```

Мы можем переписать программу, используя «на подхвате» регистр `EAX`:

```
mov eax,[number1] ;EAX = "number1"
add eax,[number2] ;EAX = EAX + number2
mov [result],eax  ;сохраняем результат в переменной result
```

Сложение двух элементов массива

Задача: сложить два 32-битных числа. Регистр EDI содержит указатель на первое слагаемое, второе слагаемое расположено непосредственно за первым. Результат записать в EDX.

Фактически мы имеем массив из двух 32-битных чисел, адрес которого задан указателем в регистре EDI. Каждый элемент массива занимает 4 байта, следовательно, второй элемент расположен по адресу, на 4 байта большему.

```
mov edx,[edi]      ;загружаем в EDX первый элемент массива
add edx,[edi+4]    ;складываем его со вторым, результат – в EDX
```

Дополним программу загрузкой регистра EDI:

```
mov edi,numbers    ;загружаем в EDI адрес массива numbers
...                ;какие-то команды
mov edx,[edi]      ;загружаем в EDX первый элемент массива
add edx,[edi+4]    ;прибавляем второй элемент
...
numbers dd 1       ;массив numbers инициализируется
                  ;значениями 1 и 2, поэтому результат
                  ;в EDX будет равен 3
dd 2               ;у второго элемента массива нет особого
                  ;имени
```

В следующем пункте мы расскажем, как работать с массивами в цикле.

Суммируем элементы массива

Задача: вычислить сумму массива 8-разрядных чисел, если в ESI загружен адрес первого элемента массива. Массив заканчивается нулевым байтом.

Сумма 8-разрядных чисел может оказаться числом большей разрядности, поэтому на всякий случай отведем для хранения результата 32-битный регистр. Элементы массива будут суммироваться до тех пор, пока не будет встречен нулевой байт.

```
mov esi,array      ;загружаем в ESI начало массива
mov ebx,0          ;EBX = 0
mov eax,ebx        ;EAX = 0
again:
mov al,[esi]       ;загружаем в AL элемент массива
inc esi            ;перемещаем указатель на след. элемент
add ebx,eax        ;EBX = EBX + EAX
cmp al,0           ;AL равен нулю?
jnz again          ;переходим к again, если AL не 0
array db 1,2,3,4,5,6,7,8,0 ;инициализируем массив.
                        ;Сумма (EBX) должна быть равна 36
```

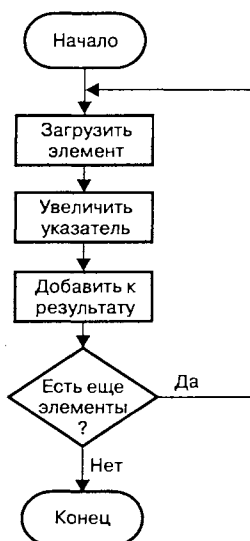


Рис. 7.1. Блок-схема алгоритма суммирования массива

Чет и нечет

Задача: определить, четное или нечетное значение содержит регистр AX.

Четное число отличается от нечетного тем, что его младший бит равен нулю. Используя SHR, мы можем сдвинуть этот бит в CF, а затем проверить этот бит, выполнив условный переход.

```

push ax      ; сохраняем исходное значение AX в стеке
shr ax,1     ; перемещаем младший бит AX в CF
pop ax       ; восстанавливаем оригинальное значение
jc odd       ; если CF = 1, переходим к odd
even:        ; действия, если число в AX — четное
...
odd:         ; действия, если число в AX — нечетное
    
```

Как обычно, мы можем переписать этот фрагмент кода проще:

```

test al,1    ; младший бит маски содержит 1, вызываем TEST
jz even      ; ZF (флаг нуля) установлен, если результат test
              ; равен 0, то есть младший бит = 0, то есть число
              ; четное

odd:
...
even:        ; действия, если AX — четное
    
```

Обратите внимание, что мы тестировали только AL, а не весь регистр AX. Старшие биты AX для проверки четности совершенно не нужны.

Перестановка битов в числе

Задача: реверсируем порядок битов числа, сохраненного в AL, то есть переставим младший бит на место старшего, второй справа — на место второго слева и т.д. Полученный результат сохраним в AH.

Например, наше число равно 0x15, то есть 00010101b. После реверсирования мы получим его «зеркальное отображение»: 10101000b, то есть 0xA8.

Как обычно, поставленную задачу можно решить несколькими способами. Можно «пройтись» по всем битам AL так, как показано в параграфе о битовых массивах, проверяя значение отдельных битов и устанавливая в соответствии с ним значения отдельных битов AH, но это не лучшее решение.

Оптимальным в нашем случае будет использование инструкций сдвига и ротации. Например, используя SHR (как в предыдущем примере), мы можем «вытолкнуть» один бит в CF (флаг переноса) и, используя RCL, переместить этот бит в младший бит операнда. Повторив это действие 8 раз, мы решим поставленную задачу.

```
mov cx,8           ;наш счетчик CX = 8
theloop:
shr al,1           ;сдвигаем AL на 1 бит вправо, младший бит —
                   ;в CF
rcl ah,1           ;сдвигаем AH на 1 бит влево, заменяем
                   ;младший бит на CF
loop theloop       ;повторяем 8 раз
```

Проверка делимости числа нацело

Задача: определить, заканчивается ли десятичная запись числа цифрой нуль.

Простого сравнения битов здесь недостаточно, мы должны разделить число на 10 (0xA). Операция целочисленного деления помещает в регистр AL частное, а в регистр AH — остаток. Нам останется только сравнить остаток с нулем: если число делится нацело, то передадим управление на метку YES.

Для определенности считаем, что наше число находится в регистре AX:

```
mov bl,0xA         ;BL = 10 — делитель
div bl             ;делим AX на BL
cmp ah,0           ;остаток = 0?
jz yes             ;если да, перейти к YES
no:                ;если нет, продолжить
...
yes:
...
```

7.2. Преобразование числа в строку

Чтобы вывести число на экран, нужно преобразовать его в строку. В высокоуровневых языках программирования это действие давно стало тривиальным: для преобразования числа в строку достаточно вызвать всего одну функцию. А вот на языке ассемблера эту подпрограмму еще нужно написать, чем мы и займемся в этом параграфе.

Как бы мы решали эту задачу на С? Делили бы в цикле на 10 и записывали остатки, преобразуя их в символы цифр добавлением кода цифры 0 (см. таблицу кодов ASCII, рис.1.2). Цикл повторялся бы до тех пор, пока частное не стало бы нулем. Вот соответствующий код:

```
#include <unistd.h>
void main(void) {
    unsigned int number;
    char remainder;
    number=12345678;
    while (number != 0)
    {
        remainder = (number % 10)+'0';
        /* remainder = number mod 10 + char('0') */
        number /=10; /* number = number div 10*/
        printf("%c",remainder);
    }
}
```

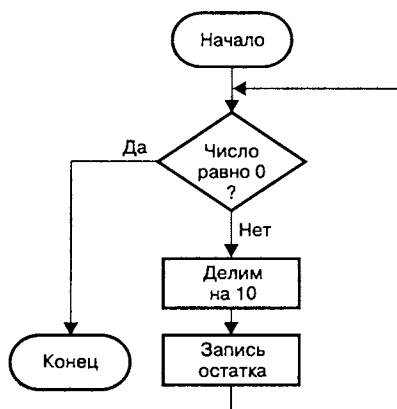


Рис. 7.2. Блок-схема алгоритма преобразования числа в строку

Правда, программа выведет немного не то, что мы ожидали: наше число равно 12345678, но на экране мы увидим 87654321, — это потому что при делении мы получаем сначала младшие цифры, а потом старшие. Как теперь перевернуть полученную последовательность символов? При программировании на языке ассемблера нам доступен стек, куда можно сохранять наши остатки, а затем, вытолкнув их из стека, получить правильную последовательность.

Мы оформим функцию преобразования числа в строку в виде подпрограммы, которую вы можете использовать в любой своей программе на языке ассемблера. Первая проблема, которую предстоит решить — передача параметров. Мы знаем, что в высокоуровневых языках программирования параметры передаются через стек. Однако мы будем использовать более быстрый способ передачи параметров — через регистры процессора.

В регистр EAX мы занесем число, а после выполнения подпрограммы регистр EDI будет содержать адрес памяти (или указатель), по которому сохранен первый байт нашей строки. Сама строка будет заканчиваться нулевым байтом (как в C).

Наша подпрограмма `convert` состоит из двух циклов. Первый цикл соответствует циклу `while` из предыдущего листинга — внутри него мы будем помещать остатки в стек, попутно преобразуя их в символы цифр, пока частное не сравняется с нулем. Второй цикл извлекает цифры из стека и записывает в указанную строку.

Наша подпрограмма `convert` может выглядеть так, как показано в листинге 7.1.

Листинг 7.1. Программа преобразования числа в строку (предварительный вариант)

```
convert:
mov ecx,0           ;ECX = 0
mov ebx,10          ;EBX = 010
.divide:
mov edx,0           ;EDX = 0
div ebx             ;делим EAX на EBX, частное в EAX,
                   ;остаток в EDX
add edx,'0'         ;добавляем ASCII-код цифры 0 к остатку
push edx            ;сохраняем в стеке
inc ecx             ;увеличиваем счетчик цифр в стеке
cmp eax,0           ;закончили? (частное равно 0?)
jnz .divide         ;если нет, переходим к .divide
                   ;иначе число уже преобразовано, цифры
                   ;сохранены в стеке, ECX содержит их
                   ;количество

.reverse:
pop eax             ;выталкиваем цифру из стека
```



```

mov [edi],al      ;помещаем ее в строку
add edi,1         ;перемещаем указатель на следующий символ
dec ecx          ;уменьшаем счетчик цифр, оставшихся
                ;в стеке
cmp ecx,0        ;цифры кончились?
jnz .reverse     ;Нет? Обрабатываем следующую
ret              ;Да? Возвращаемся

```

Эту программу я намеренно написал с недостатками. Попробуем ее оптимизировать, руководствуясь предыдущей главой.

Начнем с замены команды `MOV ecx,0` на более быструю — `XOR ecx,ecx`. Далее, вместо загрузки в `EBX` 10 мы можем сбросить этот регистр (записать в него 0), а потом загрузить 10 в `BL`: так мы избавимся от использования «длинного» непосредственного операнда.

Проверка `EAX` на 0 может быть заменена инструкцией `OR eax,eax` (или `TEST eax,eax`). А две команды записи байта в строку можно заменить одной: вместо

```

mov [edi],al
add edi,1

```

напишем:

```

stosb

```

Эта команда делает то же самое, но занимает всего один байт.

Последний цикл нашей функции можно переписать с использованием команды `LOOP`. В итоге функция будет выглядеть так, как показано в листинге 7.2.

Листинг 7.2. Программа преобразования числа в строку (промежуточный вариант)

```

convert:
xor ecx,ecx      ;ECX = 0
xor ebx,ebx      ;EBX = 0
mov bl,10        ;теперь EBX = 010
.divide:
xor edx,edx      ;EDX = 0
div ebx          ;делим EAX на EBX, частное в EAX,
                ;остаток в EDX
add dl,'0'       ;добавляем ASCII-код цифры 0 к остатку
push edx         ;сохраняем в стеке
inc ecx          ;увеличиваем счетчик цифр в стеке
or eax,eax       ;закончили? (частное равно 0?)
jnz .divide      ;если не 0, переходим к .divide. Иначе число
                ;уже преобразовано, цифры сохранены в стеке,
                ;ECX содержит их количество

```

```

.reverse:
pop eax          ;выталкиваем цифру из стека
stosb            ;записываем AL по адресу, содержащемуся в
                  ;EDI, увеличиваем EDI на 1
loop .reverse    ;ECX=ECX-1, переходим, если ECX не равно 0
ret              ;Да? Возвращаемся

```

Мы слегка улучшили программу, но она все еще работает неправильно. Необходимый нулевой байт не записывается в конец строки, поэтому нужно добавить дополнительную инструкцию:

```
MOV byte [edi],0
```

Ее нужно вставить между LOOP и RET.

Обратите внимание на слово `byte` — оно очень важно, поскольку сообщает компилятору о том, какого размера нуль нужно записать по адресу EDI. Самостоятельно он определить этого не может. Строку завершает нулевой байт, поэтому нуль размером в байт мы и запишем.

Наша программа уничтожает исходное содержимое некоторых регистров (EAX, EBX, ECX, EDX и EDI). Чтобы их сохранить, перед изменением этих регистров нужно поместить их значения в стек, а затем извлечь их оттуда в правильном порядке.

Вызывать нашу подпрограмму нужно так:

```

...
mov eax,0x12345678 ;загрузить в EAX число, подлежащее
                   ;преобразованию
mov edi,buff        ;загрузить в EDI адрес буфера для
                   ;записи строки
call convert         ;вызов подпрограммы
...

```

Число, которое мы хотим преобразовать, перед вызовом подпрограммы загружаем в регистр EAX. Второй параметр — это указатель на адрес в памяти, куда будет записана наша строка. Указатель должен быть загружен в EDI или DI (в зависимости от режима процессора). Команда CALL вызывает нашу подпрограмму. В результате ее выполнения созданная строка будет записана по указанному адресу.

Наша функция `convert` преобразует число только в десятичное представление. Сейчас мы изменим ее так, чтобы получать число в шестнадцатеричной записи.

Мы получаем десятичные цифры, прибавляя к остаткам ASCII-код цифры нуль. Это возможно, потому что символы цифр в ASCII-таблице расположены последовательно (рис.1.2). Но если наши остатки получаются от деления на 16, то нам понадобятся цифры A — F, расположенные в ASCII-таблице тоже последовательно, но не вслед за цифрой 9. Решение простое: если остаток

больше 9, то мы будем прибавлять к нему другую константу. Вместо добавления к остатку ASCII-кода нуля будем вызывать еще одну подпрограмму, HexDigit:

```
HexDigit:      ; в DL ожидается число 0-15,
               ; нужно сопоставить ему шестнадцатеричную
               ; цифру
cmp dl,10      ; сравниваем DL с 10
jb .less      ; переходим, если меньше
add dl,'A'-10  ; 10 превращается в 'A', 11 в 'B' и т.д.
ret           ; конец подпрограммы
.less:
or dl,'0'     ; можно прибавлять и так
ret           ; конец подпрограммы
```

Модифицируя подпрограмму convert, не забудьте заменить делитель 10 на 0x10, то есть 16.

А теперь обобщим нашу подпрограмму так, чтобы она преобразовывала число в строку в любой системе счисления. После того, как мы написали универсальную подпрограмму превращения остатка в N-ичную цифру, достаточно будет просто передавать основание системы счисления как еще один параметр. Все, что нужно сделать, — это удалить строки кода, загружающие делитель в регистр EBX. Кроме того, для сохранения значений регистров общего назначения мы добавим команды PUSHAD и POPAD.

Окончательную версию подпрограммы, приведенную в листинге 7.3, мы будем использовать в других примерах:

Листинг 7.3. Программа преобразования числа в строку (окончательный вариант)

```
;
; -----
; NumToASCII
; -----
;
; eax = 32-битное число
; ebx = основание системы счисления
; edi = указатель на строку-результат
; Возвращает:
; заполненный буфер
; -----
NumToASCII:
pushad        ; сохраняем все регистры общего назначения
              ; в стеке
xor esi,esi   ; ESI = 0: это счетчик цифр в стеке
convert_loop:
```

```

xor edx,edx          ;EDX = 0
div ebx              ;делим EAX на EBX , частное в EAX,
                    ;остаток в EDX
call HexDigit        ;преобразуем в ASCII
push edx             ;сохраняем EDX в стеке
inc esi              ;увеличиваем счетчик цифр в стеке
test eax,eax         ;все? (EAX = 0)
jnz convert_loop     ;если не 0, продолжаем
cld                  ;сбрасываем флаг направления DF:
                    ;запись вперед

write_loop:
pop eax              ;выталкиваем цифру из стека
stosb                ;записываем их в буфер по адресу ES:(E)DI
dec esi              ;уменьшаем счетчик оставшихся цифр
test esi,esi         ;все? (ESI = 0)
jnz write_loop       ;если не 0, переходим к следующей цифре
mov byte [edi],0     ;заканчиваем строку нулевым байтом
popad                ;восстанавливаем исходные значения
                    ;регистров
ret                  ;все!!!

```

7.3. Преобразование строки в число

Часто бывает нужно прочитать число, то есть извлечь его из строки. Высокоуровневые языки программирования предоставляют ряд библиотечных функций для преобразования строки в число (`readln`, `scanf`), а мы напишем простенькую функцию такого назначения самостоятельно.

Для преобразования одного символа в число мы напишем подпрограмму `convert_char`, которая преобразует цифры '0'-'9' в числа 0-9, а символы 'A'-'F' и 'a'-'f' в числа 10-15 (0xA-0xF). Единственным входным, он же выходной, параметром будет регистр AL, в который перед вызовом подпрограммы нужно будет загрузить один ASCII-символ. В результате работы подпрограммы в этом же регистре окажется числовое значение.

Давайте рассмотрим эту подпрограмму.

```

convert_char:
sub al,'0'           ;вычитаем из символа ASCII-код нуля
cmp al,10             ;если разность меньше 10, то была
                    ;десятичная цифра — больше ничего не нужно
                    ;делать
jb done              ;команда JB — переход, если меньше
                    ;иначе — была буква
add al,'0'           ;AL = исходная буква
and al,0x5f           ;приводим ее к верхнему регистру

```

```
sub al, 'A'-10      ;получаем диапазон от 10
and al, 0x0f        ;вернуть нужно значение 0-15. Если
                   ;буква больше F, то
                   ;очищаем 4 старших бита AL

done:
ret                ;возвращаемся
```

Наша программа пока не предусматривает никакой проверки корректности входных данных: мы просто предполагаем, что байт на входе представляет десятичную цифру либо латинскую букву в верхнем или нижнем регистре. Понятно, что против некорректного вызова она беззащитна.

Сначала мы вычитаем из исходного символа ASCII-код цифры нуля. Если результат попадает в диапазон 0–9, то нужное число уже получено: переходим к метке `done`; если нет, то считаем символ буквой. Благодаря хорошо продуманной таблице ASCII (рис.1.2) код строчной буквы отличается от кода соответствующей ей заглавной только единицей в пятом бите (считая с нулевого), поэтому для перевода буквы в верхний регистр достаточно сбросить этот бит (маска `0x5F`). Следующий шаг — вычитанием сдвинуть значение в другой диапазон, получив из 'A' — `0xA`, из 'B' — `0xB` и т.д. И, наконец, последняя операция AND ограничивает диапазон возвращаемых значений младшими четырьмя битами — числами от `0x00` до `0x0F`.

Теперь подумаем, как реализовать функцию для преобразования чисел со знаком. Довольно непростая задача, но мы ее упростим. Договоримся, что запись отрицательных чисел, и только их, будет начинаться с минуса, и только одного. То есть если в начале строки стоит символ «минус», то, значит, перед нами отрицательное число.

Преобразование отрицательного числа выполняется точно так же, как и положительного, только в самом конце подпрограммы мы выполним операцию `NEG` (отрицание).

Займемся разработкой самого алгоритма преобразования. В первой главе мы обсуждали представление любого числа в виде:

$$a = a_n * z^n + a_{n-1} * z^{n-1} + \dots + a_i * z^i + a_0 * z^0 \quad (n \text{ — это количество цифр})$$

Например, десятичное число 1234 может быть записано так:

$$1234 = 1 * 10^3 + 2 * 10^2 + 3 * 10^1 + 4 * 10^0$$

Аналогично, шестнадцатеричное значение `0x524D` может быть записано как:

$$(524D)_{16} = 5 * 16^3 + 2 * 16^2 + 4 * 16^1 + 13 * 16^0 = 21\,069$$

На первый взгляд кажется, что сама формула подсказывает алгоритм: преобразуем каждую цифру в число по программе `convert_char`, умножаем на соответствующую степень основания, складываем и получаем в итоге нужное число. Но со второго взгляда становится видно, что степени основания мы не

узнаем, пока не прочтем все цифры и по их количеству не определим порядок числа. Неуклюжим решением было бы сначала подсчитать цифры, а потом перейти к «очевидному» алгоритму, но мы поступим изящнее.

Это же число 1234 может быть записано так:

$$1234 = (((1) * 10 + 2) * 10 + 3) * 10 + 4$$

Это означает, что мы можем умножить первую слева цифру на основание, прибавить вторую цифру, снова умножить на основание и т.д. Благодаря этому постепенному умножению нам не нужно заранее знать порядок преобразуемого числа.

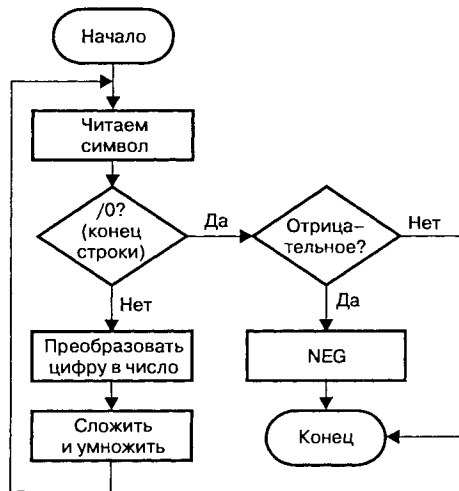


Рис. 7.3. Блок-схема алгоритма преобразования строки в число

Окончательная версия подпрограммы преобразования строки в число приведена в листинге 7.4.

Листинг 7.4. Программа преобразования строки в число

```

;-----
; ASCIIToNum
;-----
; esi = указатель на строку, заканчивающуюся символом с
; кодом 0x0
; ecx = основание системы счисления
; Возвращает:
; eax = число
;-----
ASCIIToNum:
push esi                ;сохраняем указатель в стеке
xor eax,eax             ;EAX = 0
xor ebx,ebx             ;EBX = 0: накопитель для числа
cmp byte [esi], '-'     ;число отрицательное?
jnz .next               ;если нет, не пропускаем следующий
                        ;символ
inc esi                 ;пропускаем символ '-'
.next:
lodsb                   ;читаем цифру в AL
or al,al                ;конец строки?
jz .done
call convert_char       ;преобразовать в число и сохранить
                        ;его в AL
imul ebx,ecx            ;умножить EBX на ECX, сохранить в EBX
add ebx,eax             ;сложить
jmp short .next         ;и повторить
.done:
xchg ebx,eax            ;поместить накопленное число в EAX
pop esi                 ;восстановить исходное значение ESI
cmp byte [esi], '-'     ;результат должен быть отрицательным?
jz .negate              ;да, выполним отрицание
ret                     ;нет, положительным — все готово
.negate:
neg eax                 ;выполняем отрицание
ret                     ;все!!!

```

Глава 8 **Операционная система**



Ассемблер на примерах.
Базовый курс

Операционная система предоставляет интерфейс, то есть средства взаимодействия, между прикладными программами и аппаратным обеспечением (оборудованием) компьютера. Она управляет системными ресурсами и распределяет эти ресурсы между отдельными процессами.

8.1. Эволюция операционных систем

История операционных систем началась в 1950-х годах. Первые компьютеры требовали постоянного внимания оператора: он вручную загружал программы, написанные на перфокартах, и нажимал всевозможные кнопки на пульте управления, управляя вычислительным процессом. Простой такого компьютера обходились очень дорого, поэтому первые операционные системы были разработаны для автоматического запуска следующей задачи после окончания текущей. Часть операционной системы — так называемый монитор — управлял последовательным выполнением задач и позволял запускать их в автоматизированном пакетном режиме.

Следующим этапом в развитии операционных систем стало создание специального компонента операционной системы — ее ядра (1960-е годы). Причина появления ядра была довольно простой. Периферийное оборудование компьютеров становилось все более разнообразным, и задача управления отдельными устройствами все более усложнялась. Ядро операционной системы предоставило стандартный интерфейс для управления периферийными устройствами.

При загрузке операционной системы ядро загружается в память, чтобы впоследствии программы могли обращаться к периферийным устройствам через него. Постепенно в ядро добавлялись новые функции. С исторической точки зрения особенно интересна служба учета, позволившая владельцу компьютера выставлять счет пользователю в соответствии с затраченным на его задание машинным временем.

Разработчики операционных систем старались более эффективно использовать ресурсы компьютера. Из очевидного соображения о том, что нерационально выполнять на компьютере только одно задание, если другое, использующее другие периферийные устройства, могло бы выполняться одновременно с ним, в 1964 году родилась концепция мультипрограммирования.

Мультипрограммирование поставило операционные системы перед новым вызовом: как распределить ресурсы компьютера, то есть процессор, память, периферийные устройства, между отдельными программами? В этой главе мы рассмотрим ответы на этот вопрос.

8.2. Распределение процессорного времени. Процессы

Одной из важнейших задач, решаемых операционной системой, является распределение процессорного времени между отдельными программами.

Процессы

Попросту говоря, процесс — это загруженная в оперативную память программа, которой выделено процессорное время. Каждый процесс, подобно человеку, от кого-то родился, но в отличие от человека у процесса родитель один — это запустивший его процесс. Родительский процесс еще называют предком.

Всех потомков или «детей» процесса (то есть те процессы, которые он *породил*) называют «дочерними» процессами. «Родословное дерево» называется иерархией процессов.

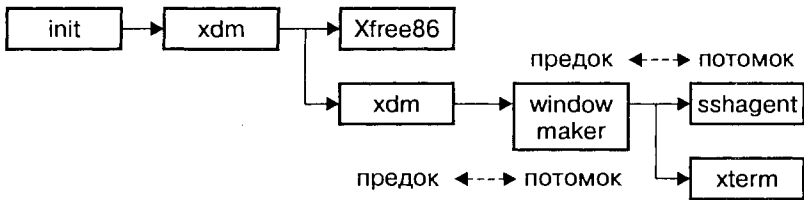


Рис. 8.1. Иерархия процессов

Загрузившись, ядро операционной системы запускает первый процесс. В операционных системах UNIX (Linux) он называется `init`. Этот процесс станет «прародителем» всех остальных процессов, протекающих в системе. В операционной системе DOS «прародителем» является командный интерпретатор `COMMAND.COM`.

Дочерний процесс может унаследовать некоторые свойства или данные от своего родительского процесса. Процесс можно «убить» (`kill`), то есть завершить. Если убит родительский процесс, то его дочерние процессы либо «усыновляются» другим процессом (обычно `init`), либо тоже завершаются.

Планирование процессов

На самом деле процессы выполняются не параллельно, как если бы они были запущены на независимых компьютерах. Если у компьютера только один процессор, то процессы будут выполняться в так называемом псевдопараллельном режиме. Это означает, что каждый процесс разбивается на множество этапов. И этапы разных процессов по очереди получают кванты процессорного времени — промежутки времени, в течение которого они монопольно занимают процессор.

Список всех запущенных процессов, называемый очередью заданий, ведет ядро операционной системы. Специальная часть ядра — **планировщик заданий** — выбирает задание из очереди, руководствуясь каким-то критерием, и «пробуждает» его, то есть выделяет ему квант времени и передает процесс диспетчеру процессов. **Диспетчер процессов** — это другая часть ядра, которая следит за процессом во время его выполнения.

Диспетчер должен восстановить контекст процесса. Восстановление контекста процесса напоминает обработку прерывания, когда все регистры должны быть сохранены, а потом, после окончания обработки, восстановлены, чтобы прерванная программа смогла продолжить работу. После восстановления контекста ядро передает управление самому процессу. Затем, по прошествии кванта времени, выполняемый процесс прерывается и управление передается обратно ядру. Планировщик заданий выбирает следующий процесс, диспетчер процессов его запускает, и цикл повторяется.

Состояния процессов

Ядро операционной системы внимательно «наблюдает» за каждым процессом. Вся необходимая информация о процессе хранится в структуре, которая называется **блоком управления процессом** (PCB, process control block).

В операционной системе UNIX процесс может находиться в одном из пяти различных состояний:

- **Рождение** — пассивное состояние, когда самого процесса еще нет, но уже готова структура для появления процесса.
- **Готовность** — пассивное состояние: процесс готов к выполнению, но процессорного времени ему пока не выделено.
- **Выполнение** — активное состояние, во время которого процесс обладает всеми необходимыми ему ресурсами. В этом состоянии процесс непосредственно выполняется процессором.
- **Ожидание** — пассивное состояние, во время которого процесс заблокирован, потому что не может быть выполнен: он ожидает какого-то события, например, ввода данных или освобождения нужного ему устройства.
- **Смерть процесса** — самого процесса уже нет, но может случиться, что его «место», то есть структура, осталось в списке процессов (процессы-зомби).

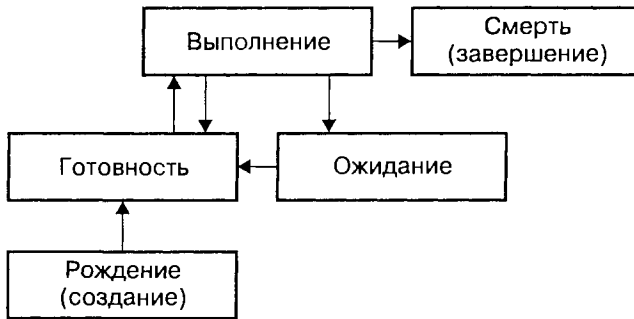


Рис. 8.2. Жизненный цикл процесса

Процессы в DOS состояний не имеют, поскольку DOS — это однозадачная операционная система: процессор предоставлен в монопольное распоряжение только одного процесса.

Ядро операционной системы хранит следующую информацию о процессе:

- ♦ Размещение в памяти.
- ♦ Ресурсы процесса (открытые файлы, устройства и т.п.).
- ♦ Контекст процесса.
- ♦ Состояние процесса.
- ♦ Имя процесса.
- ♦ Идентификационный номер процесса (PID, Process ID).
- ♦ Идентификационный номер родительского процесса.
- ♦ Права доступа.
- ♦ Текущий рабочий каталог.
- ♦ Приоритет.

Стратегия планирования процессов

Пока мы не сказали ничего о том, как именно ядро (точнее, планировщик заданий) решает, какой процесс нужно запустить следующим.

Простейшая стратегия планирования процессов называется кольцевой (Round Robin). Все процессы ставятся в очередь. Планировщик выбирает первый процесс и передает его диспетчеру. Через определенное время ядро прерывает первый процесс, перемещает его в конец очереди и выбирает следующий процесс.

Другая стратегия распределения процессорного времени основана на приоритетах. Каждому процессу в очереди назначен некоторый приоритет, и в соответствии с ним планировщик распределяет процессорное время — статически или динамически. Статическое назначение приоритетов означает,

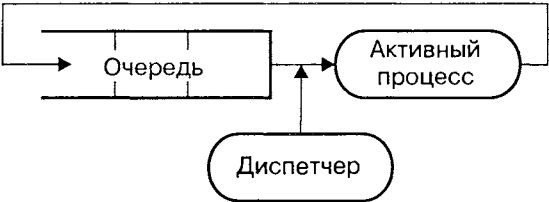


Рис. 8.3. Кольцевая стратегия Round Robin

что сначала будут выполняться процессы с высоким приоритетом, причем приоритет процесса не может быть изменен на протяжении всего времени выполнения процесса. При динамическом же назначении приоритет может быть изменен во время выполнения.

8.3. Управление памятью

Операционная система отслеживает свободную оперативную память, реализует стратегию выделения оперативной памяти и освобождения ее для последующего использования. Оперативная память делится между ядром и всеми выполняемыми процессами.

Простое распределение памяти

Стратегия простого распределения памяти была самой первой. Она состоит в том, что доступная память делится на блоки фиксированного размера, одни из которых доступны только ядру, а другие — остальным процессам. Для разделения памяти на блоки используются различные аппаратные и программные методы.

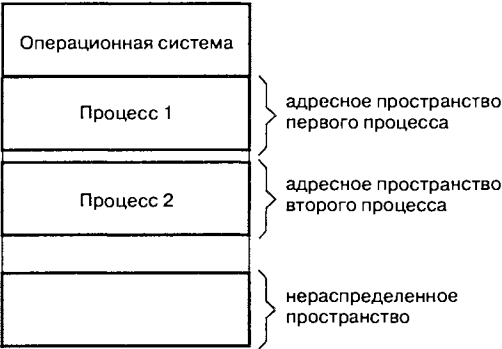


Рис. 8.4. Фиксированное распределение памяти между процессами

Каждый блок памяти может быть занят только одним процессом и будет занят, даже если процесс еще не начал выполняться. Ядро обеспечивает защиту блоков памяти, занятых различными процессами.

Позже эта схема была усовершенствована введением блоков переменной длины. При запуске процесс сообщает, сколько памяти ему нужно для размещения. Это была уже технология переменного распределения памяти.

Обе технологии распределения памяти сейчас считаются устаревшими и практически не используются, потому что они не могут удовлетворить запросы современных задач. Случается, например, что размер процесса превышает весь имеющийся объем оперативной памяти. При традиционной стратегии распределения памяти операционная система просто не сможет запустить такой процесс. Вот поэтому была разработана новая стратегия распределения памяти — стратегия свопинга, которая позволяет использовать пространство жесткого диска для хранения не помещающейся в оперативную память информации.

Свопинг (swapping) — организация подкачки

Пока процессы и размеры обрабатываемых данных были небольшими, всех вполне устраивало фиксированное распределение памяти. С появлением многозадачных операционных систем запущенные процессы не всегда умещались в оперативную память. Поэтому было решено выгружать не используемые в данный момент данные (и процессы!) на жесткий диск. В оперативной памяти остается только структура процесса, а все остальное выгружается. Эта процедура называется свопингом или подкачкой.

Когда системе нужны данные, находящиеся в файле (разделе) подкачки, система подгружает их в оперативную память, выгрузив предварительно в файл подкачки другие данные.

Виртуальная память и страничный обмен

Закон Мерфи гласит: программа растет до тех пор, пока не заполнит всю доступную память.

Решением проблемы стало «притвориться», что в распоряжении операционной системы больше оперативной памяти, чем на самом деле. Этот механизм называется виртуальной памятью и чаще всего реализуется через систему страничного обмена.

Каждому процессу выделено виртуальное адресное пространство, то есть диапазон адресов, ограниченный только архитектурой процессора, а не объемом физически установленной оперативной памяти. Виртуальное адресное пространство делится на страницы одинакового размера, обычно 4 Кб. Физическая оперативная память делится на так называемые фреймы, размер каждого фрейма равен размеру страницы.

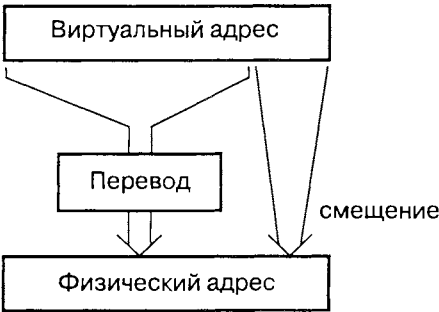


Рис. 8.5. Преобразование виртуального адреса в физический

Перед каждым обращением к памяти процессор запрашивает так называемый блок управления памятью (MMU, Memory Management Unit), чтобы он преобразовал виртуальный адрес в физический.

Трансляция виртуального адреса в физический производится с использованием таблицы страниц, каждая строка которой описывает одну виртуальную страницу.

Запись таблицы страниц

Другие флаги	Флаги буфера	Физический адрес	Р Бит присутствия
--------------	--------------	------------------	----------------------

Рис. 8.6. Запись таблицы страниц

Наиболее важный бит — бит присутствия страницы. Если он установлен, то эта страница сохранена во фрейме, номер которого (адрес) определен в физическом адресе. Другие биты в таблице определяют права доступа или разрешения для страницы (read/write/execute) или используются для буфера. Таблица страниц индексирована номером виртуальной страницы.

Рассмотрим пример трансляции (преобразования) виртуального адреса в физический (рис. 8.7).

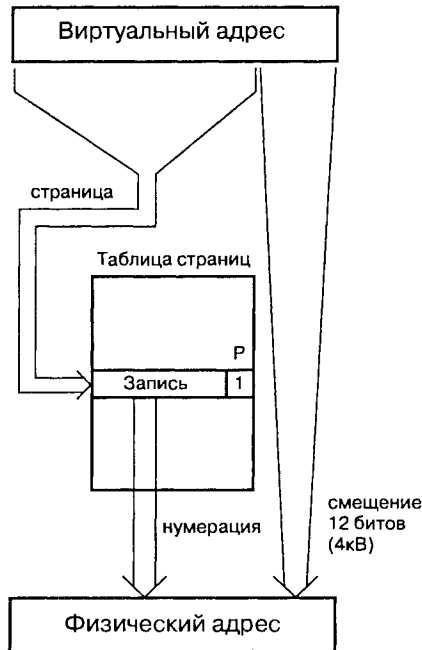


Рис. 8.7. Принцип преобразования виртуального адреса в физический

По виртуальному адресу MMU вычисляется номер виртуальной страницы, на которой он находится. Разность между адресом начала страницы и требуемым адресом называется смещением.

По номеру страницы ищется строка в таблице страниц. Бит присутствия в этой строке указывает, находится ли виртуальная страница в физической памяти компьютера (бит Р установлен в 1) или в файле (разделе) подкачки на диске.

Если страница присутствует в физической памяти, то к физическому адресу (номеру фрейма) добавляется смещение, и нужный физический адрес готов. Если страница выгружена из оперативной памяти (бит Р установлен в 0), то процессор (или его MMU) вызывает прерывание «Страница не найдена» (Page Not Found), которое должно быть обработано операционной системой.

В ответ на прерывание «Страница не найдена» операционная система должна загрузить требуемую страницу в физическую оперативную память. Если заняты не все фреймы, то ОС загружает страницу из файла подкачки в свободный фрейм, исправляет таблицу страниц и заканчивает обработку прерывания.

Если свободного фрейма нет, то операционная система должна решить, какую страницу переместить из фрейма в область подкачки, чтобы освободить физи-

ческую память. В освобожденный фрейм будет загружена страница, которая первоначально вызвала прерывание. После этого ОС исправит биты присутствия в записях об обеих страницах и закончит обработку прерывания.

Для процесса процедура преобразования виртуальных адресов в физические абсолютно прозрачна («незаметна») — он «думает», что у вашего компьютера просто много оперативной памяти.

В x86-совместимых компьютерах каждая программа может иметь несколько адресных пространств по 4 ГБ. В защищенном режиме значения, загруженные в сегментные регистры, используются как индексы таблицы дескрипторов, описывающей свойства отдельных сегментов, или областей, памяти. Рассмотрение сегментов выходит далеко за пределы этой книги.

8.4. Файловые системы

Файловая система — это часть операционной системы, предназначенная для управления данными, записанными на постоянных носителях. Она включает в себя механизмы записи, хранения и чтения данных.

Когда компьютеры только начали развиваться, программист должен был знать точную позицию данных на носителе. Почему мы используем слово «носитель», а не «диск»? В то время дисков как таковых или еще не было, или они были слишком дороги, поэтому для хранения данных использовалась магнитная лента. С появлением дисков все еще больше усложнилось, поэтому для облегчения жизни программистам и пользователям и была разработана файловая система, которая стала частью ядра.

Структура файловой системы

Кроме программного модуля в ядре, в понятие файловой системы входит структура данных, описывающая физическое местоположение файлов на диске, права доступа к ним и, конечно же, их имена.

Сам файл состоит из записей, которые могут быть фиксированной или переменной длины.

Метод доступа к записям файла зависит от его типа. Самый простой — метод последовательного доступа, при котором для чтения нужной записи требуется сначала прочитать все предшествующие ей записи. Если записи проиндексированы, то к ним можно обращаться также по индексу. Такой тип файла называется индексно-последовательным (IBM 390, AS/400).

Что же касается операционной системы, то она не устанавливает формат файла. Файл состоит из последовательности байтов, а структура самого файла должна быть понятна приложению, которое используется для его обработки.

Поговорим о методах доступа к файлу в операционных системах DOS и UNIX.

Доступ к файлу

Перед началом обработки любого файла его нужно открыть. Это делает операционная система по запросу программы. В запросе нужно указать имя файла и требуемый метод доступа (чтение или запись). Ядро возвратит номер файла, который называется файловым дескриптором. В дальнейшем этот номер будет использоваться для операций с файлом.

Имя файла обычно содержит путь к этому файлу, то есть перечисление всех каталогов, необходимых для того, чтобы система смогла найти файл. Имена подкаталогов в составе пути разделяются специальным символом, который зависит от операционной системы. В DOS это `\`, а в UNIX — `/`.

Позиция в файле определяется числом — **указателем**. Его значение — это количество прочитанных или записанных байтов. С помощью специального системного вызова можно переместить указатель на заданную позицию в файле и начать процесс чтения или записи с нужного места.

Максимальный размер блока данных, который может быть прочитан или записан за раз, ограничен буфером записи. Завершив работу с файлом, его нужно закрыть.

В мире UNIX укоренилась идея представлять устройства ввода/вывода в виде обычных файлов: чтение символов с клавиатуры и вывод их на экран выполняются теми же самыми системными вызовами, что и чтение/запись их в файл.

Дескриптор «файла» клавиатуры — 0, он называется стандартным потоком ввода (`stdin`). У экрана два дескриптора — стандартный поток вывода (`stdout`) и стандартный поток ошибок (`stderr`). Номер первого идентификатора — 1, второго — 2.

Операционная система может не только читать и записывать файлы, но также и перенаправлять потоки ввода/вывода. Например, можно перенаправить вывод программы в файл вместо вывода на экран, не меняя исходного текста программы и не перекомпилируя ее. Перенаправление известно как в DOS, так и в UNIX, и выполняется с помощью специального системного вызова, который обычно принимает два аргумента — файловые дескрипторы «соединяемых» файлов. В DOS и UNIX можно перенаправить вывод программы в файл так:

```
ls > file1
```

Фактически интерпретатор команд (или оболочка) открывает файл `file1` перед запуском `ls`, получает его дескриптор и, используя системный вызов, заменяет дескриптор стандартного вывода полученным дескриптором. Программа `ls` и понятия не имеет, куда записываются данные — она просто записывает их на стандартный вывод. Благодаря системному вызову вывод попадает в указанный файл, а не на экран.

Физическая структура диска

Файловая система обращается к диску непосредственно (напрямую), и поэтому она должна знать его физическую структуру (геометрию). Магнитный диск состоит из нескольких пластин, обслуживаемых читающими/пишущими головками (рис. 8.8). Пластины разделены на дорожки, а дорожки — на сектора. Дорожки, расположенные друг над другом, образуют «цилиндр». Исторически сложилось так, что точное место на диске определяется указанием трех «координат»: цилиндра, головки и сектора.

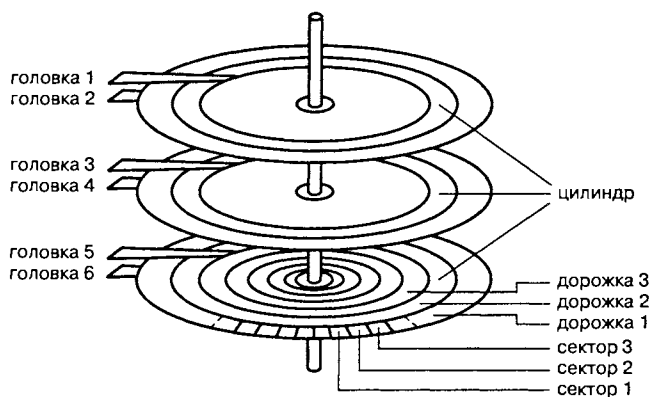


Рис. 8.8. Физическая структура диска

Дорожки — это концентрические круги, разделенные на отдельные сектора. Подобно странице памяти, сектор диска — это наименьший блок информации, размер его обычно равен 512 байтам. Головки чтения/записи «плавают» над поверхностью диска. Значения трех вышеупомянутых «координат» позволяют головкам определить нужный сектор диска для чтения или записи данных.

Для цилиндров, головок и секторов были определены диапазоны приемлемых значений. Поскольку объемы дисков росли, увеличивалось и количество цилиндров, головок и секторов. Увеличивалось оно до тех пор, пока не вышло за пределы этого самого диапазона. Вот поэтому некоторые старые компьютеры не могут прочитать диски размером 60 Гб и более (а некоторые и того меньше). Вместо увеличения этих диапазонов были разработаны различные преобразования, которые позволяют вернуть комбинации сектора, головки и цилиндра назад в указанный диапазон.

Было решено заменить адресацию геометрического типа логической (или линейной) адресацией, где отдельные секторы были просто пронумерованы от 0 до последнего доступного сектора. Теперь для того, чтобы обратиться к сектору, нужно просто указать его номер.

Логические диски

Необязательно, чтобы файловая система занимала весь диск. Обычно диск разбивают на логические диски, или разделы. Так даже безопаснее: например, на одном логическом диске у вас находится операционная система, на другом — прикладные программы, на третьем — ваши данные. Если какая-то программа повредила один раздел, остальные два останутся неповрежденными.

Первый сектор любого диска отведен под таблицу разделов (partition table). Каждая запись этой таблицы содержит адреса начального и конечного секторов одного раздела в геометрической (три «координаты») и логической (последовательный номер) форме. А на каждом разделе хранится таблица файлов, позволяющая определить «координаты» файла на диске.

8.5. Загрузка системы

Этапы загрузки

После нажатия кнопки питания компьютер «оживает». Сначала запускается программа, сохраненная в ROM (read-only memory). Память ROM содержит так называемую базовую систему ввода/вывода (BIOS, Basic Input Output System). В задачу BIOS входит инициализация компьютера — проверка видеоадаптера, памяти, дисков и других устройств — это так называемая процедура POST (Power On Self Test).

После проверки устройств компьютера BIOS начинает загрузку операционной системы. В зависимости от настроек BIOS может искать начальный загрузчик, которому и будет передано управление, на дискете, жестком диске (сектор с номером 0), CD-ROM и т.д. Первый сектор (с номером 0) диска содержит начальный загрузчик и таблицу разделов. Этот сектор называется главной загрузочной записью — MBR (Master Boot Record). BIOS загружает код начального загрузчика из MBR в оперативную память и передает ему управление.

Задача начального загрузчика — найти вторичный загрузчик операционной системы, который находится на одном из логических дисков, перечисленных в таблице разделов. Найти такой диск очень просто — в таблице разделов он единственный помечен как активный. На каждом логическом диске зарезервировано место (первый сектор) для загрузчика операционной системы. Начальный загрузчик загружает загрузчик операционной системы и передает ему управление, а тот в свою очередь загружает ядро и передает управление ему: с этого момента операционная система запущена.

Немного о том, что такое BIOS

BIOS (Basic Input/Output System — Базовая Система Ввода/Вывода) — обязательная часть каждого PC. Ее задача заключается в предоставлении стандартного способа доступа к аппаратным ресурсам компьютера. Во времена операционной системы DOS BIOS была своеобразным мостом между операционной системой и аппаратными средствами компьютера.

Современные операционные системы используют BIOS только при загрузке операционной системы в память, а все функции по управлению устройствами они предоставляют самостоятельно. Сервисы BIOS доступны через программные прерывания (табл. 8.1).

Программные прерывания и соответствующие им сервисы BIOS

Таблица 8.1

Номер прерывания	Функция
0x10	Службы экрана (управление выводом на экран)
0x13	Дисковый ввод/вывод
0x14	Управление последовательными портами
0x15	Расширенные сервисы
0x16	Работа с клавиатурой
0x17	Работа с принтером

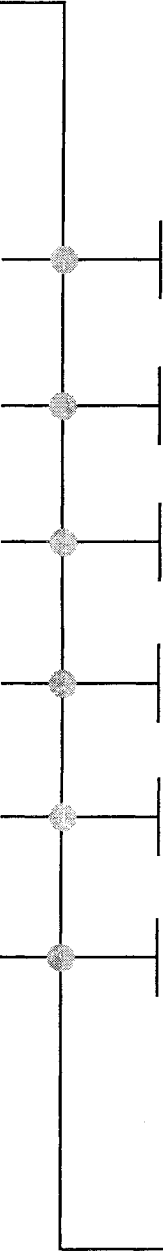
Видеоадаптеры (и некоторые другие устройства) оснащены своими собственными BIOS, поскольку невозможно учесть особенности всех видеоадаптеров в BIOS компьютера.

Мы не будем рассматривать BIOS подробнее, потому что для ввода данных и вывода на экран будем пользоваться стандартными средствами операционной системы.



Рис. 8.9. Пирамида доступа к аппаратным средствам компьютера

Глава 9 Компилятор NASM



Предложения языка ассемблера
Выражения
Локальные метки
Препроцессор NASM
Директивы Ассемблера
Формат выходного файла

Для компиляции наших программ мы будем использовать компилятор NASM (Netwide Assembler), который свободно распространяется (вместе со своими исходными кодами) по лицензии LGPL. Вы можете найти этот компилятор на сайте <http://nasm.sourceforge.net>.

Синтаксис компилятора подобен синтаксису коммерческих компиляторов MASM (Microsoft Assembler) и TASM (Turbo Assembler от Borland), поэтому использовать NASM вдвойне удобно — никаких проблем с лицензией и возможность переноса программ, написанных для других ассемблеров.

9.1. Предложения языка ассемблера

Каждая строка программы (предложение языка ассемблера) может состоять из следующих частей:

Метка Инструкция Операнды Любые комментарии

Перед меткой и после инструкции можно использовать любое количество пробельных символов (пробелы, табуляция). Операнды отделяются друг от друга с помощью запятой. Начало комментария — точка с запятой (;), а концом комментария служит конец строки.

Если предыдущая строка очень длинная, ее можно перенести на следующую, используя обратный слеш '\ ' (как в C).

Инструкцией может быть команда или псевдокоманда (директива компилятора).

9.2. Выражения

Под словом «выражение» мы понимаем константное выражение, значение которого известно уже во время компиляции. Синтаксис выражений NASM подобен синтаксису выражений языка C.

Выражение также может быть написано как сумма адреса в символической форме и некоторого числа. Выражения такого типа служат, например, для

доступа ко второму элементу массива в инструкции `MOV eax, [array+4]`. Значение выражения в квадратных скобках известно во время компиляции: это начальный адрес массива `Array`, увеличенный на 4.

Выражения можно использовать во многих местах программы, например:

```
add dl, 'A'-10
```

Выражение `'A'-10` будет вычислено во время компиляции, и компилятор сгенерирует команду `ADD dl, 55`.

Нам никто не запрещает использовать сложные выражения, которые известны нам из языка C. В следующей таблице (табл. 9.1) перечислены все поддерживаемые операторы в порядке возрастания приоритета.

Операторы, поддерживаемые NASM, расположенные
в порядке возрастания приоритета

Таблица 9.1

Оператор	Назначение
	Логическая операция OR
^	Логическая операция XOR
&	Логическая операция AND
<< >>	Сдвиг влево и вправо
+ -	Плюс и минус
* / % // %%	Умножение, целочисленное деление, деление по модулю (взятие остатка), деление со знаком, деление со знаком по модулю
унарные + - ~	Унарные операторы плюс, минус и отрицание (NOT)

Рассмотрим несколько примеров:

```
mov eax, ((5*6+2)+(0x40<<2))/8      ;вычислится как:
                                       ;MOV eax,0x24
```

Еще один пример:

```
mov cl,~1      ;результат: mov cl,11111110b
                ;или: mov cl,0xFE
```

9.3. Локальные метки

С метками мы познакомились при рассмотрении инструкций `JMP` и `CALL`, поэтому вы уже знаете, для чего они используются. **Метка — это идентификатор, за которым следует двоеточие.** Двоеточие можно и опускать, но я советую этого не делать, потому что иначе компилятор может принять имя метки за написанное с ошибкой имя команды. Каждая метка должна быть уникальной в пределах программы, поэтому при написании больших программ вам понадобится весь ваш творческий потенциал, чтобы каждый раз придумывать

новое имя. К счастью, нас выручит компилятор NASM, который позволяет использовать так называемые локальные метки.

Локальные метки, как и локальные переменные в высокоуровневых языках программирования, могут иметь одинаковые имена в разных подпрограммах. Но в языках высокого уровня область видимости переменной (блок, функция) видна сразу, а в языке ассемблера? **Локальной называется метка, действительная только на протяжении фрагмента кода, находящегося между двумя глобальными метками. Различаются локальные и глобальные метки своими именами: имя локальной метки начинается с точки, а глобальной — с любого другого символа.**

```
subprog:          ;глобальная метка subprog
...              ;код
.local_loop:      ;локальная метка
...              ;код
subprog2:         ;глобальная метка subprog2
...
.local_loop:      ; локальная метка
...
```

9.4. Преппроцессор NASM

В этом параграфе мы поговорим о средствах, несколько сближающих программирование на языке ассемблера с программированием на языках высокого уровня — макросах.

Макрос — это символическое имя, заменяющее несколько команд языка ассемблера. Команды подставляются вместо имени макроса во время компиляции. Этим занимается программа, называемая преппроцессором и вызываемая, как правило, самим компилятором до того, как он приступит непосредственно к трансляции символических имен команд в машинные коды. Преппроцессор NASM очень мощный и гибкий, он поддерживает множество различных макросов.

Директивы объявления макроса начинаются со знака процента «%». Имена макросов чувствительны к регистру символов (то есть AaA — это не aaa). Для определения не чувствительных к регистру символов макросов нужно использовать другие директивы, которые начинаются с пары символов «%i» (%idefine вместо %define и т.п.).

Однострочные макросы — %define, %undef

Эти макросы определяются подобно тому, как в языке C, но в отличие от C могут принимать аргументы. Вот пример макроса для вычисления среднего арифметического двух чисел:

```
%define average(a,b) (((a)+(b))/2)
```

Использовать данный макрос в программе очень просто:

```
mov al,average(3,7)
```

Инструкция, которая будет «подставлена» вместо макроса:

```
mov al,5
```

Очень часто **%define** используется для определения констант, как в C:

```
%define SEC_IN_MIN 60
%define SEC_IN_HOUR SEC_IN_MIN * 60
```

Или для условной компиляции, например:

```
%define USE_MMX
```

Так же, как и в C, мы можем проверить, определен ли макрос, с помощью **%ifdef** (определен) и **%ifndef** (не определен). Удалить определение макроса можно с помощью **%undef**.

Сложные макросы — **%macro %endmacro**

Директива **%define** позволяет определить простые макросы, состоящие из одной команды (строки кода). Для определения сложного, многострочного, макроса служат директивы **%macro** и **%endmacro**. Первая описывает начало макроса — имя и число аргументов. Затем идет тело макроса — команды, которые должны быть выполнены. Директива **%endmacro** «закрывает» макрос.

```
%macro subtract 3
sub %1,%2
sub %1,%3
%endmacro
```

Макрос `subtract` нужно вызывать с тремя аргументами:

```
subtract eax,ecx,[variable1]
```

Он будет преобразован в следующий код:

```
sub eax,ecx
sub eax,[variable1]
```

Необязательно указывать точное число аргументов. Препроцессор NASM позволяет указывать переменное число в виде диапазона значений. Например, 2–3 означает, что наш макрос может принимать 2 или 3 аргумента. Опущенные при вызове макроса аргументы будут заменены значениями по умолчанию, которые следуют за диапазоном аргументов:

```
%macro addit 2-3 0
add %1,%2
add %1,%3
%endmacro
```

Мы описали макрос `addit`, который принимает два обязательных аргумента, а вместо третьего, если он не указан при вызове, будет подставлено значение по умолчанию — 0:

```
addit eax ebx
```

Будет преобразовано в:

```
add eax,ebx
add eax,0
```

Объявление макроса — `%assign`

Директива `%assign` — это еще один способ объявить простой (однотрочный) макрос. Объявленный макрос не должен иметь параметров, и его результатом должно быть число. Например, с помощью

```
%assign i i+1
```

мы можем увеличить числовое значение другого макроса на 1. Обычно `%assign` используется в сложных макросах.

Условная компиляция — `%if`

Как и препроцессор C, препроцессор NASM позволяет условно компилировать программу. Это означает, что в зависимости от условия будет откомпилирован тот или иной блок кода.

Рассмотрим основные инструкции условной компиляции:

```
%if<условие>
    ;Код между if и elif будет откомпилирован
    ;только, если условие истинно.
%elif<условие2>
    ;Код между %elif и %else будет откомпилирован
    ;если первое условие ложно, а второе — истинно
%else
    ;Если оба условия ложны, будет откомпилирован код
    ;между %else и %endif
%endif
```

Директивы `%elif` и `%else` необязательны и могут быть опущены. Также можно указывать несколько блоков `%elif`, но блок `%else` (если он есть) должен быть один.

Условие может содержать константное выражение. Набор операторов в условии расширен операторами отношения: `=`, `<`, `>`, `<=`, `>=`, и `<>` (равно, меньше, больше, меньше или равно, больше или равно, не равно). Приверженцам языка C понравится возможность писать `«==»` и `«!=»` вместо `«=»` и `«<>»`. Также при написании условия могут быть использованы логические операторы `|`, `^`, `&` (OR, XOR, AND), которые нам известны из языка C.

Определен ли макрос? Директивы `%ifdef`, `%ifndef`

Директива `%ifdef` используется для проверки существования макроса. Например:

```
%define TEST_IT
%ifdef TEST_IT
  cmp eax,3
  ...
%endif
```

Инструкции, заключенные в блок `%ifdef` (в том числе и `CMP eax,3`) будут откомпилированы, только если макрос `TEST_IT` определен с помощью `%define`.

Кроме `%ifdef` можно использовать директиву `%ifndef`. В этом случае соответствующий блок инструкций будет компилироваться, только если макрос не определен.

Вставка файла — `%include`

С помощью директивы `%include` можно вставить в текущую позицию код, находящийся в другом файле. Имя файла нужно указать в двойных кавычках. Обычно `%include` используется для вставки так называемых заголовочных файлов:

```
%include «macro.mac»
```

Как правило, заголовочные файлы содержат определения констант или макросов, а не команды программы. В отличие от C, нам не нужно добавлять в начало заголовочного файла эту традиционную конструкцию:

```
%ifndef MACROS_MAC
  %define MACROS_MAC
  ;теперь следуют сами макрокоманды
%endif
```

Повторное включение файла не является ошибкой. Просто все макросы будут определены заново, что не должно вызвать ошибку при компиляции.

9.5. Директивы Ассемблера

Директивы Ассемблера NASM — это макрокоманды, определенные самим компилятором. Директив у NASM немного, в отличие от MASM и TASM, где их число огромно.

Директива BITS — указание режима процессора

Директива указывает компилятору, в каком режиме процессора будет работать программа. Синтаксис этой директивы позволяет указать один из двух режимов — 16-разрядный (BITS 16) или 32-разрядный (BITS 32). В большинстве случаев нам не нужно указывать эту директиву, поскольку за нас все сделает сам NASM.

Во второй главе, в которой рассматривался процессор 80386, мы говорили о двух режимах процессора — реальном и защищенном. Реальный режим используется для обратной совместимости с предыдущими чипами. В реальном режиме используются 16-битные код и адресация. Но 80386 также поддерживает 32-битные инструкции. Значит ли это, что 32-битные команды нельзя использовать в 16-битном режиме?

Да, нельзя. Как показано в главе 3, каждой команде соответствует свой машинный код. Разработчикам Intel пришлось решать проблему — как добавить новые команды, работающие с 32-битными операндами, когда таблица кодов команд уже почти заполнена? Они нашли довольно изящное решение.

16-битная команда `MOV AX,0x1234` транслируется в машинный код `0xB8, 0x34, 0x12`.

32-битная команда `MOV EAX,0x00001234` транслируется в машинный код `0x66, 0xB8, 0x34, 0x12, 0x00, 0x00`.

В 16-битном (реальном) режиме машинное представление всех 32-битных команд начинается с префикса `0x66` (а всех обращений к памяти — с префикса `0x67`). После префикса следует код 16-битной версии той же самой команды. Благодаря префиксу операнды тоже могут быть 32-битными.

В защищенном режиме процессор использует 32-битную адресацию и 32-битный код. Поэтому машинная команда `0xB8, 0x34, 0x12, 0x00, 0x00` (то есть без префикса `0x66`) означает `MOV EAX,0x00001234`.

Директива BITS указывает, как будут компилироваться инструкции — с префиксом или без.

Директивы SECTION и SEGMENT — задание структуры программы

Каждая программа, вне зависимости от языка программирования, состоит из трех частей: код программы, статические данные (то есть известные во время компиляции) и динамические данные (неинициализированные, то есть те, под которые во время компиляции только отводится память, а значение им не присваивается).

Эти секции могут быть определены с помощью директив SECTION и SEGMENT.

Традиционно секция кода называется **.text**, секция статических данных — **.data**, а секция динамических данных — **.bss**.

Рассмотрим пример программы, содержащий все три секции (листинг 9.1)

Листинг 9.1. Пример программы, содержащий все три секции

```
; Каждая ассемблерная программа должна начинаться с описания
; своего назначения и авторства, например,
; (c)2005 Иванов И.П. <ivan@ivanov.com>
SECTION .text
; Секция .text содержит нашу программу
; Это очень простая программа для сложения двух значений
mov eax,2           ; EAX = 2
mov ebx,5           ; EBX = 5
add eax,ebx         ; EAX = EAX + EBX
add eax,[stat1]     ; EAX = EAX + stat1
mov [dyn1],eax      ; сохраняем полученное в динамической
; переменной dyn1
SECTION .data
; используем псевдоинструкции DB, DW and DD
; для объявления статических данных
.data
stat1: dd 1
SECTION .bss
; В этой секции описываются динамические данные, они не
; инициализируются. Для объявления переменных в секции .bss
; используются псевдоинструкции RESB, RESW и RESD.
; Динамические данные не занимают места на диске, то есть
; для них не отводится место в исполняемом файле. Инструкции
; RES* указывают только, сколько байтов переменные будут
; занимать в памяти после запуска программы
dyn1: resd 1
; конец программы
```

Иногда также определяется еще одна секция — для стека (**.stack**). Но поскольку о стеке заботится операционная система, нам беспокоиться не о чем.

Директивы **SECTION** и **SEGMENT** являются синонимами и поэтому взаимозаменяемы.

Примечание переводчика.

В другой литературе вы можете встретить использование термина «сегмент» вместо «секция». Соответственно, три части программы называются сегмент кода, сегмент данных и сегмент стека.

Директивы EXTERN, GLOBAL и COMMON — обмен данными с другими программными модулями

Мы будем использовать эти три директивы в главе 13 при линковке (компоновке) программ на языке ассемблера с программами, написанными на высокоуровневых языках программирования, поэтому сейчас мы рассмотрим их очень бегло.

Директива EXTERN подобна одноименной директиве (extern) в С. Она позволяет определить идентификаторы, которые не определены в текущей программе, но определены в каком-то внешнем модуле. Она используется для «импорта» идентификаторов в текущую программу.

Директива GLOBAL определяет идентификаторы для «экспорта» — они будут помечены как глобальные и могут использоваться другими модулями (программами).

Директива COMMON используется вместо GLOBAL для экспорта идентификаторов, объявленных в секции .bss.

Если другой модуль экспортирует с помощью директивы COMMON тот же самый идентификатор, то оба символа будут размещены в памяти по одному и тому же адресу.

Директива CPU — компиляция программы для выполнения на определенном процессоре

Используя директиву CPU, мы можем заставить компилятор генерировать команды только для указанного типа процессора. Мы обсуждаем команды только процессора 80386, поэтому не нуждаемся в директиве CPU. Если эта директива опущена, то в программе можно использовать любые команды x86-совместимых процессоров.

Определение набора команд с помощью директивы CPU полезно для опытного программиста, который может использовать некоторые новые команды специфического типа процессора.

```
CPU 8086      ;Используются инструкции оригинального чипа  
              ;Intel 8086 можно указать другой тип процессора  
              ;т.е. 286, 386, 486, Pentium...
```

Директива ORG — указание адреса загрузки

Директива ORG устанавливает начальный адрес для загрузки программы — тот адрес, от которого отсчитываются все остальные адреса в программе. Например, с помощью ORG вы можете создать COM-файл (подробнее о форматах выходных файлов — следующий параграф), если укажете ORG 0x100. Операционная система DOS загружает исполняемые файлы типа COM в сегмент памяти, начинающийся с адреса 0x100.

Следующий материал предназначен для программистов, которые хотят «копнуть» NASM поглубже, начинающие программисты могут с чистой совестью пропустить этот пункт.

В отличие от компиляторов MASM и TASM, NASM запрещает использовать директиву `ORG` несколько раз в пределах одной секции.

Типичный пример многократного применения директивы `ORG`: вторая `ORG`, означающая конец загрузочного сектора. В MASM или TASM мы можем указать:

```
ORG 0
    ; тут загрузочный сектор
...
    ; конец загрузочного сектора
ORG 510
DW 0xAA55
    ; В NASM вместо второго ORG мы должны использовать
    ; директиву TIMES:
ORG 0
    ; тут загрузочный сектор
...
    ; конец загрузочного сектора
TIMES 510-($-$$) DB 0
DW 0xAA55
```

9.6. Формат выходного файла

Netwide Assembler (NASM) легко портируется на различные операционные системы в пределах x86-совместимых процессоров. Поэтому для NASM не составит труда откомпилировать программу для указанной операционной системы, даже если вы сейчас работаете под управлением другой ОС.

Формат выходного файла определяется с помощью ключа командной строки `-f`. Некоторые форматы расширяют синтаксис определенных инструкций, а также добавляют свои инструкции.

Создание выходного файла: компиляция и компоновка

Создание исполняемого файла состоит из двух этапов. Первый — это **компиляция** (трансляция) исходного кода программы в некоторый объектный формат.

Объектный формат содержит машинный код программы, но символы (переменные и другие идентификаторы) в объектном файле пока не привязаны к адресам памяти.

На втором этапе, который называется **компоновкой** или **линковкой** (linking), из одного или нескольких объектных файлов создается исполняемый файл. Процедура компоновки состоит в том, что компоновщик связывает символы, определенные в основной программе, с символами, которые определены в ее модулях (учитываются директивы EXTERN и GLOBAL), после чего каждому символу назначается окончательный адрес памяти или обеспечивается его динамическое вычисление.

Формат bin — готовый исполняемый файл

Формат **bin** не является объектным форматом. Он содержит результат перевода команд ассемблера в машинные коды.

Этот формат можно даже использовать для создания текстовых файлов. Например, у нас есть файл `hello.asm`:

```
;если не указано иначе, адресация двоичного файла
;начинается с 0x0.
;при помощи директивы DB определим строку "Hello world!",
;завершенную символом конца строки
string: db "Hello world!",0xd,0xa
```

Скомпилируем этот файл, вызвав компилятор с ключом **-f**:

```
nasm -f bin hello.asm
```

В результате получим файл `hello`, содержащий текст «Hello world». Если бы в исходном файле были еще какие-то команды, то в выходной файл `hello` попали бы их машинные коды, при выводе на экран они были бы интерпретированы как символы из ASCII-таблицы, и мы увидели бы на экране всякий мусор.

Формат **bin** позволяет объявлять секции (кода, данных, неинициализированных данных). После имени секции может стоять директива **ALIGN**, требующая располагать эту секцию по адресам, кратным указанному числу. Например, следующая директива задает выравнивание секции кода по адресам, кратным 16:

```
section .text align=16
```

Формат **bin** также можно использовать для создания файлов, исполняемых под DOS (`.COM` и `.SYS`) или для загрузчиков. Значение по умолчанию директивы **BITS** равно 16.

В коде, предназначенном для вывода в формате **bin**, может присутствовать директива **ORG**.

Формат OMF — объектный файл для 16-битного режима

OMF (Object Module Format) — это старый формат, разработанный корпорацией Intel, в котором до сих пор создает объектный код Turbo Assembler. Компиляторы MASM и NASM тоже поддерживают этот формат.

Файл в формате OMF имеет расширение **.obj**, и сам формат часто называется также OBJ. Файлы с расширением **.obj** могут быть скомпонованы в исполняемый файл.

Несмотря на то, что формат **obj** был изначально разработан для 16-битного режима, компилятор NASM также поддерживает его 32-битное расширение. Поэтому NASM можно использовать вместе с 32-битными компиляторами от Borland, которые тоже поддерживают этот расширенный 32-битный формат, а не другой объектный формат, поддерживаемый Microsoft.

Формат OBJ расширяет синтаксис некоторых директив, в частности, SEGMENT (SECTION). Подробное описание этого формата выходит за рамки данной книги, но вы можете прочитать о нем в руководстве по компилятору NASM.

Кроме того, формат OBJ добавляет новую директиву IMPORT, которую мы опишем в главе 11, посвященной программированию в Windows. Директива IMPORT позволяет импортировать указанный файл из DLL, ей нужно передать два параметра — имя идентификатора и DLL.

У каждого OBJ-файла должна быть своя точка входа (как минимум одна). Точка входа определяет позицию, на которую будет передано управление после загрузки программы в память. Точка входа (entry point) обозначена специальным идентификатором (или меткой) **..start:**.

Чтобы получить файл в формате OBJ, нужно вызвать компилятор с ключом командной строки **-f obj**.

Формат win32 — объектный файл для 32-битного режима

Для компоновки программы с помощью Microsoft Visual C++ используется 32-битный формат **win32**. Этот формат должен быть совместим с форматом COFF (Common Object File Format), но на самом деле такой совместимости нет. Если вы будете компоновать программу компоновщиком, основанным на формате COFF, компилируйте ассемблерный код в объектный файл формата **coff**, описанного в следующем пункте.

Чтобы получить файл в формате win32, нужно вызвать компилятор с ключом командной строки **-f win32**.

Форматы aout и aoutb — старейший формат для UNIX

Формат исполняемого файла **a.out** (Assembler and link editor OUTput files) преимущественно используется в Linux. Расширенная версия этого формата **a.outb** используется группой BSD-совместимых операционных систем (NetBSD, FreeBSD и OpenBSD). NASM может генерировать файлы обоих форматов, если указать ключ **-f aout** для Linux или **-f aoutb** для BSD.

Формат **coff** — наследник **a.out**

Формат COFF (Common Object File Format) представляет собой существенно усовершенствованную и доработанную версию формата **a.out**. Он распространен как в мире UNIX, так и в Windows NT, и поддерживается некоторыми свободными (не коммерческими) средами разработки, например, DJGPP. Используется для разработки приложений на C и C++. Если компилятор вызывать с параметром **-f coff**, мы получим на выходе объектный файл в формате COFF. Расширение файла по умолчанию для этого формата — **.o**

Формат **elf** — основной формат в мире UNIX

Формат ELF (Executable and Linkable Format) — это формат как объектного, так и исполняемого файла. Он применяется во многих UNIX-подобных системах. Мы можем использовать его, чтобы откомпилировать программу для операционных систем Linux, Solaris x86, Unixware, SCO-UNIX и UNIX System V, причем не только с языка ассемблера.

Расширение объектного файла по умолчанию **.o**. Для получения файла в этом формате компилятор нужно запускать с ключом **-f elf**.

Символическая информация

Некоторые выходные форматы позволяют хранить вместе с машинными кодами символы исходного текста программы. Такая возможность очень полезна при отладке, она позволяет отладчику отображать адреса памяти в виде имен переменных, меток и т.п. Символьная информация значительно увеличивает размер выходного файла, поэтому после отладки программу заново компилируют уже без нее.

Для добавления символической информации нужно запустить NASM с ключом **-g**. Этот ключ допустим только для форматов OBJ и ELF.

Глава 10 Программирование в DOS

- Адресация памяти в реальном режиме
- Организация памяти в DOS
- Типы исполняемых файлов в DOS
- Основные системные вызовы
- Файловые операции ввода-вывода
- Управление памятью
- Коды ошибок. Отладка
- Резидентные программы

Ассемблер на примерах.
Базовый курс

«640 КБ должно быть достаточно для любого»

(Билл Гейтс 1981)

Операционная система DOS (или MS-DOS, Microsoft Disk Operating System) была первой операционной системой для компьютеров IBM PC. Ее архитектура и возможности были основаны на операционной системе CP/M, которая в то время была достаточно популярна среди 8- и 16-битных компьютеров.

Интерфейсом DOS являлась командная строка. Что же касается многозадачности, то она была однозадачной операционной системой. Файловая система не предусматривала прав доступа к файлам, которые использовались в операционной системе UNIX с самого момента ее создания. Изначально DOS располагала всего 640 Кб памяти, правда, позже появились различные способы ее расширения.

Система никогда не запускалась в защищенном режиме (поскольку была разработана для 8086), и программы могли не только управлять аппаратными средствами компьютера напрямую, но и записывать данные в любую область памяти, даже не принадлежавшую им. Процессоры 80286 и 80386 поддерживали защищенный режим. Приложения, работающие в защищенном режиме, должны были переключать процессор самостоятельно. Ясно, что нельзя было запустить несколько приложений в защищенном режиме одновременно.

10.1. Адресация памяти в реальном режиме

Процессор 8086 имел 20-битную адресную шину и поэтому способен был адресовать только 2^{20} байтов (1 Мб) оперативной памяти. Но мы знаем, что его регистры были 16-битными: как же он мог тогда адресовать 20 битов адресной шины?

Ответ находится в сегментных регистрах. Окончательный адрес, «сброшенный» на адресную шину, получался как сумма значений соответствующего 16-битного регистра и сегментного регистра, умноженного на 16.

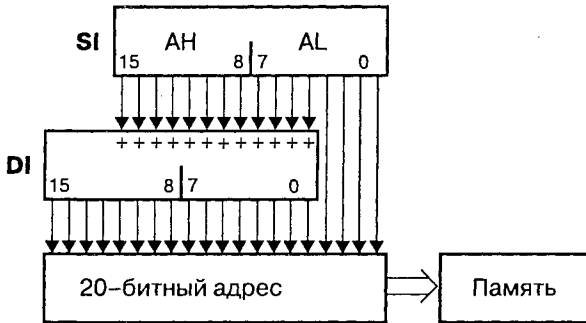


Рис. 10.1. Расчет физического адреса

Следующий пример поможет понять, как рассчитывается физический адрес:

```
mov al, [ds:si]    ; загрузить в AL содержание памяти
                   ; по адресу ds:si
```

Предположим, что $DS = 0x559E$, а SI имеет значение $0x100$.

Вычислить окончательный 20-битный адрес можно следующим образом:

$$0x559E * 0x10 \text{ (} 0x10 \text{ равно } 16 \text{)} + 0x100$$

Желаемое значение будет загружено с адреса памяти: $0x559E0 + 0x100 = 0x55AE0$.

Значение сегментного регистра называется сегментом, а другое значение (то, которое содержится в регистре общего назначения или указано как непосредственное значение) называется смещением (offset).

Заметим, что адрес $0x55AE0$ также может определяться другими парами сегментов и смещений, например, сегментом $0x558E$ и смещением $0x200$.

Адресация, использующая сегменты и смещения, имеет массу недостатков. Смещение может быть только 16-битным, и поэтому полный размер одного сегмента в памяти ограничен объемом 64 Кб. Программа, код или данные которой превышают 64 Кб, должна быть разделена на несколько сегментов.

При необходимости в сегментные регистры должны быть загружены другие адреса. Решение проблемы, которая возникает при делении программы на несколько сегментов, будет рассмотрено в параграфе 10.4.

Для сохранения совместимости даже новейшие процессоры поддерживают 16-битную адресацию в реальном режиме. Поэтому можно запустить самую древнюю DOS-программу, в том числе и саму DOS, на новейших x86-совместимых процессорах.

10.2. Организация памяти в DOS

Как мы только что объяснили, из-за реального режима процессора, операционная система DOS могла адресовать только 1 Мб оперативной памяти. Таблица 10.1. — это свод отдельных адресов и блоков памяти, выделенных для операционной системы, прикладных программ и аппаратных средств.

Таблица 10.1

Адрес (сегмент:смещение)	Что находится по адресу
0x0000:0x0000	Таблица векторов прерываний
0x0040:0x0000	Таблица переменных BIOS
0x0????:0x0000	DOS kernel (ядро системы)
0x0????:0x0000	Буферы, обработчики прерываний, драйвера устройств
0x0????:0x0000	Резидентная часть командного интерпретатора (оболочки) COMMAND.COM
0x0????:0x0000	Резидентные программы (TSR, Terminate and Stay Resident)
0x0????:0x0000	Выполняющаяся в данный момент программа (Нортон, текстовый редактор и т.п.)
0x0????:0x0000	Свободная память
0x0A000:0x0000	Начало видеопамати (в графическом режиме)
0x0B000:0x0000	Видеопамать монохромного графического адаптера (в текстовом режиме)
0x0B800:0x0000	Видеопамать графического адаптера в текстовых режимах
0x0C800:0x0000 – 0x0E000:0x0000	Свободные адреса для дополнительной памяти ROM (Read Only Memory)
0xF000:0x0000	ROM BIOS

В самом начале адресного пространства зарезервировано место для таблицы прерываний. Ее размер вычислить легко. Процессор поддерживает 256 разных векторов прерывания, а каждый вектор представляет собой пару сегмент-смещение, то есть занимает 4 байта. В итоге получается 1 Кб.

Сразу же за таблицей прерываний, по адресу 0x400 (т.е. 0x0040:0x0000), следует таблица переменных BIOS. Она содержит список устройств, подключенных к компьютеру, и их конфигурацию. Описание отдельных устройств мы рассматривать в этой книге не будем, но если кого-то интересует этот вопрос, то всегда можно найти электронный справочник Ralf Brown Interrupt List в Интернете.

В следующем блоке памяти загружена операционная система вместе с разными буферами и обработчиками прерываний. За операционной системой следует командный интерпретатор COMMAND.COM, который позволяет пользователю общаться с компьютером через интерфейс командной строки.

Далее следуют резидентные или TSR программы. Программы такого типа запускаются, как и обычные программы, но по завершении они остаются

загруженными в память. Типичная резидентная программа — это драйвер мыши, который должен оставаться в памяти, чтобы приложения могли использовать мышь.

Программный интерфейс, который обеспечивает взаимосвязь других приложений под управлением операционной системы DOS (между приложениями или между приложением и ОС), реализован через векторы прерываний. Мы рассмотрим этот вопрос более детально в параграфе «Основные системные вызовы».

Следующий блок содержит программу, работающую в данный момент. Адресное пространство этой программы простирается до предела в 640 Кб, т.е. вплоть до адреса 0x0A000:0x0000. Адреса от 640 Кб до 1 Мб зарезервированы для аппаратных средств компьютера.

Первые 64 Кб этой области используются видеоадаптером. Запись в эти адреса позволяет рисовать отдельные точки на экране.

Адреса начиная с 0x0B800:0x0000 используются видеоадаптером для хранения отдельных символов, отображаемых на экране. Адрес 0x0B800:0x0000 представляет левый верхний угол первой отображаемой строки в текстовом режиме. Данные организованы в пары байтов. Первый байт содержит ASCII-код отображаемого символа, а второй байт обозначает цвет.

Оставшееся пространство памяти зарезервировано для программ, хранящихся в памяти ROM (Read Only Memory): BIOS видеоадаптера и BIOS самого компьютера.

10.3. Расширение памяти свыше 1 МБ

Через несколько лет 640 Кб перестало хватать, и поэтому были разработаны способы расширения памяти, сохраняющие при этом реальный режим. Процессор 80286 имел 24-битную адресную шину, однако в реальном режиме он все еще мог адресовать только 20 битов, т.е. 1 Мб.

Последним пригодным для использования сегментом был 0x0F000, в котором хранится BIOS. Чтобы адресовать следующий за BIOS сегмент, нужно было бы загрузить в сегментный регистр значение 0xFFFF. Но после добавления к этому значению смещения адрес вышел бы за пределы 20 битов, то есть превысил бы ограничение в 1 Мб.

Разработчики решили оборудовать компьютер специальным устройством для контроля над двадцатым первым битом адресной шины, которое позволяло бы использовать сегмент 0x0FFFF. Процессор получит доступ к памяти выше 1 Мб, если «что-то извне» установит 21-й бит адресной шины в значение 1. Этим «чем-то» стало устройство, названное A20 (21-й бит имеет номер 20, потому что нумерация начинается с нуля). A20 управляет вышеупомянутым 21-м битом и позволяет использовать дополнительные 64 Кб оперативной памяти.

Однако таким способом можно добавить только 64 Кб памяти, что не так уж много. В диапазоне от 640 Кб до 1 Мб имеется масса неиспользованной памяти, зарезервированной для ROM-системы. Нельзя ли как-нибудь использовать эти адреса?

Конечно же, можно. Консорциум из трех компаний — Lotus, Intel и Microsoft — сформулировал спецификацию памяти, которую назвали LIM EMS (Expanded Memory Specification). В области между 640 Кб и 1 Мб было создано окно памяти с размером 64 Кб, которое состояло из четырех страниц, по 16 Кб каждая. Это окно называется кадром страницы (page frame), потому что EMS позволяет отображать в это окно память за пределами 1 Мб. У компьютеров типа 80286 EMS-память реализована с помощью аппаратных средств, но позже благодаря процессору 80386 и его виртуальному реальному режиму она могла реализовываться программно.

Последним способом получения доступа к памяти за пределами 1 Мб в реальном режиме стала спецификация XMS (Extended Memory Specification). Этот интерфейс был реализован при помощи драйвера HIMEM.SYS, который при необходимости перемещает блок памяти выше 1 Мб в память ниже 1 Мб и наоборот. При обработке запроса он переключает процессор в защищенный режим, где для адресации доступна вся память. После перемещения блока, процессор переключается обратно в реальный режим.

10.4. Типы исполняемых файлов в DOS

Операционная система DOS поддерживает три типа бинарных исполняемых файлов.

Первый тип — это системный драйвер устройства, который обычно имеет расширение .SYS. Файл такого типа загружается при запуске системы. Список всех загружаемых драйверов устройств содержится в конфигурационном файле CONFIG.SYS. Описание этого файла и его функций выходит за пределы этой книги.

Следующие два типа исполняемых файлов — COM и EXE (расширения .COM и .EXE соответственно) — предназначены для прикладных программ.

Исполняемый файл типа COM в отличие от других запускаемых файлов не имеет заголовка. Его максимальный размер — 64 Кб, из которых 256 байтов занимает префикс программного сегмента. Во время загрузки программы операционная система выделяет ей новый сегмент памяти (т.е. 64 Кб). Программа загружается с диска по адресу 0x100.

В ходе запуска программы операционная система загружает адрес выделенного ей сегмента в сегментные регистры (CS, ES, DS и SS), устанавливает указатель вершины стека SP почти в конец сегмента — на адрес 0x0FFFE, а затем передает управление на смещение 0x100, где находится первая команда исполняемого файла.

Диапазон адресов от 0 до 0x100 зарезервирован для операционной системы. Он называется PSP (Program Segment Prefix). Здесь операционная система сохраняет среди прочего информацию об открытых файлах и о PSP-адресе родительского процесса. Начиная со смещения 0x80 хранятся аргументы, с которыми запущена программа.

Рассмотрим карту памяти COM-программы, загруженной в сегмент 0x1234 (см. табл.10.2):

Карта памяти COM-программы

Таблица 10.2

Адрес (сегмент:смещение)	Что находится по адресу
0x1234:0x0000	Начало области, зарезервированной для ОС. Здесь хранятся некоторые внутренние структуры
0x1234:0x0080	С этого адреса начинаются переданные параметры
0x1234:0x0100	С 0x100 операционная система загружает COM-файл с диска
0x1234:0x????	Окончание сегмента (секции) кода
0x1234:0x????	Окончание сегмента (секции) данных
0x1234:0x????	Начало секции неинициализированных данных (.bss)
....	Память распределена до конца сегмента, а стек растет к началу сегмента
0x1234:0xFFFF	Вершина стека

Тип исполняемого файла EXE существенно улучшен по сравнению с типом COM. При этом главное улучшение заключается в том, что тип EXE пригоден для размещения программ, размер которых больше 64 Кб. Файл типа EXE может содержать несколько сегментов кода и несколько сегментов данных. У стека также есть зарезервированные сегменты.

Чтобы обратиться к другому сегменту в реальном режиме, нужно переключить значения сегментных регистров. Так, при вызове подпрограммы, сохраненной в другом сегменте, мы должны использовать инструкцию дальнего вызова (call far), которая правильно изменяет значения обоих регистров (CS и IP). Точно так же должен быть выполнен возврат (retf). Если переменная сохранена в другом сегменте данных, мы получаем адрес ее сегмента при помощи псевдокоманды SEG и загружаем ее в сегментный регистр. Правильный адрес получается в результате сложения смещения, которым заведует идентификатор, с новым значением сегмента.

Разработанная схема действительно хороша, если бы не одно «но». Как получить адрес сегмента, если мы не знаем стартовые адреса сегментов (т.е. программы)?

Скомпилированная программа сохраняется на диск так, как если бы ей предстояло быть загруженной с адреса 0x0000:0x0000, то есть первый сегмент начинается с 0x0000:0x0000 и заканчивается в 0x0000:0xFFFF, второй сегмент начинается с 0x1000:0x0000 и так далее.

Каждое место программы, в котором используются значения сегментов (т.е. 0x0000, 0x1000 и т.д.), записывается в так называемую таблицу перемещений (relocation table).

Программа загружается в память с некоторого базового адреса. По таблице перемещений определяются те места программы, содержимое которых должно быть «подправлено», и к хранящимся там адресам добавляется значение базового адреса. Таким образом программа целиком перемещается к базовому адресу загрузки. Теперь все сегментные регистры содержат правильные значения, и программа может быть запущена.

10.5. Основные системные вызовы

Операционная система предлагает запущенной программе множество сервисов. Используя операционную систему, наша программа может выводить сообщения на экран, считывать ввод с клавиатуры, работать с файлами или требовать выделения дополнительной памяти. Сервисы операционной системы доступны через прерывание 0x21. Параметры системных вызовов указываются в определенных регистрах. Под DOS номер необходимой функции указывается в регистре AH. Если системный вызов завершился ошибочно, то устанавливается флаг переноса, а в регистр AX помещается код ошибки.

Немедленное завершение программы

Простейшая программа — это та, которая завершается немедленно после запуска. Запрос на завершение программы обеспечивается системным вызовом DOS с номером 0x4C, прерывающим программу и передающим управление родительской программе.

Ввод:	Вывод:
Завершение программы: AH = 0x4c AL = код возврата	Ничего

Номер вызываемой функции передается через регистр AH, а в AL указывается так называемый код возврата, то есть значение, которое должна анализировать родительская программа. Дочерняя программа может либо сообщить родительской, что завершилась нормально, либо вернуть код ошибки. Эта функция такого же назначения, как **exit** в языке C.

Наша первая программа будет включать в себя только раздел кода (листинг 10.1).

Листинг 10.1. Простейшая программа под DOS

```
SECTION .text
;Наша первая программа для операционной системы DOS.
;Она ничего не делает, только завершается сразу после
;запуска
org 0x100                ;генерируем COM файл, загружаемый с 0x100
```

```

mov ah,0x4C      ;эта функция завершает программу
mov al,0         ;код возврата 0
int 0x21        ;вызываем ядро операционной системы
;конец программы

```

Теперь сохраните программу в файл `finish.asm`. В этом и следующих примерах мы будем компилировать выходные файлы типа COM, чтобы избежать запутанной процедуры настройки сегментных регистров и сконцентрироваться на решении других проблем.

Следующим шагом будет компиляция `finish.asm` компилятором **nasm**:

```
nasm -f bin -o finish.com finish.asm
```

Параметр **-f** указывает тип выходного формата, а параметр **-o** указывает на создание файла с заданным именем `finish.com`.

После компиляции мы можем запустить программу `finish.com`, написав «finish» в командной строке и нажав «Enter». Программа сразу же вернет управление командному интерпретатору.

Вывод строки. Пишем программу «Hello, World!»

Другая очень полезная функция — это вывод текстовой строки на экран. Сейчас мы не будем рассматривать запись в файл, а рассмотрим разные функции вывода строки на экран.

По историческим причинам выводимая строка оканчивается не нулевым байтом, а знаком доллара «\$» (0x24).

Ввод:	Вывод:
Печать строки: AH = 0x09 DS:DX = указатель на строку, завершённую «\$»	Ничего

Печать одного символа на стандартный вывод выполняет функция DOS с номером 0x02, которая считывает код ASCII необходимого символа из DL.

Ввод:	Вывод:
Печать символа: AH = 0x02 DL = ASCII-код печатаемого символа	Ничего

Теперь у нас достаточно знаний, чтобы написать программу, которая выведет на экран всем известный текст «Hello, World!» и завершит свою работу.

Сейчас мы с вами отредактируем нашу программу `finish.asm`. Будет добавлена новая секция `.data` с описанием переменной «hello», содержащая строку, состоящую из текста «Hello, World!», служебных символов CR (0xD) и LF (0xA), которые во время вывода на экран выполняют перевод строки, и завершающего символа «\$» (листинг 10.2).

Листинг 10.2. Программа Hello, World! под DOS

```
SECTION .text
;Эта программа выводит на экран текст "Hello, World!"
;конец строки (EOL – End Of Line) и завершает работу
org 0x100          ;создаем файл типа COM
mov ah,0x9         ;функция DOS для печати строки
mov dx,hello       ;загружаем в DX указатель на строку
int 0x21           ;вызываем DOS
mov ah,0x4C        ;функция DOS для завершения программы
mov al,0           ;код возврата 0
int 0x21           ;вызов ядра операционной системы
SECTION .data
hello DB "Hello, World!",0xd,0xa,'$'
;конец программы
```

Нам не нужно устанавливать сегментный регистр DS: операционная система установит его сама.

Эту программу назовем `hello.asm` и скомпилируем командой

```
nasm -f bin -o hello.com hello.asm.
```

Запустим программу с помощью команды `hello`.

Ввод с клавиатуры

Для последовательного считывания символов используется функция DOS с номером `0x01`, которая похожа на `getchar` в языке C и на `readkey` в языке Pascal.

Ввод:	Вывод:
Чтение символа: AH = 0x01	AL содержит символ, считанный со стандартного ввода (клавиатуры)

Функция 0x01 ждет нажатия клавиши. После получения символа она отражает символ на стандартном выводе (функция `0x08` только считывает символ, но не отражает его).

В регистре AL окажется значение 0, если была нажата клавиша с так называемым расширенным (extended) ASCII-кодом, то есть одна из клавиш Page Up, Page Down, Home, End, функциональные клавиши F1..F12 и т.п. Следующий вызов функции `0x01` поместит в AL расширенный ASCII-код нажатой клавиши.

Часто нам нужно прочесть целую строку определенной длины, завершенную нажатием клавиши «Enter». Для этого в DOS имеется функция, накапливающая прочитанные с клавиатуры символы в буфере.

Ввод:	Вывод:
Считывание строки: AH = 0x0A DS:DX = указатель на буфер в специальном формате	В буфере находится прочитанная строка

Перед вызовом функции 0x0A указатель должен содержать адрес специальной структуры, которая в результате работы функции будет заполнена прочитанной строкой.

Первый байт буфера содержит его максимальный размер, то есть количество символов (1–254), по достижении которого выдается звуковой сигнал (ASCII-код 7) и требуется нажать «Enter». Следующий байт содержит фактическое количество введенных символов, не считая символа 0x0D (кода клавиши «Enter»).

```
mov ah, 0x0A      ;загрузить в AH 0x0A, функция чтения строки
mov dx,string     ;загрузить в DX адрес (смещение) буфера
int 0x21          ;вызов DOS
...
string db 5,0,0,0,0,0,0
mov ah, 0x0A      ;загрузить в AH номер функции чтения строки
mov dx,string     ;загрузить в DX адрес (смещение) буфера string
int 0x21          ;вызов DOS
...
string db 5,0,0,0,0,0,0
```

После выполнения команды **int 0x21** операционная система начнет чтение символов и будет продолжать чтение до нажатия клавиши «Enter». Под буфер отведено 7 байтов, то есть в нем можно разместить не более 4 символов плюс символ завершения — код клавиши «Enter». Если мы наберем A,B,C, а затем нажмем «Enter», то структура «string» будет выглядеть следующим образом:

```
string db 5,3,0x65,0x66,0x67,0x0D,0
```

Первый байт остается неизменным, второй байт содержит длину считанной строки без символа окончания 0xD, а с третьего байта начинается сама строка (ABC), завершающаяся символом окончания 0xD.

Рассмотрим несколько примеров.

Пример 1: напишем программу, которая будет отображать каждую нажатую клавишу на экране дважды. Программа завершается по нажатии клавиши «Enter» (листинг 10.3).

Для чтения символов мы будем использовать DOS-функцию 0x01, отображающую нажатую клавишу на экране. Следующий, идентичный предыдущему, символ будем выводить DOS-функцией 0x02.

Листинг 10.3. Программа, отображающая дважды на экране каждую нажатую клавишу

```
; Двойное отображение каждой нажатой клавиши на экране
SECTION .text
again:
mov ah,0x01          ;DOS-функция чтения символа
int 0x21             ;вызов DOS
mov dl,al            ;копирование считанного символа в DL
cmp dl, 0xD          ;нажат "Enter"?
jz endprog           ;если да, то переход к концу программы
mov ah, 0x02         ;DOS-функция вывода символа
int 0x21             ;вызов DOS
jmp again            ;вернуться к чтению следующего символа
endprog:
mov ah,0x4C          ;DOS-функция завершения программы
int 0x21             ; вызов DOS
                    ;конец программы
```

Эту программу назовем `echo.asm`. Исполняемый файл `echo.com` будет создан командой

```
nasm -f bin -o echo.com echo.asm
```

Пример 2: напишем программу, считывающую строку символов до нажатия клавиши «Enter» и выводящую эту строку в обратном порядке (листинг 10.4).

Используя DOS-функции `0x0A`, мы считываем строку и сохраняем ее во временной переменной, которая после будет выведена в обратном порядке с помощью DOS-функции `0x02`.

Листинг 10.4. Программа, считывающая строку символов до нажатия клавиши «Enter» и выводящая эту строку в обратном порядке

```
SECTION .text
;Эта программа выводит прочитанную строку
;на экран в обратном порядке.
org 0x100            ;создаем файл типа COM
mov ah, 0x0A         ;DOS-функция чтения строки
mov dx,string        ;загружаем в DX адрес буфера строки
int 0x21             ;вызов DOS
xor ax,ax            ;обнуляем AX
mov al,[string+1]    ;читаем количество введенных символов
inc dx               ;пропускаем первый байт буфера
add dx,ax             ;прибавляем длину строки к адресу начала,
                    ;получаем адрес последнего символа
mov si,dx             ;копируем DX в SI
```

```

std                                ;устанавливаем флаг направления для
                                ;прохода обратно
print_next_char:
lodsb                             ;читаем символ из DS:SI и уменьшаем SI на 1
cmp si,string+1                   ;конец строки? (то есть начало)
jb endprog                        ;если да, то переход к концу программы
mov dl,al                         ;загружаем прочитанный из строки символ в DL
mov ah,0x02                       ;DOS-функция вывода символа
int 0x21                          ;вызов DOS
jmp print_next_char               ;возвращаемся к следующему символу
endprog:
mov ah,0x4C                       ;DOS-функция завершения программы
mov al,0                          ;код возврата 0
int 0x21                          ;вызов DOS
SECTION .data
string db 254,0
times 253 db ' '                  ;заполняем буфер строки пробелами
;конец программы

```

Назовем эту программу `reverse.asm` и скомпилируем ее так же, как и предыдущую:

```
nasm -f bin -o reverse.com reverse.asm
```

Эта программа немного сложнее, поэтому объясним ее функционирование шаг за шагом.

Прежде всего, мы должны прочитать строку. Для этого мы вызываем функцию `0x0A` для чтения строки, передав ей подготовленный в специальном формате буфер для размещения считанной строки.

```

mov ah, 0x0A                      ;DOS-функция чтения строки
mov dx,string                      ;загружаем в DX адрес буфера строки
int 0x21                          ;вызов DOS

```

После того, как пользователь нажмет «Enter», наша программа продолжит выполнение. Поскольку строка должна быть выведена в обратном порядке, то прежде всего мы должны вычислить адрес последнего символа.

```

xor ax,ax                         ;обнуляем AX
mov al,[string+1]                 ;читаем количество введенных символов

```

После возврата из системного вызова регистр `DX` все еще содержит адрес буфера «string». Теперь нам необходимо узнать адрес (смещение) конца строки, который равен `DX + 2 + длина_строки — 1` (в программах типа `COM` нам не нужно рассматривать весь адрес — сегментная часть одинакова для всех секций программы, и поэтому достаточно изменения смещения).

Начало строки находится по адресу `DX+2` и, добавив длину строки, мы получим символ `0x0D` (клавиша «Enter»). Поскольку он нам не нужен, мы отнимаем единицу.

Конечно же, вместо прибавления 2 и вычитания 1 можно просто прибавить 1, сократив программу на одну команду. Заметьте, что вначале нужно сбросить AX и что длина строки — число восьмибитовое.

```
inc dx          ; пропускаем первый байт буфера
add dx, ax      ; прибавляем длину строки к адресу начала,
                ; получаем адрес последнего символа
mov si, dx      ; копируем DX в SI
```

Несмотря на то, что длина строки — число восьмибитовое, мы прибавляем к адресу начала строки не AL, а AX, потому что если один операнд представляет собой адрес или смещение, то второй должен быть такого же размера.

В результате получается адрес последнего символа в строке. Команда LODSB выводит символ, адрес которого хранится в регистре SI, поэтому скопируем вычисленный адрес в этот регистр. Для получения следующего в обратном порядке символа нужно уменьшить этот адрес на 1: значит, нужно установить флаг направления.

```
std            ; устанавливаем флаг направления для
                ; прохода обратно
print_next_char:
lodsb         ; читаем символ из DS:SI и уменьшаем SI на 1
```

LODSB загружает в AL символ, считанный из [DS:SI], и уменьшает SI на 1 после считывания. Сразу же после LODSB следует сравнение с адресом string+1, который содержался бы в SI после чтения символа на последнем шаге (то есть первого символа строки). Если бы мы использовали условный переход по равенству адресов (JZ), то первый символ строки не выводился бы, поэтому мы написали команду «перейти, если меньше». Это условие станет истинным не до, а после вывода первого символа.

```
cmp si, string+1 ; конец строки?
jb endprog       ; если да, то переход к концу программы
```

Оставшаяся часть программы предельно проста. Символ, сохраненный в AL, копируется в DL, как того требует функция DOS для вывода символа. После вывода мы должны вернуться к LODSB, чтобы извлечь из строки следующий символ.

```
mov dl, al      ; загружаем прочитанный из строки символ
                ; в DL
mov ah, 0x02    ; DOS-функция вывода символа
int 0x21        ; вызов DOS
jmp print_next_char ; возвращаемся к следующему символу
```

Выполнение программы завершается системным вызовом 0x4C.

```
endprog:
mov ah, 0x4C    ; DOS-функция завершения программы
mov al, 0       ; код возврата 0
int 0x21        ; вызов DOS
```

Секция данных описывает структуру буфера.

```
SECTION .data
string db 254,0
times 253 db ' ' ;заполняем буфер строки пробелами
```

Первый байт определяет максимальную длину строки, не считая завершающего символа. Второй байт содержит фактическую длину прочитанной строки (ведь строка может быть и короче, чем 254 символа), и на данный момент она составляет 0 байтов. Используя директиву TIMES, оставшуюся часть структуры заполняем 253 пробелами.

10.6. Файловые операции ввода-вывода

В главе 8, посвященной операционным системам, мы говорили о технике работы с файловой системой. Операционная система DOS работает по такому же принципу. Перед какой-либо операцией ввода-вывода файл должен быть открыт. От операционной системы необходимо получить идентификатор (дескриптор) файла, который нужно указывать при вызове других функций для работы с файлами. По окончании процесса обработки файл должен быть закрыт.

Оригинальная система DOS до слияния с ОС Windows позволяла использовать только короткие имена файлов: 8 символов в имени и 3 символа в расширении файла. В Windows появились так называемые длинные имена файлов, состоящие из 256-и символов в имени, включая несколько расширений. Давайте начнем с системных вызовов, использующих короткие имена файлов 8+3.

Открытие файла

Файл открывает функция **0x3D**. Указатель на имя файла передается через пару регистров DS:DX (DS — сегмент, а DX — смещение). В регистр AL нужно записать требуемый режим доступа. В большинстве случаев файл открывается для чтения и записи (AL=0). Функция возвращает в AX дескриптор открытого файла или код ошибки. При ошибочном завершении функция устанавливает флаг переноса CF=1, при корректном завершении CF=0.

Ввод:	Вывод:
Открытие файла AH = 0x3D DS:DX = указатель на имя файла, завершённое нулевым байтом 0x0 AL = режим доступа: AL = 0 чтение и запись AL = 1 только запись AL = 2 только чтение	CF = 0 если успешно, тогда AX = дескриптор, назначенный файлу Или CF = 1 ошибка, тогда AX = код ошибки: AX = 0x0002 — файл не найден AX = 0x0003 — путь не найден

Заккрытие файла

Прежде чем переходить к другим функциям, познакомимся с функцией за-
крытия файла 0x3E:

Ввод:	Вывод:
Заккрытие файла AH = 0x3E BX = дескриптор файла	CF = 0 если успешно Или CF = 1 ошибка, тогда AX = код ошибки

Чтобы закрыть файл, поместите его дескриптор, присвоенный файлу при
открытии, в регистр BX. Напишем простую программку для открытия и за-
крытия файлов (листинг 10.5).

Листинг 10.5. Простая программка-пример
для открытия и закрытия файлов

```
SECTION .text
org 0x100
mov ax, 0x3D00      ;функция DOS для открытия файла, режим
                    ;чтение-запись
mov dx,file_name    ;передаем указатель на имя файла, DS уже
                    ;установлен
int 0x21            ;вызов DOS
jc error            ;ошибка? если да, перейти к метке error
mov bx,ax           ;AX содержит дескриптор файла,
                    ;копируем его в BX
mov ah, 0x3E        ;функция DOS для закрытия файла
int 0x21            ;вызов DOS
mov al,0            ;устанавливаем код возврата: успешное
                    ;завершение

endprog:
mov ah,4Ch          ;функция DOS для завершения программы
int 0x21            ;вызов DOS
error:
mov al,1            ;устанавливаем код возврата: ошибка
jmp short endprog   ;переходим к метке окончания программы
                    ;endprog

SECTION .data
file_name db "text.txt",0 ;имя файла text.txt
```

Чтение из файла

Очевидно, что приведенная в листинге 10.5 программа абсолютно бесполезна,
поэтому давайте усовершенствуем ее: мы будем выводить содержимое файла
на экран. Чтение данных из файла выполняется DOS-функцией 0x3F.

Данные могут читаться из файла блоками произвольного размера. Следующий системный вызов считывает следующий блок (внутренний указатель позиции файла увеличивается после операции чтения), то есть для чтения всего файла нам нужен цикл.

Ввод:	Вывод:
Чтение из файла: AH = 0x3F BX = дескриптор файла DS:DX = указатель на буфер для хранения прочитанных данных CX = размер буфера	В случае успеха: CF = 0 AX = фактическое количество байтов, считанных из файла (ноль означает конец файла, EOF) В случае ошибки: CF = 1 AX = код ошибки

Функции нужно передать дескриптор файла в регистре BX. Указатель на буфер, в который нужно поместить прочитанный блок данных, передается через пару DS:DX. Через регистр CX передается максимальное количество байтов, которое должно быть считано за раз: большую часть времени оно равно длине буфера. В регистр AX функция записывает количество фактически прочитанных байтов (ведь если в файле осталось меньше данных, чем мы хотим прочитать, значение в AX будет меньше размера буфера).

В случае ошибки устанавливается флаг переноса CF=1, а в AX будет код ошибки.

Данные, прочитанные из файла, могут быть выведены на экран с помощью функции DOS 0x09. Но тогда у нас будут проблемы с любым файлом, содержащим символ \$, который обозначает конец строки для функции 0x09. Чтобы обойти эту проблему, мы не будем использовать функцию 0x09, а вместо вывода на экран будем записывать прочитанные данные в поток стандартного вывода, а для вывода на экран используем функцию перенаправления ввода/вывода, предоставленную нам операционной системой. Дескриптор файла стандартного вывода равен 0x0001.

Запись в файл

Для записи в файл используется функция 0x40. Она имеет те же аргументы, что и функция чтения файла:

Ввод:	Вывод:
Запись в файл: AH = 0x40 BX = дескриптор файла DS:DX = указатель на буфер CX = количество байтов, которые нужно записать	В случае успеха: CF = 0 AX = фактическое количество байтов, записанных в файл При ошибке: CF = 1 AX = код ошибки

В DS:DX нужно указать адрес буфера, содержащий данные, которые нам нужно записать в файл. А в CX нужно занести количество байтов, которые нужно записать: обычно это длина буфера.

Если вызов прошел успешно, то в AX окажется количество байтов, записанных в файл. Если оно отличается от требуемого, то, возможно, нет свободного места на диске.

Теперь у нас достаточно знаний, чтобы вывести текстовый файл text.txt на экран.

Прежде всего, нам необходимо открыть файл text.txt, используя функцию 0x3D. Потом мы будем читать из него блоки одинакового размера при помощи функции 0x3F, пока количество считанных байтов не станет равно нулю. Прочитанный блок мы будем выводить на стандартный вывод функцией 0x40. Перед завершением программы мы закроем файл.

Листинг 10.6. Программа вывода текстового файла на экран

```
%define B_LENGTH 80      ;определяем константу – размер блока
%define STDOUT 0x0001    ;дескриптор стандартного вывода
SECTION .text
org 0x100
mov bp,STDOUT             ;пока храним дескриптор файла
                           ;стандартного вывода
                           ;в отдельном регистре
mov ax, 0x3D00            ;открываем файл в режиме чтения-
                           ;записи
mov dx,file_name          ;указатель на имя файла. DS уже
                           ;установлен
int 0x21                 ;вызов DOS
jc error                 ;ошибка? Если да, переход к метке error
mov bx,ax                ;копируем дескриптор открытого
                           ;файла в BX

read_next:
mov ah,0x3F              ;функция DOS для чтения из файла
mov dx,buffer            ;буфер для хранения блока данных
mov cx,B_LENGTH          ;размер блока
int 0x21                 ;вызов DOS
jc error                 ;ошибка? Если да, переход к метке error
or ax,ax                 ;прочитано 0 байтов? Можно
                           ;использовать стр ax,0

jz end_reading           ;если да, переход к концу чтения
mov cx,ax                ;на последнем шаге количество
                           ;прочитанных байтов может быть
                           ;меньше CX, а записать нужно столько
                           ;же байтов, сколько было считано
```

```

mov ah,0x40          ;функция DOS для записи в файл
xchg bp,bx           ;временно сохраняем дескриптор
                     ;файла в BP, а
                     ;дескриптор стандартного вывода
                     ;из BP кладем в BX
int 0x21             ;вызов DOS
xchg bp,bx           ;возвращаем исходные значения BX и BP
jmp read_next        ;переход к метке read_next, читаем
                     ;следующий блок

end_reading:
mov ah, 0x3E         ;функция DOS для закрытия файла
int 0x21             ;вызов DOS
mov al,0             ;код возврата: успешное завершение
endprog:
mov ah,4Ch           ;функция DOS для завершения
                     ;программы
int 0x21             ;вызов DOS. Это последняя команда
                     ;программы

error:
mov al,1             ;устанавливаем код возврата: ошибка
jmp short endprog    ;переходим к метке endprog
SECTION .data
file_name db "text.txt",0 ;определяем имя файла text.txt
SECTION .bss
buffer RESB B_LENGTH ;определяем неинициализированную
                     ;переменную
                     ;buffer размером B_LENGTH байтов

```

Назовем наш файл `listit.asm` и откомпилируем его с помощью команды

```
nasm listit.asm -o listit.com -f bin
```

Теперь создадим текстовый файл `text.txt` с помощью текстового редактора или прямо из командной строки:

```
echo I'll BE BACK! > text.txt
```

После запуска программы текст из файла `text.txt` будет выведен на экран.

Мы можем модифицировать нашу программу так, чтобы вместо записи на стандартный вывод она записывала прочитанные данные в другой файл. Тогда мы получим простейшую программу копирования файла. Но мы до сих пор не знаем, как создать файл!

Открытие/создание файла

В отличие от функции 0x3D функция 0x6C не только открывает файл, но и создает новый файл, если при открытии файла обнаружится, что файла с таким именем не существует.

Ввод:	Вывод:
Открытие/создание файла AH = 0x6C AL = 0x00 (всегда должен быть 0) BX = массив битов для расширенного режима CX = атрибуты для нового файла DX = массив битов для требуемой операции DS:DX = указатель на имя файла	Успех: CF = 0 AX = идентификатор открытого файла CX = отчет об операции (см. ниже) Ошибка: CF = 1 AX = код ошибки

Данная функция позволяет использовать несколько режимов открытия файлов: открытие существующего файла, создание нового файла с его последующим открытием и открытие файла с усечением его длины до нуля (открытие для перезаписи).

Требуемый режим мы можем указать в регистре DX. Можно использовать следующие комбинации битов при системном вызове.

4 младших бита (в регистре DL):

- ♦ DL = 0000b — выйти с ошибкой, если файл существует (можно использовать для проверки существования файла).
- ♦ DL = 0001b — открыть файл, если он существует.
- ♦ DL = 0010b — открыть файл для перезаписи (удалить все содержимое), если он существует.

4 старших бита (в регистре DH):

- ♦ DH = 0000b — выйти с ошибкой, если файл не существует.
- ♦ DH = 0001b — создать файл, если он не существует.

Когда мы будем вызывать функцию 0x6C в программе копирования файла, нам нужно записать в DX значение 0x0012. В этом случае файл назначения будет создан и открыт при условии, что он не существует или усечен и открыт, если файл существует.

В регистр CX (это тоже массив битов) нужно занести атрибуты вновь создаваемого файла. DOS — не UNIX, она не поддерживает ни имени владельца, ни прав доступа, поэтому набор атрибутов в DOS крайне ограничен:

- ♦ Бит 0: если 1, то будет установлен атрибут «только чтение».
- ♦ Бит 1: если 1, то будет установлен атрибут «скрытый».
- ♦ Бит 2: если 1, то будет установлен атрибут «системный».
- ♦ Бит 3: если 1, то это метка тома.

- ♦ Бит 4: если 1, то это каталог.
- ♦ Бит 5: если 1, то будет установлен атрибут «архивный».
- ♦ Биты 6–15: зарезервированы для дальнейшего использования.

В большинстве случаев в регистр CX нужно помещать значение 0x20: создание обычного архивного файла.

В регистре BX указываются флаги расширенного режима. Для нас пока имеют значение только два младших бита, рассмотрение остальных выходит за пределы этой книги.

- ♦ BX = 0: файл открывается только для чтения.
- ♦ BX = 1: файл открывается только для записи.
- ♦ BX = 2: файл открывается для чтения и для записи.

После успешного завершения системного вызова (CF=0) регистр AX будет содержать дескриптор файла (как и в случае с функцией 0x3D). В CX будет возвращен «отчет» о выполненной операции:

- ♦ CX = 1: файл был открыт.
- ♦ CX = 2: файл был создан и открыт.
- ♦ CX = 3: файл был перезаписан и открыт.

Модифицируем нашу программу listit.asm (листинг 10.6) так, чтобы она копировала файлы. Файл назначения будем создавать (или перезаписывать) функцией 0x6C. Осталось только передать полученный из нее дескриптор файла назначения в то место программы, где происходит запись в файл. Таким образом, допишем в начало нашей программы строки для открытия второго файла:

```
mov ax, 6C00h          ;функция DOS для создания/открытия файла
                        ;заполняем сразу оба регистра:
                        ;0x6C в AH и 0 в AL
mov cx, 0x20           ;атрибуты нового обычного файла
mov dx, 0x12           ;операция открытия
mov si, other_file_name;адрес имени второго файла
int 0x21               ;вызов DOS
jc error               ;переход к обработке ошибки
mov bp, ax              ;сохраняем дескриптор открытого файла в BP
```

Ясно, что этот фрагмент должен занять место команды MOV bp,STDOUT.

Второй файл тоже должен быть закрыт: добавим после закрытия первого файла следующие строки:

```
mov ah, 0x3e           ;функция DOS для закрытия файла
mov bx, bp              ;помещаем дескриптор файла в BX
int 0x21
```

И, наконец, определим имя второго файла в секции данных:

```
other_file_name db "text1.txt",0    ;имя второго файла
```


Наша программа копирует первый файл `text.txt` во второй файл `text1.txt`. Первый файл читается через свой дескриптор, хранящийся в `BX`, а второй файл записывается через свой дескриптор, который после его открытия сохранен в `BP`.

Назовем нашу программу `copy.asm` и откомпилируем ее как обычно:

```
nasm -f bin -o copy.com copy.asm
```

Поиск позиции в файле (SEEK)

До сих пор мы читали и записывали файлы последовательно, то есть в том порядке, в котором они записаны на диске. В более сложных случаях нам нужно читать файлы выборочно (например, 1 блок в начале файла и 2 блока в конце файла), передвигая указатель внутренней позиции файла вперед/назад. В языке `C` для этого служит функция `seek()`, а в `DOS` имеется системный вызов `0x42`.

Ввод:	Вывод:
Изменение позиции в файле AH = 0x42 AL = начало отсчета BX = дескриптор файла CX:DX = смещение	Успех: CF = 0 DX:AX = новая позиция CX = отчет об операции Ошибка: CF = 1 AX = код ошибки

Дескриптор файла передается через регистр `BX`. Требуемое значение смещения внутреннего указателя передается через пару `CX:DX`. Два регистра, а не один, нужны для того, чтобы можно было адресовать файлы размером до 4 Гб (размер файла в `FAT16` ограничен 2 Гб). Регистр `CX` содержит старшие 16 битов, а `DX` — младшие 16 битов смещения.

Регистр `AL` определяет начало отсчета, то есть от какого места в файле будет отсчитываться заданное в `CX:DX` смещение:

- `AL = 0`: смещение отсчитывается от начала файла (`SEEK_SET`), и новое значение позиции будет равно `CX:DX`.
- `AL = 1`: смещение отсчитывается от текущей позиции (`SEEK_CUR`).
- `AL = 2`: смещение отсчитывается от конца файла (`SEEK_END`).

Новая позиция внутреннего указателя будет возвращена в паре `DX:AX`.

Функция `SEEK` может также быть использована для определения размера файла. Откройте файл и вызовите функцию `0x42` со следующими параметрами:

```
mov ax,0x4202      ;функция поиска позиции от конца файла
mov bx,filedes     ;BX = дескриптор файла
xor dx,dx           ;DX = 0
xor cx,cx           ;CX = 0
int 0x21            ;вызываем DOS
```

В результате отсчета нулевого смещения от конца файла в паре DX:AX окажется длина файла.

Другие функции для работы с файлами

Рассмотрим функции удаления, изменения атрибутов и переименования файла.

Файл на диске мы можем удалить с помощью функции 0x41:

Ввод:	Вывод:
Удаление файла AH = 0x41 DS:DX = указатель на имя файла	Успех: CF = 0 Ошибка: CF = 1 AX = код ошибки

Если у файла установлен атрибут «только чтение», сначала нужно снять этот атрибут, иначе мы не сможем удалить его. **Изменить атрибуты можно с помощью 0x43:**

Ввод:	Вывод:
Получение атрибутов AH = 0x43 AL = 0x00 DS:DX = указатель на имя файла ИЛИ: Установка атрибутов CX = требуемые атрибуты AL = 0x01 DS:DX = указатель на имя файла	Успех: CF = 0 CX = атрибуты файла Ошибка: CF = 1 AX = код ошибки

Переименование файла осуществляется с помощью системного вызова 0x56.

При переименовании файла мы также можем переместить его в новый каталог, но при условии, что целевой каталог находится на том же логическом диске, что и исходный файл.

Ввод:	Вывод:
Переименование файла AH = 0x56 DS:DX = указатель на исходное имя файла ES:DI = указатель на новое имя файла	Успех: CF = 0 Ошибка: CF = 1 AX = код ошибки

Рассмотрим пример: переместим файл text.txt из текущего каталога в корневой каталог диска. Мы будем использовать системный вызов перемещения файла. Программа очень проста и состоит всего из двух системных вызовов: переименование файла и завершение работы (листинг 10.7).

**Листинг 10.7. Программа перемещения файла
из текущего каталога в корневой**

```

SECTION .text      ;секция кода
org 0x100
mov ah, 0x56        ;функция DOS 0x56 для переименования файла
mov dx,src          ;загружаем в DX адрес исходного имени файла
mov di,dest         ;а в DI – адрес нового имени файла
int 0x21
mov ax,0x4c00       ;завершение программы
int 0x21
SECTION .data
src db "text.txt",0
dest db "\text.txt",0

```

Назовем программу `rename.asm` и откомпилируем как обычно. Ее размер будет всего несколько байтов, но ее можно сделать еще меньше. Имена файлов отличаются только символом обратного слэша, обозначающим корневой каталог диска. Для уменьшения размера программы перепишем секцию данных так:

```

SECTION .data
dest db "\"
src db "text.txt",0

```

Теперь оба имени «делают» одну и ту же область памяти. В высокоуровневых языках программирования, как правило, таких фокусов позволить себе нельзя.

Длинные имена файлов и работа с ними

Windows 95 позволяет преодолеть ограничение 8+3 на длину файлов. Для обеспечения обратной совместимости со старыми программами длинные имена хранятся также в сокращенном, 8+3, виде. Например, каталог `LinuxRulez` будет сохранен как `LINUXR~1`. Для работы с длинными именами файлов мы должны использовать новые системные вызовы, которые доступны только под управлением Windows. Их описание вы сможете найти в документе `Ralf Brown Interrupt List` (<http://www.ctyme.com/rbrown.htm>).

Все функции для работы с длинными именами файлов доступны через функцию с номером 0x70. Через регистр `AL` передается номер оригинальной функции (работающей с короткими именами), то есть те из рассмотренных функций, которые сами принимают аргумент через регистр `AL`, не поддерживаются. Например, вместо функции открытия файла `0x3D` при работе с длинными именами придется использовать функцию `0x6C`: в регистр `AX` нужно поместить значение `0x706C`, а остальные аргументы передавать как обычно.

10.7. Работа с каталогами

Операции над каталогами в языке ассемблера ничем не отличаются от таких же операций в языках высокого уровня. Привычные функции MKDIR, RMDIR, CHDIR и т.п. реализованы как набор системных вызовов, которые мы рассмотрим в этом параграфе.

Создание и удаление каталога (MKDIR, RMDIR)

Для создания каталога используется функция **0x39**, принимающая указатель на имя нового каталога. Удалить каталог можно с помощью функции **0x3A**. Эта функция удаляет только пустой каталог, то есть предварительно нужно удалить из него все файлы и подкаталоги.

Ввод:	Вывод:
Создание нового каталога AH = 0x39 Удаление каталога AH = 0x3A DS:DX = указатель на имя каталога	Успех: CF = 0 Ошибка: CF = 1 AX = код ошибки

Смена текущего каталога (CHDIR)

Очень полезной является функция **0x3B** — функция смены текущего каталога (CHDIR). Именно от текущего каталога все функции DOS, работающие с файлами, отсчитывают относительное имя файла. То есть если имя файла не указано полностью, то DOS будет искать файл в текущем каталоге.

Ввод:	Вывод:
Смена текущего каталога AH = 0x3B DS:DX = указатель на имя каталога	Успех: CF = 0 Ошибка: CF = 1 AX = код ошибки

Получение текущего каталога (GETCWD)

Иногда нам нужна обратная операция — мы хотим знать, какой каталог является текущим. Функция **0x47** возвращает имя текущего каталога, но без буквы диска и начального слэша.

Ввод:	Вывод:
Получение текущего каталога AH = 0x47 DL = на каком диске: DL = 0x00 текущий диск DL = 0x01 диск A: ... DS:SI = указатель на буфер, в котором будет сохранено название текущего каталога	Успех: CF = 0 Ошибка: CF = 1 AX = код ошибки

Если нам нужно узнать имя текущего диска, можно использовать функцию 0x19:

Ввод:	Вывод:
Получить имя диска AH = 0x19	AL = номер текущего диска: AL = 0x00 диск A: AL = 0x01 диск B: ...

В листинге 10.8. приведена программа pwd, выводящая на экран текущий каталог, включая букву диска

Листинг 10.8. Программа, выводящая на экран текущий каталог, включая букву диска

```
SECTION .text
org 0x100
mov ah,0x19          ;функция DOS для получения текущего диска
int 0x21             ;вызываем DOS
add byte [buffer],al ;добавляем номер к букве 'A',
                    ;которую заранее поместили в первый
                    ;байт буфера
xor dl,dl            ;будем искать текущий каталог текущем
                    ;диске
mov ah,0x47          ;функция DOS для получения текущего
                    ;каталога
mov si,buffer+3      ;первые три байта буфера отведены под "C:\"
int 0x21             ;вызываем DOS
mov ah,0x40          ;функция DOS для вывода на экран
mov bx,0x0001        ;это дескриптор стандартного вывода
mov cx,BUFF_LEN      ;сколько символов нужно вывести
mov dx,buffer
int 0x21             ;вызываем DOS
mov ax,0x4c00        ;функция DOS для завершения программы
int 0x21             ;вызываем DOS
SECTION .data
buffer db "A:\\"
times 64 db " "      ;оставшуюся часть буфера заполняем
                    ;пробелами
db 0x0D,0x0A         ;и завершаем строку
BUFF_LEN equ $-buffer ;оператор $ обозначает текущий адрес,
                    ;то есть конец буфера. Отнимая от него
                    ;начальный адрес, получаем длину буфера
```

Для полноты описания давайте рассмотрим обратную функцию — установку текущего диска 0x0E:

Ввод:	Вывод:
Установка текущего диска AH = 0x0E DL = 0x00 диск A: DL = 0x01 диск B: ...	AL = количество имеющихся дисков

10.8. Управление памятью

Каждая программа, работающая под управлением DOS, получает в свое распоряжение всю оставшуюся свободную память. Можно вернуть временно не нужную память обратно в распоряжение DOS, а потом при необходимости затребовать ее обратно. Чтобы «отдать» блок памяти операционной системе, нужно объявить его свободным при помощи системного вызова 0x4A.

Изменение размера блока памяти

Для изменения размера блока памяти используется функция 0x4A. Можно запросить как увеличение, так и уменьшение блока. Если запрос не может быть выполнен, то флаг переноса будет содержать 1, а в регистре BX будет максимальный доступный размер блока памяти.

Ввод:	Вывод:
Изменение размера блока памяти AH = 0x4A BX = новый размер в «параграфах» ES = сегмент блока, размер которого подлежит изменению	Успех: CF = 0 Ошибка CF = 1 AX = код ошибки BX = максимально доступный размер блока памяти

Новый размер блока указывается в «параграфах» — единицах величиной в 16 байтов.

COM-программы обычно «заказывают» блок памяти размером в 64 Кб (то есть 0x1000 параграфов). Выделить 0x1000 параграфов можно с помощью следующего кода:

```

mov ah, 0x4A      ; функция DOS для изменения размера памяти
mov bx, 0x1000    ; к-во параграфов; ES уже установлен
int 0x21          ; вызов DOS
jc error

```

Запросить еще несколько блоков памяти можно с помощью команды выделения памяти 0x48, рассмотренной ниже.

Выделение памяти

С помощью функции 0x48 мы можем попросить операционную систему выделить новый блок памяти в любое время. Если такой блок доступен, он будет распределен. Адрес его сегмента будет возвращен в регистре AX.

Ввод:	Вывод:
Распределение блока памяти AH = 0x48 BX = размер в «параграфах»	Успех: CF = 0 AX = адрес сегмента выделенного блока Ошибка: CF = 1 AX = код ошибки BX = максимально доступный размер блока памяти (в параграфах)

Посмотрим, как выделить блок размером 64 Кб, используя эту функцию. Предположим, что ненужную память мы уже вернули операционной системе, вызвав функцию 0x4A.

```
mov ah,0x48      ;функция DOS для распределения памяти
mov bx,0x1000    ;нам нужно 64 KB
int 0x21         ;вызываем DOS
jc error         ;если CF = 1, то запрошенный блок нам не дали,
                ;в BX — максимальное число доступных
                ;параграфов
push es          ;сохраняем оригинальный ES в стеке
mov es,ax        ;загружаем в ES адрес нового сегмента
```

Освобождение памяти

Выделенная (распределенная) память должна быть возвращена обратно операционной системе. Сделать это можно с помощью функции 0x49:

Ввод:	Вывод:
Освобождение блока памяти AH = 0x49 ES = адрес сегмента освобождаемого блока	Успех: CF = 0 Ошибка CF = 1 AX = код ошибки

10.9. Аргументы командной строки

Для получения доступа к аргументам командной строки в языке C используются переменные argc (количество аргументов) и argv (массив самих аргументов). Операционная система DOS передает все аргументы одной строкой, записывая эту строку по адресу 0x81 от начала сегмента программы. Длина этой строки с учетом завершающего символа 0x0D сохраняется по смеще-

нию 0x80 (один байт). Нам придется извлекать отдельные аргументы из этой строки вручную.

Следующая программа выводит переданные ей параметры на экран и завершает работу. Ничего нового в ней нет: мы просто выводим строку, записанную по адресу 0x81 (листинг 10.9).

Листинг 10.9. Программа, выводящая переданные ей параметры на экран

```
SECTION .text                ;начало кода
%define STDOUT 1            ;стандартный вывод
org 0x100                   ;COM-файл
mov ah,0x40                 ;функция DOS для записи в файл
mov bx,STDOUT
mov dx,0x81                 ;начало строки
xor cx,cx                   ;сбрасываем длину строки
mov cl,[0x80]               ;загружаем в CL длину строки
mov di,cx                   ;копируем в DI
add di,dx                   ;добавляем длину и получаем конец
                           ;строки [0x0D]
inc di                      ;устанавливаем DI после последнего
                           ;символа строки
mov byte [di],0xA           ;дописываем 0x0A, чтобы получить
                           ;полноценный EOL
inc cx                      ;и увеличиваем длину строки на 1
int 0x21                   ;выводим строку на экран
mov ax,0x4c00               ;завершаем работу
int 0x21                   ;вызываем DOS
```

10.10. Коды ошибок

Определить, что произошла ошибка во время системного вызова, можно по флагу переноса: если он установлен ($CF=1$), то вызов завершился ошибочно. В регистре AX будет код ошибки. Подробную информацию об ошибке можно получить с помощью системного вызова $AH=0x59$. Также описание ошибок можно найти в Ralf Brown Interrupt List.

В таблице 10.3 перечислены только коды самых распространенных ошибок.

DOS-коды самых распространенных ошибок

Таблица 10.3

Код	Описание ошибки
0x0002	Файл не найден
0x0003	Путь не найден
0x0005	Отказано в доступе

Код	Описание ошибки
0x0008	Недостаточно памяти
0x0009	Неверный адрес блока памяти
0x000f	Диск указан неправильно
0x0010	Невозможно удалить текущий каталог
0x0011	Некорректный носитель

10.11. Отладка

10.11.1. Что такое отладка программы и зачем она нужна

При разработке программы на языке ассемблера очень важным является этап отладки программы. Отладка позволяет найти и проанализировать не опечатки, а логические ошибки в вашей программе: именно из-за них программа работает не так, как нужно. Для отладки ваших программ вам понадобится специальная программа — отладчик (debugger).

По-английски ошибка в программе называется «bug» (жук) в честь жука, который, согласно легенде, поселился в одном из первых компьютеров и нарушил его работу. Отсюда и название класса программ для «поиска и удаления жуков» — debugger. Эти программы позволяют шаг за шагом следить за исполнением программы, наблюдать за состоянием регистров и памяти, изменять на ходу их значения и т.п.

В операционной системе DOS (и Windows) есть свой собственный отладчик — программа debug.exe. А программу Turbo Debugger, входящую в состав любой среды разработки от Borland, можно порекомендовать для опытных пользователей. Также можно использовать очень хороший отладчик IDA (Interactive Disassembler) от DataRescue.

После запуска отладчика нам нужно загрузить отлаживаемую программу в память (используя команды вроде open, load). Если программа требует параметров командной строки, не забудьте их указать: любой отладчик позволяет сделать это. Есть два вида отладчиков: графические (более дружелюбные, но требующие больших ресурсов) и текстовые (ориентированные на опытных пользователей). Графические отладчики сразу открывают несколько окон, отображающих состояние регистров и памяти, а также окно с исходным текстом (деассемблированным машинным кодом) программы. Для отображения этой информации в текстовых отладчиках имеются команды (отдельная команда для вывода кода, отдельная — для отображения регистров и отдельная — для дампа памяти).

Наиболее важная функция отладчика — пошаговое выполнение программы (команда `step`). За один шаг выполняется одна команда программы. После выполнения команды выводятся новые значения регистров процессора, а также команда, которая будет выполнена следующей.

Абсолютно все отладчики имеют функцию `step-over`, позволяющую «проскакивать» подпрограмму, то есть все команды подпрограммы будут выполнены за один шаг. Эту функцию можно использовать, если вы уверены в правильности действий подпрограммы.

Для продолжения выполнения всей программы без дальнейшей отладки имеются команды `go` или `continue`.

Любой отладчик позволяет устанавливать точки прерывания (контрольные точки или `breakpoints`). Программа будет выполнена до этой точки, после чего при выполнении заданного условия управление будет возвращено отладчику и вы сможете просмотреть содержание регистров и памяти. Условием может быть выполнение команды, хранящейся по указанному адресу, изменение значения определенной ячейки памяти и т.п.

При отладке вместо символических имен идентификаторов (меток, переменных) будут показаны их адреса. Если вас такое положение вещей не устраивает (конечно, удобнее, когда вместо адреса отображается имя переменной, например, `result`), вы должны перекомпилировать программу, «упаковав» в нее символьные имена. Размер программы существенно увеличится, но после отладки вы сможете заново перекомпилировать программу, исключив символы.

В этой книге мы будем использовать свободно распространяемый отладчик **grdb**, который вы сможете найти в Интернете.

10.11.2. Отладчик **grdb.exe**

Методика использования

Отладчик **grdb** (Get Real Debugger, то есть «настоящий отладчик») написан на языке ассемблера и распространяется бесплатно. Скачать его можно по адресу: <http://www.members.tripod.com/~ladsoft/grdb.htm>. Интерфейс программы — классический, то есть текстовый. Для запуска программы введите команду **grdb**.

```
C:\>grdb
GRDB version 3.6 Copyright (c) LADsoft 1997-2002
History enabled
eax:00000000 ebx:00000000 ecx:00000000 edx:00000000 esi:00000000
edi:00000000 ebp:00000000 esp:0000FFEE eip:00000100 eflags:00000202
NV UP EI PL NZ NA PO NC
```

```
ds: 10FB es:10FB fs:10FB gs:10FB ss:10FB cs:10FB
10FB:0100 74 1E      jz      0120
->
```

Появление на экране приглашения (->) означает, что отладчик готов к выполнению наших команд. Загрузим в отладчик нашу программу `pwd.com` (она выводит текущий каталог на стандартный вывод). Для этого введем команду `l` `pwd.com` (`l` — это `load`, то есть загрузить). После имени исполняемого файла можно указать через пробел аргументы программы.

```
->l pwd.com
Size: 00000069
->
```

Теперь начнем пошаговое выполнение программы. Для выполнения одной команды программы служит команда отладчика `t`. Введите `t` и нажмите «Enter»:

```
->t
eax:00001900 ebx:00000000 ecx:00000069 edx:00000000 esi:00000000
edi:00000000 ebp:00000000 esp:0000FFEE eip:00000102 eflags:00000202
NV UP EI PL NZ NA PO NC
ds: 10FB es:10FB fs:10FB gs:10FB ss:10FB cs:10FB
10FB:0102 CD 21      int     21
->
```

Команда `t` выполнила первую команду нашей программы, и в регистре `АН` появилось значение `0x19`. Следующая команда — `INT 21`, то есть вызов операционной системы. Слева от нее вы видите ее адрес (пара регистров `CS:IP`) и машинный код. После нажатия `t` и клавиши «Enter» будет выполнен вызов `DOS`, и в `AL` окажется номер текущего диска.

```
->t
eax:00001902 ebx:00000000 ecx:00000069 edx:00000000 esi:00000000
edi:00000000 ebp:00000000 esp:0000FFEE eip:00000104 eflags:00000202
NV UP EI PL NZ NA PO NC
ds: 10FB es:10FB fs:10FB gs:10FB ss:10FB cs:10FB
10FB:0104 00 06 24 01 add     [0124],al      ds:[0124]=41
->
```

Теперь `AL` содержит значение `02`, означающее диск `C:`.

Следующая инструкция — это `ADD [buffer],al`, добавляющая содержимое `AL` к символу 'A'. Как видите, вместо символьного имени `buffer` указан адрес этой переменной. Мы можем этим воспользоваться: чтобы увидеть значение переменной `buffer`, нужно сделать дамп памяти по адресу (смещению) `0124`. Для этого введем команду `d 124`. Для команды `d` самый распространенный аргумент — это начальный адрес памяти, содержимое которой нужно вывести, хотя можно указывать и имя регистра.

```

->d 124
10FB:0120          -41 3A 5C 20-20 20 20 20-20 20 20 20 A:\
10FB:0130 20 20 20 20-20 20 20 20-20 20 20 20-20 20 20 20
10FB:0140 20 20 20 20-20 20 20 20-20 20 20 20-20 20 20 20
10FB:0150 20 20 20 20-20 20 20 20-20 20 20 20-20 20 20 20
10FB:0160 20 20 20 20-20 20 20 0D-0A C7 06 8C-CD 01 00 EB
.....
10FB:0170 60 26 8A 47-01 32 E4 40-D1 E0 03 D8-26 8A 0F 32 '&.G.2.@....&..2
10FB:0180 ED 0B C9 74-0F 43 53 26-8B 1F E8 C6-00 5B 73 41 ...t.CS&.....[sA
10FB:0190 43 43 E2 F2-2E C7 06 8C-CD 03 00 EB-34 26 8A 47 CC.....4&.G
10FB:01A0 01 32 E4 40-D1 E0 03 D8-26 8A 07 32-E4 D1 E0 40 .2.@....&..2...@
->

```

После выполнения ADD введите еще одну команду **t**. Строка A:\ будет заменена строкой C:\ (наш текущий диск). Проверьте это с помощью команды **d**:

```

->d 124
10FB:0120          -43 3A 5C 20-20 20 20 20-20 20 20 20 C:\

```

Остальное содержимое памяти не изменилось, поэтому часть вывода я отрезал.

Для выполнения оставшейся части программы вместо команды **t** введите команду **g** (go), которая возвращает управление отладчику только после завершения программы.

Команда **a** позволяет модифицировать команды и данные программы. Ее параметры — это адрес (смещение) для сохранения новой инструкции. Начинаящим рекомендуется не использовать эту команду, а исправлять ошибки в исходном коде, после чего перекомпилировать свою программу.

Если вам нужно выполнить подпрограмму за один шаг (функция step over), используйте команду **p**.

Просмотреть код программы можно с помощью команды **u**, которой в качестве параметра нужно передать стартовый адрес:

```

->u 100
10FB:0100 B4 19      mov     ah,0019
10FB:0102 CD 21      int     21
10FB:0104 00 06 24 01 add     [0124],al
10FB:0108 30 D2      xor     dl,dl
10FB:010A B4 47      mov     ah,0047
10FB:010C BE 27 01   mov     si,0127
10FB:010F CD 21      int     21
10FB:0111 B4 40      mov     ah,0040
10FB:0113 BB 01 00   mov     bx,0001
10FB:0116 B9 45 00   mov     cx,0045
10FB:0119 BA 24 01   mov     dx,0124

```

```
10FB:011C CD 21      int      21
10FB:011E B8 00 4C    mov     ax, 4C00
->
```

Завершить работу отладчика можно командой **q**.

Основные команды отладчика grdb

Основные команды отладчика **grdb** приведены в табл.10.4.

Основные команды отладчика **grdb** Таблица 10.4

Команда	Назначение
l <filename> [p]	Загружает программу <filename> с аргументами (необязательными) [p]
t	Выполняет одну команду, возвращая управление отладчику после ее выполнения
p	Подобна предыдущей команде, но проходит подпрограмму за один шаг
u	Преобразует машинный код команд обратно в символический код ассемблера. Обычно используется для просмотра листинга программы
g	Продолжает выполнение программы, передавая управление отладчику после завершения программы
a <addr>	Записывает новую команду по адресу <addr>
?	Выводит список всех команд и краткое их описание
b <num>,addr	Устанавливает точку прерывания номер num по адресу addr. Номер точки должен быть в диапазоне от 0 до F, то есть всего можно установить 16 различных точек прерывания
b	Выводит список всех установленных точек прерывания
q	Выход

Пример разработки и отладки программы

Напишем небольшую программу, читающую два или более десятичных числа из командной строки и выводящую на экран их сумму.

Для преобразования строк в числа и наоборот будем использовать разработанную нами подпрограмму из главы 7. Как выводить строку на экран, мы тоже знаем, поэтому в данной главе сосредоточим наше внимание именно на отладке нашей программы.

Давайте вспомним, как нужно вызывать нашу подпрограмму **ASCIIToNum**:

```
; _____
; ASCIIToNum
; _____
; esi = указатель на конвертируемую строку, заканчивающуюся символом 0x0
; ecx = основание системы счисления
; Результат:
; eax = число
```

Поскольку мы работаем в операционной системе DOS, мы будем использовать только 16-битную адресацию. Указатель (смещение) будет передан в SI, а не в ESI.

Наша программа должна уметь работать с аргументами командной строки. Все аргументы передаются операционной системой программе как одна строка, внутри которой аргументы разделены пробелами. Для извлечения отдельных аргументов из этой строки мы напишем несколько подпрограмм. Первая из них — SkipSpace, пропускающая любое количество пробелов в начале строки. Подпрограмме нужно передать указатель на строку, а на выходе мы получим адрес первого символа строки, отличного от пробела.

```

;-----
; SkipSpace
;-----
; si = указатель на строку
;
; Возвращает:
; si = первый символ, отличный от пробела
SkipSpace:
.again:
lodsb                ;загружаем в AL байт по адресу DS:SI,
                    ;увеличиваем SI
cmp al,' '           ;сравниваем символ с пробелом
jz again             ;если равно, продолжаем
ret

```

Итак, мы удалили из начала строки аргументов все пробелы, первый пробел в строке будет после первого аргумента. Напомним, что строка аргументов начинается с адреса 0x81, а заканчивается символом 0xD. Теперь преобразуем первый аргумент в число. У нас есть подпрограмма, которая умеет считывать число из строки, завершенной нулевым байтом. Чтобы воспользоваться ею, нужно поместить нулевой байт на место символа пробела после первого аргумента. Напишем еще одну подпрограмму, которая пропускает в строке аргументов последовательность подряд идущих цифр и возвращает указатель на позицию в строке после числа.

```

;-----
; SkipNum
; Ввод: SI = указатель на строку
; Вывод: DI = указатель на позицию в строке после числа
;-----
SkipNum:
mov di,si            ;DI используется как указатель
.find_end_of_number:
inc di               ;DI=DI+1
cmp [di],'0'         ;сравнить символ с ASCII-кодом
                    ;цифры ноль

```

```

jb .end_found          ;если меньше, конец числа найден
cmp [di], '9'          ;сравнить символ с ASCII-кодом
                        ;цифры 9
ja .end_found          ;если больше, конец числа найден
jmp .find_end_of_number ;переходим к следующему символу
.end_found:
ret

```

А вот тело основной программы:

```

mov si, 0x81           ;загружаем в SI указатель на начало
                        ;строки аргументов
call SkipNum           ;переходим в конец первого числа
mov [di], 0            ;пишем нулевой байт
mov ecx, 10            ;читаем число в десятичной записи
call ASCIIToNum        ;в EAX будет результат, SI остается
                        ;неизменным
mov edx, eax           ;сохраняем число в EDX
mov si, di             ;загружаем в SI указатель на позицию
                        ;в строке сразу после первого числа

inc si
call SkipSpace         ;пропускаем любые пробелы
call SkipNum           ;и пропускаем число
mov [di], 0            ;записываем нулевой символ после
                        ;второго аргумента
call ASCIIToNum        ;в EAX — результат, SI остается
                        ;неизменным
add eax, edx           ;добавляем предыдущее число,
                        ;сохраненное в EDX

```

Первым делом мы пропускаем число и записываем нулевой байт после него. Подпрограмма преобразования строки в число будет «думать», что строка закончилась. После этого мы преобразуем фрагмент строки параметров от ее начала до первого нулевого байта в число. Регистр DI теперь указывает на нулевой байт, значит, нужно увеличить указатель на единицу. Наша подпрограмма преобразования строки в число ожидает указателя в регистре SI, и мы копируем его из DI.

Теперь наша строка будет начинаться со второго аргумента, перед обработкой которого мы снова пропускаем все пробелы в начале строки.

После преобразования второго числа мы складываем его с первым. После этого в EAX будет сумма двух чисел, которую нужно преобразовать в строку и вывести на экран. Для преобразования числа в строку будем использовать подпрограмму NumToASCII:

```

;-----
; NumToASCII
;-----
;
; eax = 32-битное число
; ebx = основание системы счисления
; edi = указатель на буфер для сохранения строки
; Результат:
; заполненный буфер
;-----
mov ebx,10                ;вносим в EBX основание системы
                           ;счисления
mov di,buffer              ;в DI — адрес буфера
call NumToASCII            ;перед вызовом подпрограммы в EAX
                           ;находится вычисленная сумма

```

Теперь выведем полученную строку на экран, используя функцию записи в файл стандартного вывода. Но мы не знаем длину строки, которая получилась после преобразования. Пока сделаем так: зарезервированное для строки место заполним пробелами и выведем строку вместе с лишними пробелами. Потом мы можем модифицировать программу, удалив из строки лишние пробелы.

```

mov ah,0x40                ;функция DOS для записи в файл
mov dx,di                  ;эта функция ожидает указателя на
                           ;строку в DS:DX
mov cx,25                  ;количество выводимых символов
int 0x21
mov ax, 0x4c00              ;завершаем программу
int 0x21

```

В секции данных заполним наш буфер 25 пробелами:

```
buffer times 25 db ' '
```

Теперь нам осталось написать только секцию кода для нашей программы.

```

SECTION .text
org 0x100
mov si, 0x81                ;загружаем в SI указатель на начало
                           ;строки аргументов
call SkipNum                ;переходим в конец первого числа
mov [di],0                  ;пишем нулевой байт
mov ecx,10                  ;читаем число в десятичной записи
call ASCIIIToNum             ;в EAX будет результат, SI остается
                           ;неизменным
mov edx,ecx                  ;сохраняем число в EDX
mov si,di                   ;загружаем в SI указатель на позицию
                           ;в строке сразу после первого числа
inc si
call SkipSpace               ;пропускаем любые пробелы
call SkipNum                 ;и пропускаем число

```



```

mov [di],0          ;записываем нулевой символ после
                    ;второго аргумента
call ASCIIToNum      ;в EAX – результат, SI остается
                    ;неизменным
add eax,edx          ;добавляем предыдущее число,
                    ;сохраненное в EDX
mov ebx,10           ;основание системы счисления для
                    ;преобразования числа
mov di,buffer        ;загружаем в DI адрес буфера выводимой
                    ;строки
call NumToASCII      ;преобразуем EAX в строку
mov ah,0x40          ;функция ДОС для записи в файл
mov dx,di            ;указатель на строку ожидается в DX,
                    ;у нас он в DI
mov cx,25            ;количество выводимых символов
int 0x21             ;вызываем DOS
mov ax, 0x4c00       ;завершаем программу
int 0x21

;-----
; SkipSpace – пропуск пробелов
;-----
; si = указатель на строку
;
; Возвращает:
; si = первый символ, отличный от пробела
SkipSpace:
.again:
lodsb               ;загружаем в AL байт по адресу DS:SI,
                    ;увеличиваем SI
cmp al,' '          ;сравниваем символ с пробелом
jz again            ;если равно, продолжаем
ret

;-----
; SkipNum – пропуск числа
; Ввод: SI = указатель на строку
; Вывод: DI = указатель на позицию в строке после числа
;-----
SkipNum:
mov di,si           ;DI используется как указатель
.find_end_of_number:
inc di              ;DI=DI+1
cmp [di],'0'        ;сравнить символ с ASCII-кодом цифры нуля
jb .end_found       ;если меньше, конец числа найден
cmp [di],'9'        ;сравнить символ с ASCII-кодом цифры 9
ja .end_found       ;если больше, конец числа найден
jmp .find_end_of_number ;переходим к следующему символу
.end_found:
ret

```

```
; ***** Далее следуют все необходимые подпрограммы *****
SECTION .data
buffer times 25 db ' ' ;буфер для сохранения результата
```

Теперь сохраним нашу программу и попытаемся ее откомпилировать. В результате компиляции увидим это:

```
nasm -f bin -o test.com test.asm
test.asm:6: error: operation size not specified
test.asm:13: error: symbol 'SkipSpace' undefined
test.asm:15: error: operation size not specified
test.asm:41: error: symbol 'again' undefined
test.asm:53: error: operation size not specified
test.asm:55: error: operation size not specified
test.asm:141: error: phase error detected at end of assembly.
```

Как видите, в нашей программе много ошибок. Давайте их исправлять. Первая ошибка — в строке 6:

```
mov [di],0 ;пишем нулевой байт
```

Эта команда требует указания размера операнда, выполним ее требование:

```
mov byte [di],0 ;пишем нулевой байт
```

Следующая наша ошибка заключается в том, что мы неправильно написали имя подпрограммы. Ассемблер чувствителен к регистру символов, поэтому SkipSpace — это не SkipSpace. Исправим это.

После этого исправим ошибку в строке 13 — это та же самая ошибка, что и в строке 6. В строке 41 мы забыли указать точку перед 'again':

```
jz .again
```

Команды в строках 53 и 55 тоже требуют указания размера операнда. Просто добавьте ключевое слово 'byte' между командой **cmp** и квадратной скобкой.

Исправив все ошибки, откомпилируем программу снова. Она выполняется, но ничего не выводит на экран. Проверим код для вывода строки на экран:

```
mov ah,0x40 ;функция DOS для записи в файл
mov dx,di ;указатель ожидается в DX, у нас он в DI
mov cx,25 ;количество печатаемых символов
int 0x21 ;вызываем DOS
```

Все ясно, мы ведь забыли поместить дескриптор потока стандартного вывода 0x01 в BX — в этом регистре все еще содержится значение 10. Исправим эту ошибку и снова откомпилируем программу. Теперь можно запустить программу с параметрами 45 и 50. Мы получим следующее значение:

```
C:\test 45 50
945
```

Мы должны получить 95, но никак не 945. Что произошло? Ответить на этот вопрос нам поможет отладчик. Запустим **grdb**:

```
C:>grdb
GRDB version 3.6 Copyright (c) LADsoft 1997-2002
->
```

Введем команду 'l test.com 45 50' для загрузки программы с параметрами:

```
->l test.com 45 50
Size: 000000E1
->
```

Предположим, что наши подпрограммы работают правильно, поэтому будем использовать команду **p** для прохождения подпрограмм за один шаг. Посмотрим, что возвращает подпрограмма **ASCIIToNum** в регистр **EAX**. Первый вызов **ASCIIToNum** следует сразу после инструкции **MOV ecx,10**. Рассмотрим весь сеанс отладки до первого вызова **ASCIIToNum**.

```
->l test.com 45 50
Size: 000000E1
->p
eax:00000000 ebx:00000000 ecx:000000E1 edx:00000000 esi:00000081
edi:00000000 ebp:00000000 esp:0000FFEE eip:00000103 eflags:00000202
NV UP EI PL NZ NA PO NC
ds: 10FB es:10FB fs:10FB gs:10FB ss:10FB cs:10FB
10FB:0103 E8 44 00 call 014A
->p
eax:00000000 ebx:00000000 ecx:000000E1 edx:00000000 esi:00000081
edi:00000084 ebp:00000000 esp:0000FFEE eip:00000106 eflags:00000287
NV UP EI MI NZ NA PE CY
ds: 10FB es:10FB fs:10FB gs:10FB ss:10FB cs:10FB
10FB:0106 C6 05 00 mov byte [di],0000 ds:[0084]=20
->p
eax:00000000 ebx:00000000 ecx:000000E1 edx:00000000 esi:00000081
edi:00000084 ebp:00000000 esp:0000FFEE eip:00000109 eflags:00000287
NV UP EI MI NZ NA PE CY
ds: 10FB es:10FB fs:10FB gs:10FB ss:10FB cs:10FB
10FB:0109 66 B9 0A 00 00 00 mov ecx,0000000A
->p
eax:00000000 ebx:00000000 ecx:0000000A edx:00000000 esi:00000081
edi:00000084 ebp:00000000 esp:0000FFEE eip:0000010F eflags:00000287
NV UP EI MI NZ NA PE CY
ds: 10FB es:10FB fs:10FB gs:10FB ss:10FB cs:10FB
10FB:010F E8 6D 00 call 017F
->p
eax:000003B1 ebx:00000000 ecx:0000000A edx:00000000 esi:00000081
edi:00000084 ebp:00000000 esp:0000FFEE eip:00000112 eflags:00000297
```

```

NV UP EI MI NZ AC PE CY
ds: 10FB es:10FB fs:10FB gs:10FB ss:10FB cs:10FB
10FB:0112 66 89 C2      mov     edx, eax
->

```

После первого преобразования получим результат 0x3B1, который никак не соответствует десятичному 45. 0x3B1 — это 945. Почему мы получили именно этот результат? Если мы предположили, что подпрограмма корректна, то тогда нужно проверить параметры, которые мы передали подпрограмме. Посмотрим, что находится в памяти по адресу, который содержится в SI:

```

->d si
10FB:0080      20 34 35-00 35 30 0D-01 01 01 01-01 01 01 01 45.50.....
10FB:0090 01 01 01 01-01 01 01 01-01 01 01 01-01 01 01 01 .....
10FB:00A0 01 01 01 01-01 01 01 01-01 01 01 01-01 01 01 01 .....
10FB:00B0 01 01 01 01-01 01 01 01-01 01 01 01-01 01 01 01 .....
10FB:00C0 01 01 01 01-01 01 01 01-01 01 01 01-01 01 01 01 .....
10FB:00D0 01 01 01 01-01 01 01 01-01 01 01 01-01 01 01 01 .....
10FB:00E0 01 01 01 01-01 01 01 01-01 01 01 01-01 01 01 01 .....
10FB:00F0 01 01 01 01-01 01 01 01-01 01 01 01-01 01 01 01 .....
10FB:0100 BE 81 00 E8-44 00 C6 05-00 66 B9 0A-00 00 00 E8 ....D....f.....
->

```

Наша функция преобразования не проверяет входящие символы, поэтому у нас и вышла ошибочка. Обратите внимание: перед первым символом у нас образовался пробел (код 20). Лечится это очень просто: перед вызовом подпрограммы преобразования нужно вызвать SkipSpace, чтобы удалить все символы в начале строки. Отредактируем нашу программу и снова откомпилируем ее. Запускаем с теми же самими параметрами:

```

C:\test3 45 50
5

```

Опять мы получили не то, что нужно. Снова запускаем отладчик и переходим к первому вызову ASCIIToNum:

```

->p
eax:00000034 ebx:00000000 ecx:0000000A edx:00000000 esi:00000083
edi:00000084 ebp:00000000 esp:0000FFEE eip:00000112 eflags:00000287
NV UP EI MI NZ NA PE CY
ds: 10FB es:10FB fs:10FB gs:10FB ss:10FB cs:10FB
10FB:0112 E8 6D 00      call    0182
->d si
10FB:0080      35-00 35 30 0D-01 01 01 01-01 01 01 01 5.50.....
10FB:0090 01 01 01 01-01 01 01 01-01 01 01 01-01 01 01 01 .....
10FB:00A0 01 01 01 01-01 01 01 01-01 01 01 01-01 01 01 01 .....
10FB:00B0 01 01 01 01-01 01 01 01-01 01 01 01-01 01 01 01 .....
10FB:00C0 01 01 01 01-01 01 01 01-01 01 01 01-01 01 01 01 .....
10FB:00D0 01 01 01 01-01 01 01 01-01 01 01 01-01 01 01 01 .....

```

```

10FB:00E0 01 01 01 01-01 01 01 01-01 01 01 01-01 01 01 01 .....
10FB:00F0 01 01 01 01-01 01 01 01-01 01 01 01-01 01 01 01 .....
10FB:0100 BE 81 00 E8-41 00 E8 44-00 C6 05 00-66 B9 0A 00 ....A..D....f...
->

```

Получается, что виновата подпрограмма SkipSpace, которая пропускает один лишний символ, поэтому в качестве аргументов мы получаем 5 и 0 — отсюда и результат $5 + 0 = 5$. Команда LODSB увеличивает указатель SI в любом случае, даже тогда, когда встреченный символ — не пробел. Значит, перед выходом из подпрограммы этот указатель нужно уменьшить и проблема исчезнет. Перепишем подпрограмму так:

```

SkipSpace:
.again:
lodsbyte             ;загружаем в AL байт по адресу DS:SI,
                    ;увеличиваем SI
cmp al, ' '          ;сравниваем символ с пробелом
jz again             ;если равно, продолжаем
dec si
ret

```

Все, «баг» исправлен, сохраним полученную программу, откомпилируем и запустим ее:

```

C:\test4 45 50
95

```

Полученное значение правильно. Поздравляю! Процесс разработки программы завершен!

10.12. Резидентные программы

Обычно после завершения очередной программы DOS освобождает память, которую эта программа занимала, и загружает на ее место новую. Но некоторые программы, завершившись, продолжают оставаться в памяти. Такие программы называются резидентными. В большинстве случаев резидентные программы используются для обработки прерываний процессора. Типичный пример резидентной программы — драйвер мыши, который опрашивает последовательный порт компьютера, читая коды, передаваемые мышью. Этот драйвер использует программное прерывание 0x33, через которое он передает свои события (перемещение мышки, нажатие той или иной клавиши мышки) прикладным программам.

На резидентные программы накладывается ряд ограничений. Так, им нельзя самим вызывать программные прерывания. Дело в том, что DOS — принципиально однозадачная система, поэтому функции прерываний DOS не обладают свойством реентерабельности (повторной входимости). Если программа, об-

рабатывающая системный вызов DOS, сама выполнит системный вызов, то ОС рухнет. Существуют, правда, способы обойти это ограничение, но они слишком сложны для того, чтобы рассматривать их в этой книге.

Напишем небольшую резидентную программу, которая активизируется при нажатии клавиши Scroll Lock. Программа изменяет цвет всех символов, отображенных на экране, на красный.

До сих пор мы старались избегать прямого взаимодействия с периферийными устройствами и использовали функции операционной системы. В случае с резидентной программой использовать DOS мы больше не можем, поэтому ничего другого нам не остается — мы должны взаимодействовать с устройствами напрямую.

Примите пока как данность, что из порта 0x60 можно получить так называемый скан-код нажатой клавиши, то есть не ASCII-код посылаемого ею символа, а просто число, указывающее ее положение на клавиатуре. Скан-код клавиши Scroll Lock равен 0x46.

Каждое нажатие клавиши генерирует аппаратное прерывание IRQ1, которое принимает скан-код от клавиатуры, преобразовывает его в ASCII-код и ставит в очередь непрочитанных клавиш. Эта очередь доступна операционной системе и прикладным программам через вызовы BIOS.

Чтобы активизировать программу, мы должны перехватить прерывание IRQ 1 (int 0x9), то есть сделать нашу программу его обработчиком. Затем мы будем в цикле читать скан-код из порта 0x60. Если нажатая клавиша — не Scroll Lock, то мы вернем ее скан-код первоначальному обработчику прерывания. В противном случае мы выполним намеченное действие (сменим цвет символов) и вернем управление первоначальному обработчику прерывания. Выход из прерывания осуществляется по команде `iret`.

Во время обработки прерывания мы должны сохранить контекст выполняемой программы, то есть сохранить все регистры «внутри» нашей подпрограммы. Если этого не сделать, последствия предугадать будет сложно, но в большинстве случаев крах выполняемой программы будет обеспечен.

Первым делом напишем подпрограмму `color`, изменяющую цвет всех символов на красный. Организация видеопамати была рассмотрена (хоть и очень кратко) в начале этой главы. Символы, выведенные на экран, хранятся начиная с адреса сегмента 0xB800 в формате символ:цвет. Наша задача — модифицировать значения байтов, находящихся по нечетным адресам: 0xB800:0x0001, 0xB800:0x0003, 0xB800:0x0005 и т.д. — это и есть цвет символов. Не вдаваясь в подробности, скажем, что красному цвету соответствует значение 0x04.

Подпрограмма заполняет 80x25 байтов памяти, что соответствует стандартному разрешению текстового режима.

```

color:
push ax                ;сохраняем в стеке значения
                        ;регистров, которые
push cx                ;будем использовать
push si
push es
xor si,si              ;сбрасываем SI
mov ax, 0xB800          ;загружаем адрес сегмента в AX
mov es,ax              ;и в сегментный регистр
mov cx,80*25           ;количество повторений
.repeat:
inc si                 ;увеличиваем SI на 1
mov byte [es:si],0x04  ;записываем цвет 0x04 — красный
inc si                 ;увеличиваем на 1
dec cx                 ;уменьшаем CX на 1
jnz .repeat            ;переходим к .repeat, пока CX > 0
pop es                 ;восстанавливаем все регистры
pop si
pop cx
pop ax
ret

```

Мы можем протестировать нашу подпрограмму. Допишем к ней заголовок секции кода и завершим обычную, не резидентную, программу системным вызовом 0x4C:

```

SECTION .text
call color
mov ax, 0x4c00
int 0x21
color:
...

```

Теперь напишем обработчик прерывания IRQ 1:

```

new_handler:
push ax                ;сохраняем значение AX
in al, 0x60            ;читаем скан-код клавиши
cmp al, 0x46           ;сравниваем с 0x46 (Scroll-Lock)
jnz pass_on            ;если нет, переходим к pass_on
call color              ;вызываем подпрограмму
pass_on:
pop ax                 ;восстанавливаем значение AX
jmp far [cs:old_vector] ;передаем управление
                        ;первоначальному обработчику

```

Переменная `old_vector` должна содержать адрес первоначального обработчика прерывания (старое значение вектора прерывания вектора). В качестве

сегментного регистра мы используем CS, потому что другие сегментные регистры в ходе обработки прерывания могут получить произвольные значения. Сохраним значение старого вектора прерывания в переменную `old_vector`, а указатель на наш обработчик поместим вместо него в таблицу прерываний. Назовем эту процедуру `setup`.

```

setup:
cli                ;отключаем прерывания
xor ax,ax          ;сбрасываем AX
mov es,ax          ;таблица прерываний находится
                  ;в сегменте 0
mov ax,new_handler ;с метки new_handler начинается
                  ;новый обработчик прерывания
xchg ax,[es:0x9*4] ;вычисляем адрес старого вектора
                  ;и меняем местами со значением AX.
                  ;Теперь в таблице смещение нового
                  ;обработчика, в AX – старого
mov [ds:old_vector],ax ;сохраняем старый адрес в
                  ;переменной old_vector
mov ax,cs          ;адрес текущего сегмента – CS
xchg ax,[es:0x9*4+2] ;пишем в таблицу сегмент нового
                  ;обработчика,
                  ;в AX загружаем сегмент старого
mov [ds:old_vector+2],ax ;сохраняем сегмент на 2 байта
                  ;дальше old_vector
sti                ;включаем прерывания
ret                ;выходим из подпрограммы

```

Теперь мы попросим операционную систему не разрушать программу после ее завершения, а хранить ее в памяти. Для этого используется системный вызов `0x31`.

Ввод:	Вывод:
Сохраняем программу резидентной AH = 0x31 AL = выходной код DX = число параграфов памяти, необходимых для хранения резидентной программы	Ничего

Код всей нашей резидентной программы `resident.asm` приведен в листинге 10.10.

Листинг 10.10. Пример резидентной программы

```

SECTION .text
org 0x100
jmp initialize

```



```

new_handler:
push ax                ;сохраняем значение AX
in al, 0x60            ;читаем скан-код клавиши
cmp al, 0x46           ;сравниваем с 0x46 (Scroll-Lock)
jnz pass_on           ;если нет, переходим к pass_on
call color            ;вызываем подпрограмму
pass_on:
pop ax                ;восстанавливаем значение AX
jmp far [cs:old_vector] ;передаем управление
                        ;первоначальному обработчику

color:
push ax               ;сохраняем в стеке значения
                    ;регистров, которые
push cx               ;будем использовать
push si
push es
xor si,si             ;сбрасываем SI
mov ax, 0xB800        ;загружаем адрес сегмента в AX
mov es,ax             ;и в сегментный регистр
mov cx,80*25          ;количество повторений
.repeat:
inc si                ;увеличиваем SI на 1
mov byte [es:si],0x4  ;записываем цвет 0x04 – красный
inc si                ;увеличиваем на 1
dec cx                ;уменьшаем CX на 1
jnz .repeat           ;переходим к .repeat, пока CX > 0
pop es                ;восстанавливаем все регистры
pop si
pop cx
pop ax
ret
old_vector dd 0
initialize:
call setup            ;вызываем регистрацию своего
                    ;обработчика
mov ax,0x3100         ;функция DOS: делаем программу
                    ;резидентной
mov dx,initialize     ;вычисляем число параграфов:
                    ;в памяти нужно разместить
                    ;весь код вплоть до метки initialize
shr dx,4              ;делим на 16
inc dx                ;добавляем 1
int 0x21              ;завершаем программу и остаемся
                    ;резидентом

setup:
cli                  ;отключаем прерывания

```

```

xor ax,ax                ;сбрасываем AX
mov es,ax                ;таблица прерываний находится в
                        ;сегменте 0
mov ax,new_handler       ;с метки new_handler начинается
                        ;новый обработчик прерывания
xchg ax,[es:0x9*4]       ;вычисляем адрес старого вектора
                        ;и меняем
                        ;местами со значением AX.
                        ;Теперь в таблице
                        ;смещение нового обработчика,
                        ;в AX — старого
mov [ds:old_vector],ax   ;сохраняем старый адрес
                        ;в переменной old_vector
mov ax,cs                ;адрес текущего сегмента — CS
xchg ax,[es:0x9*4+2]     ;пишем в таблицу сегмент нового
                        ;обработчика,
                        ;в AX загружаем сегмент старого
mov [ds:old_vector+2],ax ;сохраняем сегмент на 2 байта
                        ;дальше old_vector
sti                      ;включаем прерывания
ret                      ;выходим из подпрограммы

```

Теперь откомпилируем программу:

```
nasm -f bin -o resident.com resident.asm.
```

После ее запуска мы ничего не увидим, но после нажатия клавиши Scroll Lock весь текст на экране «покраснеет». Избавиться от этого резидента (выгрузить программу из памяти) можно только перезагрузкой компьютера или закрытием окна эмуляции DOS, если программа запущена из-под Windows.

10.13. Свободные источники информации

Дополнительную информацию можно найти на сайтах:

- www.ctyme.com/rbrown.htm — HTML-версия списка прерываний Ральфа Брауна (Ralf Brown's Interrupt List);
- http://programmistu.narod.ru/asm/lib_1/index.htm — «Ассемблер и программирование для IBM PC» Питер Абель.

Глава 11 Программирование в Windows

- «Родные» Windows-приложения
- Программная совместимость
- Запуск DOS-приложений
од Windows
- Свободные источники информации

Ассемблер на примерах.
Базовый курс

11.1. Введение

Когда-то Microsoft Windows была всего лишь графической оболочкой для операционной системы DOS. Но со временем она доросла до самостоятельной полнофункциональной операционной системы, которая использует защищенный режим процессора. В отличие от UNIX-подобных операционных систем (Linux, BSD и др.), в Windows функции графического интерфейса пользователя (GUI) встроены непосредственно в ядро операционной системы.

11.2. «Родные» Windows-приложения

«Родные» Windows-приложения взаимодействуют с ядром операционной системы посредством так называемых API-вызовов. Через API (Application Programming Interface) операционная система предоставляет все услуги, в том числе управление графическим интерфейсом пользователя.

Графические элементы GUI имеют объектную структуру, и функции API часто требуют аргументов в виде довольно сложных структур со множеством параметров. Поэтому исходный текст простейшей ассемблерной программы, управляющей окном и парой кнопок, займет несколько страниц.

В этой главе мы напишем простенькую программу, которая отображает окошко со знакомым текстом «Hello, World!» и кнопкой ОК. После нажатия ОК окно будет закрыто.

11.2.1. Системные вызовы API

В операционной системе DOS мы вызывали ядро с помощью прерывания 0x21. В Windows вместо этого нам нужно использовать одну из функций API. Функции API находятся в различных динамических библиотеках (DLL). Мы должны знать не только имя функции и ее параметры, но также и имя библиотеки, которая содержит эту функцию: user32.dll, kernel32.dll и т.д. Описание API можно найти, например, в справочной системе Borland Delphi (файл win32.hlp). Если у вас нет Delphi, вы можете скачать только файл win32.zip (это архив, содержащий файл win32.hlp):

<ftp://ftp.borland.com/pub/delphi/techpubs/delphi2/win32.zip>

11.2.2. Программа «Hello, World!» с кнопкой под Windows

Наша программа должна отобразить диалоговое окно и завершить работу. Для отображения диалогового окна используется API-функция `MessageBoxA`, а для завершения программы можно использовать функцию `ExitProcess`.

В документации по Windows API `MessageBoxA` написано:

```
int MessageBox(  
    HWND hWnd,           // дескриптор окна владельца  
    LPCTSTR lpText,       // адрес текста окна сообщения  
    LPCTSTR lpCaption,    // адрес текста заголовка окна сообщения  
    UINT uType            // стиль окна  
);
```

Первый аргумент — это дескриптор окна владельца, то есть родительского окна. Поскольку у нас нет никакого родительского окна, мы укажем значение 0. Второй аргумент — это указатель на текст, который должен быть отображен в диалоговом окне. Обычно это строка, заканчивающаяся нулевым байтом. Третий аргумент аналогичен второму, только он определяет не текст окна, а текст заголовка. Последний аргумент — это тип (стиль) диалогового окна, мы будем использовать символическую константу `MB_OK`.

Второй вызов API — это `ExitProcess`, ему нужно передать всего один аргумент (как в DOS), который определяет код завершения программы.

Чтобы упростить разработку Windows-программы на языке ассемблера, подключим файл `win32.inc`, который определяет все типы аргументов API-функций (например, `HWND` и `LPCTSTR` соответствуют простому типу `dword`) и значения констант. Подключим этот файл:

```
%include «win32n.inc»;
```

Мы будем использовать API-функции, которые находятся в динамических библиотеках, поэтому нам нужно воспользоваться директивами `EXTERN` и `IMPORT`:

```
EXTERN MessageBoxA      ;MessageBoxA определен вне программы  
IMPORT MessageBoxA user32.dll ;а именно — в user32.dll  
EXTERN ExitProcess      ;ExitProcess определен вне программы  
IMPORT ExitProcess kernel32.dll ;а именно — в kernel32.dll
```

Точно так же, как в DOS, нужно создать две секции: кода и данных.

```
SECTION CODE USE32 CLASS=CODE      ;наш код  
SECTION DATA USE32 CLASS=DATA     ;наши статические данные
```

Осталось понять, как можно вызвать функции API. Подробнее мы обсудим это в главе 13, посвященной компоновке программ, написанных частью на ассемблере, а частью на языках высокого уровня, а сейчас рассмотрим только правила передачи аргументов функциям API.

Соглашение о передаче аргументов называется **STDCALL**. Аргументы передаются через стек в порядке справа налево (так же, как в языке C), а очистка стека входит в обязанности вызванной функции (как в Паскале).

Мы помещаем аргументы в стек по команде **PUSH**, а затем вызываем нужную функцию по команде **CALL**. Не нужно беспокоиться об очистке стека. Код нашей программы приведен в листинге 11.1.

Листинг 11.1. Программа «Hello, World!» с кнопкой под Windows

```
%include <win32n.inc>      ;подключаем заголовочный файл
EXTERN MessageBoxA         ;MessageBoxA определен вне программы
IMPORT MessageBoxA user32.dll ;а именно – в user32.dll
EXTERN ExitProcess         ;ExitProcess определен вне программы
IMPORT ExitProcess kernel32.dll ;а именно – в kernel32.dll
SECTION CODE USE32 CLASS=CODE ;начало кода
..start:                   ;метка для компоновщика,
                           ;указывающая точку входа
push UINT MB_OK            ;помещаем в стек последний аргумент.
                           ;тип окна: с единственной кнопкой ОК
push LPCTSTR title         ;теперь помещаем в стек адрес
                           ;нуль-завершенной строки заголовка
push LPCTSTR banner        ;адрес нуль-завершенной строки,
                           ;которая будет отображена в окне
push HWND NULL             ;указатель на родительское окно –
                           ;нулевой: нет такого окна
call [MessageBoxA]         ;вызываем функцию API. Она выведет
                           ;окно сообщения и вернет управление
                           ;после нажатия ОК
push UINT NULL             ;аргумент ExitProcess – код возврата
call [ExitProcess]         ;завершаем процесс
SECTION DATA USE32 CLASS=DATA
banner db 'Hello world!',0xD,0xA,0 ;строка сообщения
                                       ;с символом EOL
title db 'Hello',0           ;строка заголовка
```

Для того, чтобы откомпилировать эту программу, нам нужна версия **NASM** для Windows, которую можно скачать по адресу: <http://nasm.sourceforge.net>. **NASM** создаст объектный файл, который нужно будет скомпоновать в исполняемый файл. Для компоновки мы используем свободно распространяемый компоновщик **alink**, который можно скачать по адресу: <http://alink.sourceforge.net>.

Назовем наш файл **msgbox.asm**. Теперь запустим **nasmw** с параметром **-fobj**:

```
C:\WIN32>NASMW -fobj msgbox.asm
```

В результате получим файл `msgbox.obj`, который нужно передать компоновщику **alink**:

```
C:\WIN32>ALINK -oPE msgbox
```

Параметры `-o` определяет тип исполняемого файла. Родным для Windows является тип PE. После компоновки появится файл `msgbox.exe`, который можно будет запустить.

11.3. Программная совместимость

Для процессора, работающего в защищенном режиме, доступен другой специальный режим — VM86, обеспечивающий эмуляцию реального режима. Все привилегированные команды (то есть `cli`, `porf` и др.), а также все обращения к периферийным устройствам (команды `in` и `out`) перехватываются и эмулируются ядром операционной системы так, что прикладной программе «кажется», что она действительно управляет компьютером. На самом же деле все вызовы функций DOS и BIOS обрабатываются ядром операционной системы.

11.4. Запуск DOS-приложений под Windows

Для запуска DOS-приложения в среде Windows мы используем так называемый «Сеанс MS DOS». Для запуска DOS-режима выполните команду `cmd` (Пуск → Выполнить → `cmd`). Открывшееся окно работает в режиме VM86 и полностью эмулирует функции DOS. Если файлы компилятора NASM находятся в каталоге `C:\NASM`, мы можем использовать команду DOS, чтобы перейти в этот каталог:

```
cd C:\NASM
```

Помните, что в DOS существует ограничение на длину файлов — все имена файлов должны быть в формате 8+3 (8 символов — имя, 3 — расширение). Для редактирования исходных файлов используйте редактор, который показывает номера строк — так будет проще найти ошибку. Избегайте также «слишком дружественных» приложений, которые добавляют расширение `«.txt»` после расширения `«.asm»`.

11.5. Свободные источники информации

Мы рекомендуем следующие источники, посвященные программированию на языке ассемблера в Windows:

<http://win32asm.cjb.net>

<http://rs1.szif.hu/~tomcat/win32>

<http://asm.shadrinsk.net/toolbar.html>

Глава 12 Программирование в Linux

- Структура памяти процесса
- Передача параметров командной строки и переменных окружения
- Вызов операционной системы
- Коды ошибок
- Облегчим себе работу: утилиты Asmutils. Макросы Asmutils
- Отладка. Отладчик ALD
- Ассемблер GAS
- Ключи командной строки компилятора

Ассемблер на примерах.
Базовый курс

12.1. Введение

Linux — современная многозадачная операционная система. Большая часть ядра Linux написана на C, но небольшая его часть (аппаратно-зависимая) написана на языке ассемблера. Благодаря портируемости языка C Linux быстро распространилась за пределы x86-процессоров. Ведь для переноса ядра на другую аппаратную платформу разработчикам пришлось переписать только ту самую маленькую часть, которая написана на ассемблере.

Как любая другая современная многозадачная система, Linux строго разделяет индивидуальные процессы. Это означает, что ни один процесс не может изменить ни другой процесс, ни тем более ядро, вследствие чего сбой одного приложения не отразится ни на других приложениях, ни на операционной системе.

В x86-совместимых компьютерах процессы в памяти защищены так называемым защищенным режимом процессора. Этот режим позволяет контролировать действия программы: доступ программы к памяти и периферийным устройствам ограничен правами доступа. Механизмы защиты разделены между ядром операционной системы (которому разрешается делать абсолютно все) и процессами (им можно выполнять только непривилегированные команды и записывать данные только в свою область памяти).

Защищенный режим также поддерживает виртуальную память. Ядро операционной системы предоставляет все операции для работы с виртуальной памятью. Кроме, конечно, трансляции логических адресов в физические — эта функция выполняется «железом» (см. главу 8).

Благодаря виртуальной памяти (и, конечно же, 32-битным регистрам), любой процесс в Linux может адресовать 4 Гб адресного пространства. Именно 4 Гб и выделены каждому процессу. Что имеется в виду? Если сделать дамп памяти от 0 до самого конца (4 Гб), вы не обнаружите ни кода другого процесса, ни данных другого процесса, ни кода ядра — 4 Гб полностью предоставлены в ваше распоряжение. Ясно, что реальный объем виртуальной памяти будет зависеть от физической памяти и от объема раздела подкачки, но суть в том, что процессы скрыты друг от друга и друг другу не мешают.

12.2. Структура памяти процесса

Мы уже знаем, что нам доступны все 4 Гб и ни один процесс не может вторгнуться в наше адресное пространство. Как же распределена память нашего процесса? Как было сказано в предыдущих главах, программа состоит из четырех секций: секция кода, секция статических данных, секция динамических данных (куча) и стек. Порядок, в котором загружаются эти секции в память, определяется форматом исполняемого файла. Linux поддерживает несколько форматов, но самым популярным является формат ELF (Executable and Linkable Format). Рассмотрим структуру адресного пространства ELF-файла. Предположим для простоты, что программа не использует динамических библиотек.

Адрес:

0x08048000

.text	Исполняемый код
.data	Статические данные (известны во время компиляции)
.bss	Динамические данные (так называемая куча)
...	Свободная память
.stack	Стек

0xBFFFFFFF (3 Гб)

Обычно программа загружается с адреса 0x08048000 (примерно 128 Мб). При загрузке программы загружается только одна ее страница. Остальные страницы загружаются по мере необходимости (неиспользуемые части программы никогда физически не хранятся в памяти).

На диске программа хранится без секций .bss и .stack — эти секции появляются только тогда, когда программа загружается в память.

Если программа подключает какие-нибудь динамические библиотеки, их модули загружаются в ее адресное пространство, но начинаются с другого адреса (обычно с 1 Гб и выше). Секции этих модулей аналогичны секциям обычной программы (то есть .text, .data, .bss).

А что хранится в трехгигабайтном «зазоре» между .bss и .stack, то есть в свободной памяти? Эта память принадлежит процессу, но она не распределена по страницам. Запись в эту область вызовет сбой страницы (page fault) — после этого ядро уничтожит вашу программу.

Для создания динамических переменных вам нужно попросить ядро распределить соответствующую память, но об это мы поговорим чуть позже.

12.3. Передача параметров командной строки и переменных окружения

Если процессы полностью изолированы друг от друга, то как нам получить параметры командной строки и переменные окружения? Ведь они — собственность другого процесса — оболочки, запустившей нашу программу. Решили эту проблему очень просто: при запуске программы параметры командной строки и переменные окружения помещаются в стек программы. А поскольку стек нашей программы — это наша собственность, мы можем без проблем получить к нему доступ. Рассмотрим структуру стека:

ESP после запуска программы	Свободная память
argc	Количество параметров (dword)
argv[0]	Указатель на имя программы
argv[1] ... argv[argc-1]	Указатели на аргументы программы
NULL	Конец аргументов командной строки
env[0] env[1] ... env[n]	Указатели на переменные окружения
NULL	Конец переменных окружения

Нужный аргумент легко вытолкнуть из стека по команде POP, а потом записать в какую-нибудь переменную. Первое значение, которое мы получим из стека — это количество аргументов командной строки (argc), второй — указатель на имя нашей программы.

Если argc > 1, значит, программа была запущена с аргументами и дальше в стеке находятся они.

Примеры обработки аргументов командной строки будут рассмотрены после того, как вы научитесь выводить данные на экран.

12.4. Вызов операционной системы

Вызвать функции операционной системы DOS можно было с помощью прерывания 0x21. В Linux используется похожий метод: прерывание с номером 0x80. Но как мы можем с помощью прерывания добраться до ядра, если оно находится за пределами нашего адресного пространства? Благодаря защищенному режиму, при вызове прерывания 0x80 изменяется контекст программы (значение сегментного регистра) и процессор начинает выполнять код ядра. После завершения обработки прерывания будет продолжено выполнение нашей программы.

Подобно DOS, аргументы отдельных системных вызовов (syscalls) передаются через регистры процессора, что обеспечивает небольшой выигрыш в скорости. Номер системного вызова помещается в регистр EAX. Если системный вызов принимает аргументы, то он ожидает их в регистрах EBX, ECX и т.д. По соглашению для передачи аргументов служит набор регистров в фиксированном порядке: EBX, ECX, EDX, ESI и EDI. Ядро версии 2.4.x и выше допускает также передачу аргумента через регистр EBP.

12.5. Коды ошибок

Возвращаемое системным вызовом значение заносится в EAX. Это код ошибки, определяющий состояние завершения системного вызова, то есть уточняющий причину ошибки. Код ошибки всегда отрицательный. В программе на языке ассемблера, как правило, достаточно отличать успешное завершение от ошибочного, но в ходе отладки значение кода ошибки может помочь выяснить ее причину.

Справочная система Linux содержит man-страницы, описывающие каждый системный вызов вместе с кодами, значениями и причинами всех ошибок, которые он может вернуть.

12.6. Man-страницы

В отличие от DOS и Windows ОС Linux полностью документирована. В ее справочной системе (которая называется Manual Pages — Страницы Руководства) вы найдете исчерпывающие сведения не только обо всех установленных программах, но и обо всех системных вызовах Linux. Конечно, для эффективного использования man-страниц (как и остальной Linux-документации) вам придется выучить английский, поскольку далеко не все man-страницы переведены на русский язык. Достаточно уметь читать руководства хотя бы со словарем.

Предположим, что мы хотим написать простейшую программку, которая завершает свою работу сразу после запуска. В DOS нам нужно было использовать системный вызов AH=0x4C. Помните? Теперь давайте напишем подобную программку под Linux. Для этого найдите файл unistd.h, который обычно находится в каталоге /usr/src/linux/include/asm:

```
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_
/*
 * This file contains the system call numbers.
 */
#define __NR_exit          1
#define __NR_fork          2
```

```
#define __NR_read          3
#define __NR_write        4
#define __NR_open         5
#define __NR_close        6
....
#define __syscall1(type,name,type1,arg1) \
type name(type1 arg1) \
{ \
long __res; \
__asm__ volatile (<int $0x80> \
: <=<a> (__res) \
: <0> (__NR_##name), <b> ((long)(arg1))); \
__syscall__return(type,__res); \
}
...
```

В этом файле вы найдете номера системных вызовов Linux. Нас интересует вызов `__NR_exit`:

```
#define __NR_exit          1
```

Это значит, что функция ядра, завершающая процесс, имеет номер 1. Описания функций ядра (системных вызовов) собраны в секции 2 справочного руководства. Посмотрим, что там сказано о функции `exit()`. Для этого введите команду:

```
man 2 exit
```

Вы увидите содержимое ман-страницы:

```
_EXIT(2)          Linux Programmer's Manual          _EXIT(2)
NAME
_exit, _Exit - terminate the current process
SYNOPSIS
#include <unistd.h>
void _exit(int status);
#include <stdlib.h>
void _Exit(int status);
DESCRIPTION
The function _exit terminates the calling process <immedi-
ately>. Any open file descriptors belonging to the process
are closed; any children of the process are inherited by
process 1, init, and the process's parent is sent a SIGCHLD
signal. The value status is returned to the
parent process as the process's exit status, and can be col-
lected using one of the wait family of calls. The function
_Exit is equivalent to _exit.
RETURN VALUE
These functions do not return.
....
```

Системному вызову 'exit' нужно передать всего один параметр (как в DOS) — код возврата (ошибки) программы. Код 0 соответствует нормальному завершению программы.

Код нашего «терминатора» будет выглядеть так:

```
mov eax,1    ;номер системного вызова — exit
mov ebx,0    ;код возврата 0
int 0x80     ;вызов ядра и завершение текущей программы
```

12.7. Программа «Hello, World!» под Linux

Давайте немного усложним нашу программу — пусть она выведет заветную фразу и завершит работу. Вспомните, что в главе 8 говорилось о файловых дескрипторах стандартных потоков — ввода (STDIN, обычно клавиатура), вывода (STDOUT, обычно экран) и ошибок (STDERR, обычно экран). Наша программа «Hello, World!» должна вывести текст на устройство стандартного вывода STDOUT и завершиться. С точки зрения операционной системы STDOUT — это файл, следовательно, мы должны записать нашу строку в файл.

Снова заглянув в `unistd.h`, находим системный вызов `write()`. Смотрим его описание:

```
man 2 write
WRITE(2)      Linux Programmer's Manual      WRITE(2)
NAME
write — write to a file descriptor
SYNOPSIS
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
...
```

Если вы не знаете C, то данная man-страница, скорее всего, для вас будет непонятна. Как передать аргументы системному вызову? В языке ассемблера вы можете загрузить в регистр либо непосредственное значение, либо значение какого-то адреса памяти. Для простоты будем считать, что для аргументов, не отмеченных звездочкой, нужно указывать непосредственные значения, а аргументы со звездочкой требуют указателя.

Функции **write** нужно передать три аргумента: дескриптор файла, указатель на строку, данные из которой будут записаны в файл (это **buf**), и количество байтов этой строки, которые нужно записать. Функция возвратит количество успешно записанных байтов или отрицательное число, которое будет кодом ошибки.

Для компилирования нашей программы в объектный файл будем использовать **nasm**, а для компоновки — компоновщик **ld**, который есть в любой версии

Linux. Для получения объектного файла в формате ELF нужно использовать опцию компилятора `-f elf`.

Компоновщик `ld` обычно вызывается с ключами, перечисленными в табл. 12.1.

Наиболее часто используемые ключи компоновщика `ld`

Таблица 12.1

Ключ	Назначение
<code>-o <name></code>	Установить имя выходного (исполняемого) файла <code><name></code>
<code>-s</code>	Удалить символьную информацию из файла

Чтобы компоновщик `ld` определил точку входа программы, мы должны использовать глобальную метку `_start`.

Код нашей программы «Hello, World!» приведен в листинге 12.1.

Листинг 12.1. Программа «Hello, World!» под Linux

```
SECTION .text
global _start           ;указываем компоновщику точку входа.
                        ;это сделать необходимо

_start:
mov eax,4               ;первый аргумент – номер системного
                        ;вызова – write
mov ebx,1               ;константа STDOUT определена как 1
mov ecx,hello           ;адрес выводимой строки
mov edx,len             ;длина «Hello, World!» вместе с символом
                        ;конца строки

int 0x80               ;вызываем ядро для вывода строки
mov eax,1               ;системный вызов номер 1 – exit
mov ebx,0               ;код завершения программы 0
int 0x80               ;вызываем ядро для завершения программы

SECTION .data
hello db «Hello, world!»,0xa ;наша строка плюс символ
                        ;конца строки

len equ $ - hello       ;вычисляем длину строки
```

Теперь откомпилируем программу:

```
nasm -f elf hello.asm
```

А теперь запустим компоновщик:

```
ld -s -o hello hello.o
```

Ключ `-o` определяет имя результирующего исполняемого файла. Ключ `-s` удаляет символьную информацию из исполняемого файла, чтобы уменьшить его размер.

Теперь осталось запустить программу:

```
./hello
Hello, World!
```

12.8. Облегчим себе работу: утилиты Asmutils

Asmutils — это коллекция системных программ, написанных на языке ассемблера. Эти программы работают непосредственно с ядром операционной системы и не требуют библиотеки LIBC.

Asmutils — идеальное решение для небольших встроенных систем или для «спасательного» диска.

Программы написаны на ассемблере NASM и совместимы с x86-процессорами. Благодаря набору макросов, представляющих системные вызовы, Asmutils легко портируется на другие операционные системы (нужно модифицировать только конфигурационный файл). Поддерживаются следующие операционные системы: BSD (FreeBSD, OpenBSD, NetBSD), Unixware, Solaris и AtheOS.

Для каждой операционной системы Asmutils определяет свой набор символических констант и макросов, побочным эффектом которых оказывается улучшение читабельности ассемблерных программ.

В листинге 12.2 можно увидеть, как будет выглядеть наша программа hello с использованием Asmutils.

Листинг 12.2. Программа «Hello World!» под Linux с использованием Asmutils

```
%include «system.inc»
CODESEG                ;начало секции кода
START:                 ;начало программы для
                        ;компоновщика
sys_write STDOUT,hello,len
                        ;этот макрос записывает в регистры
                        ;аргументы для системного вызова write
                        ;и вызывает write
sys_exit 0              ;то же самое для системного вызова exit
DASEG                  ;начало секции данных
hello db «Hello, World!»,0xa
len equ $-hello
END
```


Теперь код нашей программы выглядит намного проще. После подстановки макросов этот код превратится в текст из предыдущего примера и работать будет точно так же. Если вы хотите запустить эту же программу на FreeBSD, модифицируйте только файл MCONFIG.

Домашняя страница Asmutils — <http://asm.sourceforge.net/asmutils.html>, там же вы сможете скачать последнюю версию Asmutils. Пакет Asmutils распространяется по Стандартной Общественной лицензии GNU, то есть бесплатно.

Рекомендуем вам скачать Asmutils прямо сейчас, поскольку в последующих программах мы будем использовать именно этот пакет. Теперь давайте откомпилируем нашу программу.

Убедитесь, что у вас установлен NASM (он входит в состав разных дистрибутивов — как RedHat-совместимых, так и Debian-совместимых). Распакуйте файл asmutils-0.17.tar.gz. После распаковки получим обычную структуру каталогов — /doc, /src и /inc. Целевая платформа задается в файле MCONFIG. Он самодокументирован, так что если вам придется его немного подправить, то вы легко разберетесь, как именно.

В каталоге /src есть файл Makefile. В первых строках этого файла указываются имена программ, подлежащих сборке (компиляции и компоновке). Пропишите в нем имя выходного файла — hello (без расширения .asm). После этого выполните команду make. В результате получите готовый исполняемый файл (компоновщик запускать уже не нужно).

12.9. Макросы Asmutils

Аргументы системного вызова передаются как часть макроса. В зависимости от операционной системы генерируются соответствующие команды ассемблера, а аргументы заносятся в соответствующие регистры.

Макросы для системных вызовов начинаются с префикса sys_, после этого следует имя системного вызова так, как оно пишется в man-странице. В Linux параметры передаются в том же порядке, в котором они описаны в man-страницах. Системный вызов возвращает значение в регистре EAX.

Макрос **sys_exit 0** после подстановки превратится в следующие команды:

```
mov eax,1          ;номер системного вызова 1 — exit
mov ebx,0          ;код возврата 0
int 0x80           ;вызываем ядро и завершаем программу
```

Аргумент макроса не обязательно должен быть непосредственным значением. Если вы храните код возврата в переменной rtn, то можете написать макрос **sys_exit [rtn]**, и после подстановки он превратится в:

```
mov eax,1          ;номер системного вызова 1 — exit
mov ebx,[rtn]      ;код возврата — содержится в переменной rtn
int 0x80           ;вызываем ядро и завершаем программу
```

Если регистр уже содержит требуемое значение, вы можете пропустить установку аргумента, используя ключевое слово `EMPTY` — тогда значение данного регистра останется как есть. Пример использования «аргумента» `EMPTY` будет приведен в программе для копирования файлов.

12.10. Операции файлового ввода/вывода (I/O)

При работе с файлами в Linux помните, что Linux — это UNIX-подобная операционная система, и просто так доступ к файлам вы не получите: у каждого файла есть права доступа, а у вашего процесса должны быть соответствующие полномочия.

Открытие файла

С помощью системного вызова `open()` мы можем не только открыть, но и создать файл, если это необходимо. Во второй секции справочной системы Linux вы найдете полное описание этого вызова (man 2 open). Мы же рассмотрим его в общих чертах:

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Первый аргумент системного вызова содержит указатель на имя файла (нуль-завершенную строку), второй аргумент — это массив битов, определяющий режим открытия файла (чтение/запись и др.). Третий аргумент необязательный — он определяет права доступа для вновь создаваемого файла.

Вызов возвращает дескриптор файла или отрицательное значение в случае ошибки.

Символические названия битовых флагов представлены в таблице 12.2 (информация из man-страницы).

Символические названия битовых флагов

Таблица 12.2

Флаг	Режим открытия
<code>O_RDONLY</code>	Открывает файл только для чтения
<code>O_WRONLY</code>	Открывает файл только для записи
<code>O_RDWR</code>	Открывает файл для чтения и для записи
<code>O_CREAT</code>	Создает файл, если его не существует
<code>O_TRUNC</code>	Обрезает файл до нулевой длины
<code>O_APPEND</code>	Открывает файл для дозаписи, то есть новые данные будут записаны в конец файла (опасно на NFS)
<code>O_LARGEFILE</code>	Используется для обработки файлов размером более 4 Гб

Если второй аргумент содержит флаг `O_CREAT`, то третий аргумент обязателен. Он указывает, какие права доступа будут присвоены новому файлу. Для указания прав доступа можно комбинировать символические имена, из которых чаще всего используются следующие:

Флаг	Описание
<code>S_IRWXU</code>	Владелец может читать, записывать и запускать файл
<code>S_IRGRP</code>	Члены группы владельца могут читать файл
<code>S_IROTH</code>	Все остальные могут читать файл

Откроем файл, имя которого хранится в переменной `name`, для чтения и записи:

```
sys_open name, O_RDWR, EMPTY
test eax,eax      ;проверяем EAX на отрицательное значение
js .error_open    ;более подробно этот пример описывается
                  ;в главе 7
```

Имя файла можно определить в секции данных с помощью псевдокоманды `DB`:

```
name DB <my_file_which_has_a_very_long_name.txt>,0
```

Отдельные флаги могут комбинироваться с помощью оператора `|` (побитовое `OR`). Откроем файл для чтения и записи, а если файла не существует, то создадим его, присвоив ему права доступа `700` (чтение/запись/выполнение для владельца, ничего для остальных):

```
sys_open name, O_RDWR | O_CREAT, S_IRWXU
test eax,eax
js .error_open  ;переходим к error_open, если произошла ошибка
...            ;в случае успеха в EAX будет дескриптор файла
```

Заккрытие файла

Как и в DOS, файл нужно закрыть с помощью системного вызова. Этот вызов называется `close()`, а соответствующий макрос — `sys_close`. Этот макрос требует одного аргумента — дескриптора закрываемого файла. Если дескриптор файла находится в `EAX`, то для закрытия файла воспользуемся макросом:

```
sys_close eax
```

Чтение из файла

Данные из файла читаются блоками различной длины. Если явно не указано иное, то чтение данных начинается с той позиции, на которой закончилась предыдущая операция чтения или записи, то есть последовательно. Для чтения данных из файла используется системный вызов `read`:

```
ssize_t read(int fd, void *buf, size_t count);
```

Первый аргумент — дескриптор файла, второй содержит указатель на область памяти, в которую будут записаны прочитанные данные, а третий — максимальное количество байтов, которые нужно прочитать. Вызов возвращает либо количество прочитанных байтов, либо отрицательное число — код ошибки.

Макрос **sys_read** можно также использовать для чтения данных с клавиатуры, если в качестве файлового дескриптора передать **STDIN** — дескриптор потока стандартного ввода.

Запись в файл

Функция записи требует таких же аргументов, как и функция чтения: дескриптора файла, буфера с данными, которые нужно записать в файл, и количество байтов, которые нужно записать. Вызов возвращает либо количество записанных байтов, либо отрицательное число — код ошибки:

```
ssize_t write(int fd, const void *buf, size_t count);
```

Давайте напишем простую программу, читающую последовательность символов с клавиатуры до нажатия клавиши «Enter» и преобразующую их в верхний регистр. Мы ограничимся только латинским алфавитом, то есть первой половиной таблицы ASCII. Работать программа будет в бесконечном цикле — прочитали, преобразовали, вывели, опять прочитали — а прервать ее работу можно будет нажатием комбинации Ctrl + C.

По нажатии клавиши «Enter» системный вызов **read** завершит работу и вернет управление нашей программе, заполнив буфер прочитанными символами.

Каждый символ из диапазона ‘a’ — ‘z’ будет преобразован в ‘A’ — ‘Z’. После преобразования мы выведем получившийся результат на **STDOUT**.

При использовании **Asmutils** секция кода должна начинаться идентификатором **CODESEG**, секция статических данных — **DATASEG**, а секция динамических данных — **UDATASEG**. Мы обязательно должны подключить заголовочный файл **system.inc** (листинг 12.3).

Листинг 12.3. Программа, читающая последовательность символов с клавиатуры до нажатия клавиши «Enter» и преобразующая их в верхний регистр

```
%include «system.inc»
#define MAX_DATA 10
CODESEG
START:
again:      ;читаем следующую строку
sys_read STDIN,read_data,MAX_DATA
test eax,eax      ;ошибка? (отрицательное значение EAX)
```

```

js endprog          ;да? выходим
                    ;нет? EAX содержит количество
                    ;прочитанных символов
add ecx, eax         ;в ECX был указатель на первый символ
                    ;в буфере,
                    ;теперь это позиция последнего
                    ;элемента + 1

compare_next:
dec ecx              ;уменьшаем указатель строки
cmp byte [ecx], 'a'  ;если символ < 'a', то он вне нашего
                    ;диапазона,
jb no_conversion     ;поэтому его преобразовывать не нужно
cmp byte [ecx], 'z'  ;если > 'z'
ja no_conversion     ;снова не преобразуем
sub byte [ecx], 0x20 ;преобразуем в верхний регистр
                    ;вычитанием 0x20

no_conversion:
cmp ecx, read_data   ;указатель указывает на начало буфера?
jz printit           ;да? Завершаем преобразование,
                    ;выводим строку
jmp short compare_next ;иначе переходим к следующему символу
printit:              ;количество прочитанных байтов сейчас в
                    ;EAX.
                    ;Сохранять это значение в промежуточном
                    ;регистре
                    ;не нужно, потому что макросы
                    ;подставляются справа
                    ;налево, и значение EAX будет сначала
                    ;занесено в EDI, а потом заменено на
                    ;код вызова write

sys_write STDOUT, read_data, ecx
jmp short again       ;после вывода читаем след. строку
endprog:
sys_exit 255          ;выходим с кодом 255
UDATASEG              ;секция неинициализированных переменных
read_data resb MAX_DATA
END                   ;конец программы

```

Программа ведет себя правильно, даже если ввести строку длиннее, чем предусматривает константа `MAX_DATA`. Ядро операционной системы сохранит лишние символы и передаст их программе при следующем вызове `sys_read`.

Рассмотрим еще один пример. На этот раз мы будем копировать файл А в файл В. Имена файлов мы будем передавать программе как аргументы командной строки.

Сначала нашей программе нужно прочитать число аргументов из стека: если оно меньше 3, программа должна завершить работу с ошибкой (первый аргу-

мент — это имя нашей программы, остальные два — имена файлов). Затем по команде `POR` программа прочитает имена файлов. Файл `A` будет открыт на чтение, а файл `B` будет создан или «обрезан» (перезаписан). В случае ошибки, программа завершит работу.

Если ошибки нет, программа будет читать блоками файл `A`, пока не «упрется» в конец файла, и записывать прочитанные блоки в файл `B`. По окончании работы программа закроет оба файла. Код программы приведен в листинге 12.4.

Листинг 12.4. Программа копирования файла `A` в файл `B`

```
%include «system.inc»
#define BUFF_LEN 4096
CODESEG
START:
por eax                                ;извлекаем в EAX количество аргументов
                                        ;командной строки
cmp eax,3                             ;их должно быть 3
jae enough_params                    ;если не меньше, пытаемся открыть файлы
mov eax,255                           ;если меньше, выходим с кодом ошибки 255
endprog:
sys_exit eax                          ;системный вызов для завершения работы
enough_params:
por ebx                                ;извлекаем из стека первый аргумент.
                                        ;он нам не нужен, поэтому сразу
                                        ;извлекаем
por ebx                                ;второй — указатель на имя файла A.
sys_open EMPTY,O_RDONLY|O_LARGEFILE
                                        ;открываем только для чтения
test eax,eax                          ;ошибка? Выходим...
js endprog
mov ebp,eax                           ;дескриптор файла записываем в EBP
por ebx                                ;извлекаем в EBX имя файла B
sys_open EMPTY,O_WRONLY|O_LARGEFILE|O_CREAT|O_TRUNC,S_IRWXU
                                        ;открываем и перезаписываем,
                                        ;или создаем заново с правами 700

test eax,eax                          ;ошибка? выходим
js endprog
mov ebx,eax                           ;дескриптор файла B записываем в EBX
copy_next:
xchg ebp,ebx                          ;меняем местами EBX и EBP,
                                        ;в EBX — дескриптор файла A
sys_read EMPTY,buff,BUFF_LEN
                                        ;читаем 1й блок данных из файла A
test eax,eax                          ;проверка на ошибку
```

```

js end_close          ;ошибка? закрываем файлы и выходим
jz end_close          ;конец файла? закрываем файлы и выходим
xchg ebp,ebx          ;снова меняем местами ЕВР и ЕВХ,
                      ;в ЕВХ – дескриптор файла В
sys_write EMPTY,EMPTY,eax
                      ;выводим столько байтов, сколько
                      ;прочитано из файла А

test eax,eax
js endprog            ;ошибка?
jmp short copy_next   ;копируем новый блок
end_close:
sys_close EMPTY       ;перед завершением закрываем оба файла
xchg ebp,ebx          ;теперь закрываем второй файл
sys_close EMPTY
jmp short endprog      ;все
UDATASEG
buff resb BUFF_LEN    ;зарезервировано 4 Кб для буфера
END

```

Поиск позиции в файле

Иногда нужно изменить позицию чтения/записи файла. Например, если нам нужно прочитать файл дважды, то после первого прохода намного быстрее будет «перемотать» его, установив указатель позиции на начало файла, чем закрыть файл и открыть его заново.

Изменить позицию следующей операции чтения/записи можно с помощью системного вызова `lseek`:

```
off_t lseek(int fildes, off_t offset, int whence);
```

Первый аргумент — это, как обычно, дескриптор файла, второй — смещение — задает новую позицию в байтах, а третий определяет место в файле, откуда будет отсчитано смещение:

- `SEEK_SET` — смещение будет отсчитываться с начала файла;
- `SEEK_CUR` — с текущей позиции;
- `SEEK_END` — с конца файла.

Системный вызов `lseek` возвращает новую позицию — расстояние от начала файла в байтах — или код ошибки.

Небольшой пример: используя `lseek`, можно легко вычислить длину файла:

```
sys_lseek [fd], 0, SEEK_END
```

Вызов «перематывает» файл в конец, поэтому возвращаемое значение — это номер последнего байта, то есть размер файла.

А что случится, если создать файл, переместить указатель за пределы его конца, записать данные и закрыть файл? В DOS место от начала файла до позиции записи окажется заполнено случайными байтами. А в UNIX-подобных ОС вместо этого получится «разреженный» файл, логический размер которого больше его физического размера: физическое место под незаписанные данные отведено не будет.

Другие функции для работы с файлами

Файловая система UNIX-подобных операционных систем позволяет создавать файлы типа ссылки — жесткой или символической. Жесткая ссылка — это не что иное, как альтернативное имя файла: одно и то же содержимое, с одним и тем же владельцем и правами доступа, может быть доступно под разными именами. Все эти имена (жесткие ссылки) равноправны, и вы не можете удалить файл до тех пор, пока существует хотя бы одна жесткая ссылка на него. Механизм жестких ссылок не позволяет организовать ссылку на файл, находящийся в другой файловой системе (на другом логическом диске).

Для создания жесткой ссылки используется системный вызов **link**:

```
int link(const char *oldpath, const char *newpath);
```

Первый аргумент — оригинальное имя файла, второй — новое имя файла (имя жесткой ссылки).

Более гибким решением являются символические ссылки (**symlinks**). Такая ссылка указывает не непосредственно на данные файла, а только на его имя. Поэтому она может вести на файл на любой файловой системе и даже на несуществующий файл. Когда операционная система обращается к символической ссылке, та возвращает имя оригинального файла. Исключением из этого правила являются операции удаления и переименования файла — эти операции работают со ссылкой, а не с оригинальным файлом.

Для создания символической ссылки используется вызов **symlink**:

```
int symlink(const char *oldpath, const char *newpath);
```

Аргументы этого системного вызова такие же, как у системного вызова **link**.

Теперь давайте рассмотрим, как можно удалить и переместить файл.

Для удаления файла используется системный вызов **unlink**, который удаляет файл или одно из его имен. Напомним, что файл удаляется только после удаления последней жесткой ссылки на него.

```
int unlink(const char *pathname);
```

Системному вызову нужно передать только имя файла. В случае успеха вызов возвращает 0, а если произошла ошибка — отрицательное значение.

Для переименования файла используется вызов **rename**:

```
int rename(const char *oldpath, const char *newpath);
```


Оба аргумента полностью идентичны аргументам вызова **link**: это исходное и новое имя (точнее, путь) файла. Если новый путь отличается от старого только родительским каталогом, то результатом переименования окажется перемещение файла в другой каталог.

Напишем небольшую программу `symhard.asm`, которая создает одну жесткую и одну символическую ссылку на указанный файл (листинг 12.5). Созданные ссылки будут называться 1 и 2 соответственно.

Аргументы программы будут прочитаны из стека. Как обычно, первый элемент стека — количество аргументов, второй — имя программы, остальные — переданные аргументы.

Листинг 12.5. Пример программы создания жестких и символических ссылок

```
%include «system.inc»
CODESEG                                ;начало кода
START:
pop ebx                                ;читаем количество аргументов
cmp ebx,2                              ;первый — имя программы, второй — имя
                                        ;файла, для которого будут
                                        ;созданы ссылки
jz ok                                   ;да, переданы все параметры
endprog:
sys_exit 0                             ;выходим из программы
ok:
pop ebx                                ;это имя нашей программы — нам оно не
                                        ;нужно
pop ebx                                ;загружаем в EBX адрес имени файла
sys_link EMPTY,one                     ;создаем жесткую ссылку
sys_symlink EMPTY,two                  ;и символическую ссылку
                                        ;коды ошибок не проверяем
jmp short endprog                      ;заканчиваем обработку
DASEG
one DB «1»,0
two DB «2»,0
END                                     ;конец
```

«Пропишите» имя программы в **Makefile** и запустите **make**. В качестве аргумента вы можете передать имя файла или каталога. По окончании работы программы будут созданы два файла «1» и «2». Файл «1» — это жесткая ссылка, «2» — символическая. Если какой-то из этих файлов не создан, значит, произошла ошибка — например, ваша файловая система не поддерживает этого типа ссылок.

По команде `./symhard ./symhard` вы можете создать символическую и жесткую ссылки на сам исполняемый файл программы, а потом с помощью команд `ls -l`, `chown`, `chmod` и `rm` изучить на них свойства и поведение ссылок разного типа.

12.11. Работа с каталогами

Как и в DOS, в Linux есть набор системных вызовов, предназначенных для работы с каталогами. Благодаря Asmtutils использовать эти вызовы почти так же просто, как в высокоуровневых языках программирования.

Создание и удаление каталога (MKDIR, RMDIR)

Создать каталог позволяет системный вызов `mkdir`:

```
int mkdir(const char *pathname, mode_t mode);
```

Первый аргумент — это указатель на имя файла, а второй — права доступа, которые будут установлены для нового каталога. Права доступа указываются точно так же, как для системного вызова `open`.

Программка из листинга 12.6 создаст новый каталог `my_directory` в каталоге `/tmp`.

Листинг 12.6. Пример программы создания нового каталога

```
%include <system.inc>
CODESEG                                ;начало секции кода
START:                                ;начало программы
sys_mkdir name, S_IRWXU                ;создаем каталог, права 0700
sys_exit 0                             ;завершаем программу
DATASEG
name DB <"/tmp/my_directory">,0
END
```

Права доступа могут быть указаны также в восьмеричной форме. В отличие от языка C (и команды `chmod`) они записываются немного по-другому: без предваряющего нуля и с символом `q` в конце. Например, права доступа 0700 в восьмеричной системе будут выглядеть как 700q.

Для удаления каталога используется системный вызов `RMDIR`, которому нужно передать только имя каталога:

```
int rmdir(const char *pathname);
```

Смена текущего каталога (CHDIR)

Для смены текущего каталога используется системный вызов `chdir`:

```
int chdir(const char *path);
```

Чтобы в предыдущем примере перейти в только что созданный каталог, допишите перед вызовом макроса `sys_exit` вызов

```
sys_chdir name
```

Определение текущего каталога (GETCWD)

Системный вызов `getcwd`, возвращающий рабочий каталог, появился в ядре Linux версии 2.0 (он есть и в современных версиях 2.4–2.6). Этот системный вызов принимает два аргумента: первый — это буфер, в который будет записан путь, а второй — количество байтов, которые будут записаны:

```
long sys_getcwd(char *buf, unsigned long size)
```

Вот фрагмент программы, выводящей текущий каталог:

```
sys_getcwd path, PATHSIZE ;записываем в переменную path имя
                           ;текущего каталога
mov esi, ebx               ;сохраняем указатель в ESI
xor edx, edx               ;сбрасываем EDX
.next:
inc edx                    ;в EDX считаем длину строки path
lodsb                      ;читаем символ в AL, увеличиваем ESI
or al, al                  ;конец строки?
jnz .next                  ;нет? Читаем следующий символ
mov byte [esi-1], __n      ;заменяем нулевой байт на символ
                           ;конца строки
sub esi, edx               ;заново получаем адрес строки
sys_write STDOUT, esi, EMPTY ;выводим текущий каталог на STDOUT
                           ;длина строки уже находится в EDX
sys_exit_true              ;макрос для нормального завершения
```

12.12. Ввод с клавиатуры. Изменение поведения потока стандартного ввода. Системный вызов IOCTL

В этой главе мы уже рассматривали один способ ввода с клавиатуры — с помощью системного вызова `read`. Этот системный вызов ожидает нажатия клавиши «Enter» и только после этого возвращает прочитанную строку. Иногда полезно читать символы по одному или не дублировать их на устройство стандартного вывода (например, при вводе пароля). Для изменения поведения потока стандартного ввода служит системный вызов `IOCTL`.

Драйверы устройств не добавляют новых системных вызовов для управления устройствами, а вместо этого регистрируют наборы функций управления, вызываемых через системный вызов `IOCTL`.

IOCTL — это сокращение от **Input/Output Control** — управление вводом/выводом. Чтобы описать все виды `IOCTL`, нужно написать другую книгу или даже несколько книг, поэтому сосредоточимся только на терминальном вводе/выводе.

В Linux мы можем управлять поведением терминала (то есть клавиатуры и экрана), используя две `IOCTL`-функции — `TCGETS` и `TCSETS`. Первая, `TCGETS`, получает текущие настройки терминала, а вторая — устанавливает их. Подробное описание обеих функций может быть найдено в `man`-странице `termios`.

Структура данных, описывающая поведение терминала, доступна из `Asmutils`. Если нам нужно читать с клавиатуры символ за символом, не отображая при этом символы на экране, нам нужно изменить два атрибута терминала — `ICANON` и `ECHO`. Значение обоих атрибутов нужно установить в 0. Мы прочитаем эти атрибуты с помощью функции `TCGETS`, изменим значение битовой маски и применим новые атрибуты с помощью функции `TCSETS`. Для работы с терминалом нам понадобится структура `B_STRUC`, которая описана в файле `system.inc`.

```
mov edx,termattrs           ;заносим адрес структуры в EDX
sys_ioctl STDIN,TCGETS      ;загружаем структуру
mov eax,[termattrs.c_lflag] ;читаем флаги терминала
push eax                   ;и помещаем их в стек
and eax,~(ICANON|ECHO)      ;переключаем флаги ECHO и ICANON
mov [termattrs.c_lflag],eax ;заносим флаги в структуру
sys_ioctl STDIN, TCSETS      ;устанавливаем опции терминала
pop dword [termattrs.c_lflag] ;загружаем старые опции
```

Сама структура должна быть объявлена в сегменте `UDATASEG`:

```
termattrs B_STRUC termios,.c_lflag
```

К сожалению, более простого способа изменить настройки терминала нет.

12.13. Распределение памяти

При запуске новой программы ядро распределяет для нее память, как указано во время компиляции. Если нам нужна дополнительная память, мы должны попросить ядро нам ее выделить. В отличие от DOS, вместо того, чтобы запрашивать новый блок памяти определенного размера, мы просим увеличить секцию `.bss` (это последняя секция программы, содержащая неинициализированные переменные). Ответственность за всю свободную и распределенную в секции `.bss` память несет сам программист, а не ядро операционной системы.

Если мы не хотим использовать C-библиотеку для распределения памяти, мы должны сами программировать управление памятью (проще всего использовать `heap.asm` из `Asmutils`).

Системный вызов, предназначенный для увеличения или уменьшения секции `.bss`, называется `brk`:

```
void * brk(void *end_data_segment);
```

Ему нужно передать всего один параметр — новый конечный адрес секции `.bss`. Функция возвращает последний доступный адрес в секции данных `.bss`. Чтобы просто получить этот адрес, нужно вызвать эту функцию с нулевым указателем. Обычно `brk` используется следующим образом:

```
sys_brk 0                                ;получаем последний адрес
add eax,сколько_еще_байтов_мне_нужно    ;увеличиваем это значение
sys_brk eax                              ;устанавливаем новый адрес
```

После этого секция `.bss` будет увеличена на указанное количество байтов.

12.14. Отладка. Отладчик ALD

В UNIX-подобных операционных системах стандартным инструментом отладки является **gdb**, но из-за своей громоздкости он больше подходит для отладки программ на языке C, а не на языке ассемблера. В нашем случае лучше использовать более компактный отладчик, разработанный специально для программ на языке ассемблера — отладчик ALD (Assembly Language Debugger).

Это компактный, быстрый и бесплатный отладчик, распространяющийся по Стандартной Общественной лицензии GNU. Скачать его можно с сайта <http://ald.sourceforge.net>. Пока он поддерживает только x86-совместимые процессоры и только ELF в качестве формата исполняемых файлов, но вам этого должно хватить.

Давайте рассмотрим работу с ALD на примере нашей программы для преобразования символов из нижнего регистра в верхний.

Запустите отладчик с помощью команды **ald**:

```
ald
Assembly Language Debugger 0.1.3
Copyright (C) 2000-2002 Patrick Alken
ald>
```

Теперь загрузим нашу программу `convert`:

```
ald> load convert
echo: ELF Intel 80386 (32 bit), LSB, Executable, Version 1
```

```
(current)
Loading debugging symbols...(no symbols found)
ald>
```

Как и в любом другом отладчике, важнейшей из всех команд для нас является команда пошагового прохождения программы. В ALD она называется **s (step)**. Она выполняет одну команду программы и возвращает управление отладчику:

```
ald> s
eax = 0x00000000 ebx = 0x00000000 ecx = 0x00000000 edx = 0x00000000
esp = 0xBF88F8CC ebp = 0x00000000 esi = 0x00000000 edi = 0x00000000
ds = 0x0000002B es = 0x0000002B fs = 0x00000000 gs = 0x00000000
ss = 0x0000002B cs = 0x00000023 eip = 0x08048082 eflags = 0x000000346
Flags: PF ZF TF IF
08048082 5A                                pop edx
```

Следующая команда, которая будет выполнена, — **pop edx**. Она расположена в памяти по адресу 0x8048082. В регистре признаков установлен флаг нуля — **ZF** (остальные флаги нам не нужны). Чтобы снова выполнить последнюю введенную команду (в нашем случае **s**), просто нажмите «Enter». Выполняйте пошаговое прохождение, пока не дойдете до команды **int 0x80**, которая прочитает строку со стандартного ввода (**EAX = 0x00000003**):

```
ald>
eax = 0x00000003 ebx = 0x00000000 ecx = 0x080490C8 edx = 0x0000000A
esp = 0xBF88FD0 ebp = 0x00000000 esi = 0x00000000 edi = 0x00000000
ds = 0x0000002B es = 0x0000002B fs = 0x00000000 gs = 0x00000000
ss = 0x0000002B cs = 0x00000023 eip = 0x0804808D eflags = 0x000000346
Flags: PF ZF TF IF
0804808D CD80                                int 0x80
```

Значение регистра **EDX** (0x0000000A = 10d) ограничивает максимальную длину строки 10 символами. В **ECX** находится указатель на область памяти, в которую будет записана прочитанная строка. Мы можем просмотреть содержимое памяти с помощью команды **e (examine)**: **e ecx**:

```
ald> e ecx
Dumping 64 bytes of memory starting at 0x080490C8 in hex
080490C8: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
.....
080490D8: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
.....
080490E8: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
.....
080490F8: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
.....
```

Теперь снова введем команду отладчика `s` — этим мы выполним команду программы `int 0x80`. Программа будет ждать, пока вы введете строку и нажмете «Enter». Сейчас снова введите `e ecx` — посмотрим, что за строку мы ввели:

```
ald> e ecx
Dumping 64 bytes of memory starting at 0x080490C8 in hex
080490C8: 61 73 6D 20 72 75 6C 65 7A 0A 00 00 00 00 00 00
asm rulez.....
080490D8: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
.....
080490E8: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
.....
080490F8: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
.....
```

Используя команду `s`, мы можем выполнить оставшуюся часть программы без остановок. Она преобразует и выведет строку и снова будет ожидать ввода. Напоминаю, что мы написали программу так, что прервать ее можно только комбинацией клавиш `Ctrl+C`.

Команда `help` выводит список всех доступных команд отладчика, а `help имя_команды` — справку по указанной команде. В табл. 12.3 сведены наиболее полезные команды отладчика `ALD`.

Наиболее полезные команды отладчика `ALD`

Таблица 12.3

Команда	Назначение
load <filename>	Загружает файл (программу) в отладчик
set args <args>	Устанавливает аргументы программы
step [n]	Выполняет одну или несколько (n) команд программы. Вместо step можно использовать сокращение s
next [n]	Подобна step , но каждая подпрограмма выполняется за один шаг
disassemble	Преобразовывает машинный код команд обратно в символические имена языка ассемблера. Можно использовать сокращение d . Аргументом команды служит начальный адрес в программе, например d 0x08048061
continue	Продолжает выполнение программы (можно использовать сокращение c)
examine	Дамп памяти — выводит содержимое памяти начиная с указанного адреса в указанном формате и системе счисления. Можно использовать сокращение e . В качестве аргумента можно указать непосредственное значение адреса или регистр, содержащий адрес, например, e edx или e 0x08048000
register	Выводит значения всех регистров
help	Выводит список команд. Если нужно получить справку по той или иной команде, укажите ее имя как аргумент, например, help examine
break <addr>	Устанавливает точку прерывания (breakpoint) по адресу addr
lbreak	Выводит список установленных точек прерывания
quit	Выход. Можно использовать сокращение q

Отладчик `ALD` поддерживает установку точек прерывания (они же — контрольные точки, breakpoints). Когда программа во время выполнения «дохо-

дит» до точки прерывания, ее выполнение приостанавливается и управление передается отладчику. Таким способом можно проверить значения регистров или памяти в ключевых точках программы без пошагового выполнения.

Последняя на момент перевода книги версия отладчика 0.1.7 уже поддерживает работу с символьной информацией (именами меток и переменных), что очень помогает при отладке больших программ.

Напомним, что символьная информация может быть «упакована» в исполняемый файл с помощью ключа `-g` компилятора `nasm`. А при использовании `Asmutils` в файле `MCONFIG` нужно указать опцию `DEBUG=y`.

12.15. Ассемблер GAS

В этом параграфе мы вкратце рассмотрим «родной» ассемблер мира UNIX — `GAS`. Он используется вместе с компилятором `gcc`, когда в коде C-программы есть ассемблерные вставки.

Будучи «заточен» под сотрудничество с `gcc`, этот ассемблер не имеет развитых средств обработки макросов. Еще один его недостаток: при описании ошибок он очень лаконичен.

Синтаксис `GAS` заметно отличается от синтаксиса `NASM`: `NASM`-подобные (то есть `MASM` и `TASM`) компиляторы используют синтаксис `Intel`, а `GAS` использует синтаксис `AT&T`, который ориентирован на отличные от `Intel` чипы.

Давайте рассмотрим программу «Hello, World!», написанную в синтаксисе `AT&T` (листинг 12.7).

Листинг 12.7. Программа «Hello, World!», написанная на ассемблере GAS

```
.data                                # секция данных
msg:
.ascii «Hello, world!\n» # наша строка
len = . - msg            # ее длина
.text                    # начало секции кода
                        # метка _start — точка входа,
                        # то есть
.global _start          # начало программы для компоновщика

_start:
                        # выводим строку на stdout:
movl $len,%edx         # третий аргумент — длина строки
movl $msg,%ecx         # второй — указатель на строку
movl $1,%ebx           # первый — дескриптор файла STDOUT = 1
```



```
movl $4,%eax      # номер системного вызова 'write'
int $0x80          # вызываем ядро
                  # завершаем работу:
movl $0,%ebx       # помещаем код возврата в EBX
movl $1,%eax       # номер системного вызова 'exit' в EAX
int $0x80          # и снова вызываем ядро
```

12.16. Свободные источники информации

Если вы заинтересовались программированием на языке ассемблера под Linux, рекомендуем сайт <http://linuxassembly.org>. Здесь вы найдете не только ссылки на различные программы (компиляторы, редакторы), но и различную документацию по использованию системных вызовов. На этом же сайте вы найдете самый маленький в мире Web-сервер, занимающий всего 514 байтов (ищите `httpd.asm`).

12.17. Ключи командной строки компилятора

Чаще всего компилятор NASM вызывается со следующими ключами:

Ключ	Назначение
-v	Вывести номер версии компилятора
-g	«Упаковать» символьную информацию
-f <fmt>	Установить формат выходного файла <fmt> (см. главу 9)
-fh	Вывести список всех поддерживаемых форматов
-o <name>	Задаёт имя выходного файла <name>
-I <path>	Добавляет каталог для поиска «упакованных» файлов

Глава 13 Компоновка — стыковка ассемблерных программ с программами, написанными на языках высокого уровня

- Передача аргументов
- Что такое стек-фрейм?
- Компоновка с С-программой
- Компоновка с Pascal-программой

До сих пор при написании своих программ мы использовали только функции операционной системы, не обращаясь ни к каким библиотекам. В этой главе мы поговорим о «стыковке» нашей программы с программами, написанными на языках высокого уровня, а также с различными библиотеками. Поскольку различных языков высокого уровня очень много, мы ограничимся только языками C и Pascal.

13.1. Передача аргументов

Языки программирования высокого уровня поддерживают два способа передачи аргументов подпрограммам (функциям, процедурам): по значению и по ссылке. В первом случае значение подпрограммы не может изменить значения переданной переменной, поэтому способ «по значению» служит для передачи данных только в одном направлении: от программы к подпрограмме.

Во втором случае подпрограмма может изменить значение переменной, то есть передать значение в основную программу. Аргументы подпрограмм в языках высокого уровня всегда передаются через стек и никогда — через регистры процессора. Подпрограммы работают с переданными аргументами так, как если бы они были расположены в памяти, то есть как с обычными переменными — программисту не нужно ни извлекать самостоятельно аргумент из стека, ни помещать его в стек.

С точки зрения программиста на языке ассемблера, подпрограммы вызываются с помощью команды CALL, а возврат из подпрограммы осуществляется с помощью команды RET. Вызову команды CALL обычно предшествуют одна или несколько команд PUSH, которые помещают аргументы в стек. После входа в подпрограмму (то есть после выполнения CALL) в стеке выделяется определенное место для временного хранения локальных переменных. Определить позицию аргумента в стеке очень просто, поскольку во время компиляции известен порядок следования аргументов.

Пространство для локальных переменных в стеке выделяется простым уменьшением указателя вершины стека (ESP) на требуемый размер. Поскольку стек нисходящий, то, уменьшая указатель на вершину стека, мы получаем дополнительное пространство для хранения локальных переменных. Нужно помнить,

что значения локальных переменных при их помещении в стек не определены, поэтому в подпрограмме нужно первым делом их инициализировать.

Перед выходом из подпрограммы нужно освободить стек, увеличив указатель на вершину стека, а затем уже выполнить инструкцию RET.

Различные языки высокого уровня по-разному работают со стеком, поэтому в этой главе мы рассмотрим только С и Паскаль.

13.2. Что такое стек-фрейм?

Доступ к локальным переменным непосредственно через указатель вершины стека (Е) SP не очень удобен, поскольку аргументы могут быть различной длины, что усложняет нашу задачу. Проще сразу после входа в подпрограмму сохранить адрес (смещение) начала стека в некотором регистре и адресовать все параметры и локальные переменные, используя этот регистр. Идеально для этого подходит регистр (Е)BP.

Исходное значение (Е)BP сохраняется в стеке после входа в подпрограмму, а в (Е)BP загружается указатель вершины стека (Е)SP. После этого в стеке отводится место для локальных переменных, а переданные аргументы доступны через (Е)BP.

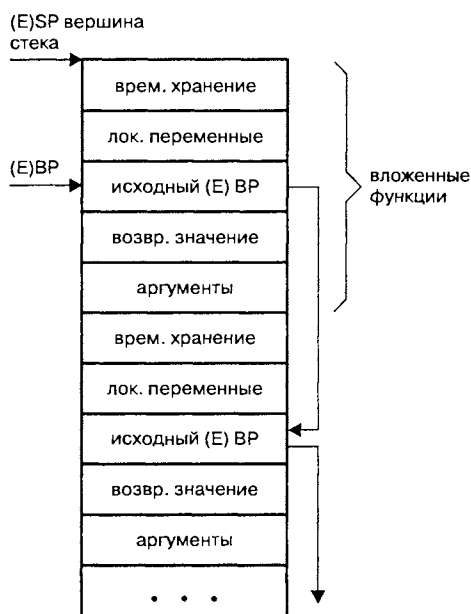


Рис. 13.1. Распределение стека (stackframe)

«Окно» стека, содержащее аргументы, возвращаемое значение, локальные переменные и исходное значение (Е)ВР, называется стек-фреймом.

Чем больше глубина вложенности подпрограммы (то есть количество подпрограмм, вызывающих одна другую), тем больше окон в стеке. Исходное значение (Е)ВР, сохраненное в стеке, указывает на предыдущий стек-фрейм. Получается связный список окон, который при отладке служит для отслеживания вызовов функций.

13.2.1. Стек-фрейм в языке С (32-битная версия)

Рассмотрим передачу аргументов функции в языке С. Следующая программа откомпилирована в 32-битном окружении, а именно в операционной системе Linux:

```
int addit(int a ,int b) {
    int d = a + b;
    return d;
}
int main(void) {
    int e;
    e = addit(0x55,0xAA);
}
```

Основная программа передает два значения 0x55 и 0xAA подпрограмме **addit**, которая их складывает и возвращает сумму. Вот листинг основной программы (функции **main**) на языке ассемблера:

080483F4	55	push ebp
080483F5	89E5	mov dword ebp, esp
080483F7	83EC18	sub dword esp, 0x18
080483FA	83C4F8	add dword esp, 0xffffffff8
080483FD	68AA000000	push 0xaa
08048402	6A55	push 0x55
08048404	E8C7FFFFFFF	call near +0xffffffc7
08048409	83C410	add dword esp, 0x10
0804840C	89C0	mov dword eax, eax
0804840E	8945FC	mov dword [ebp+0xfc], eax
08048411	89EC	mov dword esp, ebp
08048413	5D	pop ebp
08048414	C3	ret

Функция **main** первым делом сохраняет исходное значение ЕВР (указатель на предыдущий фрейм) в стеке. Вторая команда копирует в ЕВР текущее значение ESP, начиная этим новый фрейм. Две следующие команды, SUB и ADD, отнимают значения 0x18 и 0x8 от указателя вершины стека, выделяя место для локальных переменных. Обратите внимание, что вычитание 8 записано как прибавление отрицательного числа в дополнительном коде.

Аргументы передаются подпрограмме справа налево, то есть обратно тому порядку, в котором они записаны в исходном коде на С. Это сделано для того, чтобы разрешить использование в языке С функций с переменным количеством аргументов. Пара команд **PUSH** помещает наши значения (0x55 и 0xAA) в стек. Следующая команда, **CALL**, вызывает подпрограмму **addit**.

Рассмотрим окно стека после входа в **addit**. Сделаем это в табличной форме (см. табл. 13.1).

Окно стека после входа в **addit**

Таблица. 13.1

Адрес стека	Значение по адресу
0xBFFFFFF860 = ESP	0x08048409 (адрес возврата в основную функцию)
0xBFFFFFF864	0x00000055
0xBFFFFFF868	0x000000AA
0xBFFFFFF86C — 0xBFFFFFF84	временная память функции main
0xBFFFFFF880	не определено: локальная переменная e
0xBFFFFFF88C = EBP	0xBFFFFFF8C8 — исходный EBP какой-то функции библиотеки libc
0xBFFFFFF890	0x40039CA2 — адрес возврата в библиотечную функцию

Вершина стека содержит 32-битный адрес возврата, то есть адрес той команды в теле вызывающей функции, которая следует сразу после команды **CALL**. За ним следуют два 32-битных значения — это аргументы, переданные функции **addit**. Для локальной переменной **e** зарезервировано место в окне стека, но ее значение пока не определено. За ней следует значение регистра **EBP**, которое он имел сразу перед вызовом функции **main**, а за ним — адрес возврата в библиотеку **libc**, откуда была вызвана функция **main**. Подробности вызова самой **main** для нас сейчас не важны.

Вот листинг функции **addit**:

```

080483D0    55                push ebp
080483D1    89E5             mov dword ebp, esp
080483D3    83EC18           sub dword esp, 0x18
080483D6    8B4508           mov eax, dword [ebp+0x8]
080483D9    8B550C           mov edx, dword [ebp+0xc]
080483DC    8D0C02           lea ecx, [eax+edx]
080483DF    894DFC           mov dword [ebp+0xfc], ecx
080483E2    8B55FC           mov edx, dword [ebp+0xfc]
080483E5    89D0             mov dword eax, edx
080483E7    EB07             jmp short +0x7
080483E9    8DB42600000000  lea esi, [esi+0x0]
080483F0    89EC             mov dword esp, ebp
080483F2    5D               pop ebp
080483F3    C3               retn

```

Начало кода `addit` выглядит точно так же. Сначала она сохраняет в стеке исходное значение `EBP`, указывающее на стек-фрейм функции `main`. Затем создается новое окно стека для функции `addit`: в `EBP` загружается значение `ESP`. Команда `SUB` резервирует в стеке место для локальной переменной `d`, которая будет использоваться для хранения суммы `a+b`. Рассмотрим состояние стека до выполнения первой собственной команды подпрограммы `addit`, то есть после выполнения команды `SUB` (табл. 13.2).

Состояние стека до выполнения первой собственной команды подпрограммы `addit` Таблица 13.2

Адрес стека	Значение по адресу
0xBFFFF844 = ESP	не определено: выравнивание
0xBFFFF848 — 0xBFFFF854	не определено: выравнивание
0xBFFFF858 = EBP — 4 (+0xFFFFF8FC)	не определено: локальная переменная <code>d</code>
0xBFFFF85C = EBP	0xBFFFF88C — исходный <code>EBP</code> функции <code>main</code> (указатель на ее стек-фрейм)
0xBFFFF860 = EBP + 4	0x08048409 (адрес возврата в функцию <code>main</code>)
0xBFFFF864 = EBP + 8	0x00000055
0xBFFFF868 = EBP + 0xC	0x000000AA
0xBFFFF86C — 0xBFFFF874	временное хранилище функции <code>main</code>
0xBFFFF888	не определено: локальная переменная <code>e</code> из функции <code>main</code>
0xBFFFF88C	0xBFFFF8C8 — исходный <code>EBP</code> какой-то функции библиотеки <code>libc</code>
0xBFFFF890	0x40039CA2 — адрес возврата в библиотечную функцию

Как видите, в стеке есть несколько двойных слов (`dword`) — они используются исключительно для выравнивания памяти по адресам, кратным степени двойки. За ними следует двойное слово, отведенное для переменной `d`, значение которой пока не определено. После этого следует указатель на окно стека функции `main`. Далее следуют аргументы подпрограммы `addit`, место для локальной переменной функции `main` и данные для возврата из самой `main`.

Чтение аргументов функции `addit` в регистры `EAX` и `EDX` выполняют следующие команды:

```
080483D6    8B4508                mov eax, dword [ebp+0x8]
080483D9    8B550C                mov edx, dword [ebp+0xc]
```

Почему программа ищет первый аргумент именно по смещению `0x8`? Потому что описание стек-фрейма требует, чтобы аргументы подпрограммы начинались на «расстоянии» 8 байтов (то есть два элемента стек-фрейма) от значения `EBP`. Следующий аргумент находится на расстоянии `0xC` байтов, то есть $8 + 4 = 12$ байтов.

В табл. 13.3 перечислены адреса в стеке `EBP` (смещения относительно `EBP`), важные с точки зрения программиста.

Важные адреса в стеке EBP

Таблица 13.3

Адрес	Назначение
[ebp — 4]	Место для локальной переменной d
[ebp + 0]	Оригинальное значение EBP
[ebp + 4]	Адрес возврата
[ebp + 8]	Первый аргумент
[ebp + 0xC]	Второй аргумент

Первый аргумент функции **addit** равен 0x55. Помещенный в стек последним, теперь он находится ближе к вершине стека EBP, то есть извлекается из стека первым.

Оставшаяся часть функции **addit** вычисляет сумму аргументов, загруженных в EAX и EDX, с помощью команды LEA. Результат подпрограммы ожидается в EAX. После завершения **addit** ее стек-фрейм будет разрушен следующими командами:

```
080483F0    89EC                mov dword esp, ebp
080483F2    5D                  pop ebp
```

Первая команда «теряет» локальные переменные и выравнивание, а вторая восстанавливает исходный стек-фрейм функции **main**. Удаление переданных в стек аргументов входит в обязанности вызывающей функции (**main**), потому что только ей известно их количество.

Смещение переданного аргумента относительно EBP зависит от его типа. Значение, тип которого имеет размер меньший двойного слова (**char**, **short**), выравнивается по адресу, кратному 4 байтам.

13.2.2. Стек-фрейм в языке C (16-битная версия)

Все, что было сказано о стек-фрейме, остается в силе и для 16-битных версий языка C, но со следующими очевидными отличиями:

- минимальный размер передаваемого через стек значения — не двойное слово (4 байта), а слово (2 байта);
- вместо 32-битных регистров используются их 16-битные версии (т.е. BP вместо EBP, SP вместо ESP и т.д.);
- возвращаемое функцией значение ожидается не в EAX, а в паре DX:AX.

Заметьте, что в своей ассемблерной программе вы можете использовать любые 32-битные команды, если они доступны (то есть если вы не собираетесь выполнять программу на древнем 80286 процессоре).

В 16-битном мире у нас будут небольшие проблемы из-за различных способов вызова функций. В C используются так называемые модели памяти, которые задают максимальные размеры секций кода и данных программы. Компиляция

С-программы с моделью памяти `medium`, `large` или `huge` сгенерирует для вызова функций команды `CALL FAR`, а с остальными моделями — `CALL NEAR`.

«Ближний» вызов сохраняет в стеке только смещение следующей команды, поскольку предполагается, что сегментная часть адреса сохранится и так. Отсюда следует, что «расстояние» от вершины стека BP до первого аргумента будет на 2 байта меньше, чем при использовании вызова `FAR`. Для возврата из `FAR`-вызовов служит команда `RETF`, а не `RETN`.

Компилятор `NASM` содержит набор макросов, позволяющих решить проблему разных моделей памяти, что значительно облегчает компоновку с С- и Паскаль-программами.

13.3. Компоновка с С-программой

Большинство компиляторов С снабжает глобальные символы из пространства имен С префиксом «`_`». Например, функция `printf` будет доступна программисту на ассемблере как `_printf`. Исключением является только формат `ELF` (использующийся в `Linux`), не требующий символа подчеркивания.

Давайте напишем небольшую С-программу, вызывающую функцию `printit`, которая вычисляет сумму глобальной переменной `plus` и единственного аргумента и выводит ее на экран. Функцию `printit` напишем на языке ассемблера, а из нее вызовем библиотечную функцию `printf`. Писать и запускать программу будем в операционной системе `Linux`.

С-часть нашей программы очень проста:

```
const int plus = 6;
void printit(int);
int main(void) {
    printit(5);
}
```

Мы определили глобальную константу `plus` и присвоили ей значение 6. Затем идет объявление прототипа функции `printit`. После этого — функция `main`, вызывающая функцию `printit` с аргументом 5.

В ассемблерной программе нам нужно объявить используемые нами глобальные символы **`plus`** и **`printf`**:

```
extern plus
extern printf
```

Поскольку мы собираемся использовать компилятор `gcc` и формат исполняемого файла `ELF`, нам не нужно возиться с символами подчеркивания.

Благодаря следующей директиве `include` нам доступны макросы `proc`, `arg` и `endproc`, которые облегчают написание нашей функции:

```
%include «misc/c32.mac»
```

Макрос `proc` определяет начало функции `printit`. Подстановка превращает его в две команды: `push ebp` и `mov ebp,esp`. Следующий макрос `arg` определяет наш единственный аргумент. Если нужно передать несколько аргументов, то первый `arg` будет соответствовать крайнему слева аргументу в коде C. Размер аргумента по умолчанию — 4 байта (определен в `c32.mac`).

```
proc printit
  %$what arg
```

Оставшаяся часть программы понятна:

```
mov eax,[ebp + %$what] ;читаем первый аргумент из стека
add eax,[plus]         ;прибавляем глобальную
                        ;переменную plus
push eax               ;сохраняем в стеке последний аргумент
                        ;функции printf
push str1              ;первый аргумент - строка формата
                        ;вывода
call printf            ;вызываем printf
endproc               ;макрос endproc восстанавливает ЕВР,
                        ;уничтожает все локальные переменные
                        ;(у нас их нет) и выходит из подпрограммы
```

Вся программа представлена в листинге 13.1.

Листинг 13.1. Пример компоновки ассемблерного кода с C-кодом

```
%include «misc/c32.mac» ;подключаем макросы
section .text           ;начало секции кода
extern plus             ;объявляем глоб. переменную plus
extern printf           ;объявляем глоб. функцию printf
global printit          ;экспортируем функцию printit
proc printit            ;начало функции printit, принимающей
  %$what arg            ;один аргумент по имени what
  mov eax,[ebp + %$what] ;читаем из стека аргумент
  add eax,[plus]        ;прибавляем глобальную переменную plus
  push eax              ;сохраняем в стеке последний аргумент
                        ;функции printf
  push str1             ;первый аргумент - строка формата вывода
  call printf           ;вызываем printf
endproc                ;макрос endproc восстанавливает ЕВР,
                        ;уничтожает все локальные переменные
                        ;(у нас их нет) и выходит из подпрограммы

section .data           ;секция данных
str1 db «SUM = %d»,0x0A,0x0 ;строка формата, завершенная
                        ;символом перевода
                        ;строки и нулевым байтом
```

Сохраним нашу программу в файле `printit.asm` и откомпилируем его:

```
nasm -f elf printit.asm
```

Исходный код С-программы сохраним в файле `main.c` и откомпилируем его с помощью `gcc`, скомпоновав с модулем `printit.o`:

```
gcc -o printit main.c printit.o
```

Запустив полученную программу `printit`, мы увидим, что она выводит «SUM = 11».

Чтобы собрать 16-битную версию программы, нужно подключить другой заголовочный файл — `cl6.mac`. Макросы `proc` и `endproc` сами решают проблему с NEAR и FAR-вызовами. При использовании FAR-модели памяти нужно указать:

```
%define FARCODE
```

Обо всем остальном позаботятся макросы.

Если вы используете компилятор, требующий символов подчеркивания, вам поможет небольшой макрос:

```
%macro cglobal 1
global _%1
%define %1 _%1
%endmacro

%macro cextern 1
extern _%1
%define %1 _%1
%endmacro
```

После этого вы можете использовать директивы `cglobal` и `cextern` вместо обычных `global` и `extern`. Новые версии директив автоматически будут добавлять символы подчеркивания.

13.4. Компоновка с Pascal-программой

В общих чертах механизм передачи аргументов в Паскале не отличается от только что рассмотренного механизма в языке С. Аргументы передаются через стек, оба типа подпрограмм (процедуры и функции) используют стек-фрейм (или его 16-битную версию). Переменные и аргументы доступны через регистр BP.

Паскаль изначально разрабатывался как язык для обучения программированию, поэтому он не поддерживает некоторых конструкций языка С — например, функций с переменным количеством аргументов. Поэтому компилятор Паскаля помещает аргументы в стек слева направо, а не в обратном порядке, как в С.

Вызов внешних подпрограмм на Паскале из ассемблерной программы упрощен, потому что в Паскале используются только FAR-вызовы. «Генеральная уборка» стека — забота не вызывающей программы, а вызываемой функции,

которая должна выполнить команду `getf` с аргументом, указывающим, сколько байтов выбросить из стека. Рассмотрим таблицу размещения стека подпрограммы относительно `BP` (табл. 13.4).

Таблица размещения стека подпрограммы относительно `BP`

Таблица 13.4

Адрес	Назначение
<code>[BP — ...]</code>	Локальные переменные
<code>[BP + 0]</code>	Исходное значение <code>BP</code> (2 байта)
<code>[BP + 2]</code>	Адрес возврата — регистр <code>IP</code> (2 байта)
<code>[BP + 4]</code>	Сегмент адреса возврата — регистр <code>CS</code> (2 байта)
<code>[BP + 6]</code>	Первый аргумент
<code>[BP + ...]</code>	Следующие аргументы

Давайте напишем на Паскале программу, подобную только что рассмотренной. Новая версия нашей программы для вывода значения суммы будет вызывать функцию `writeln` из основной программы, то есть подпрограмма **addit** должна только подсчитать сумму и вернуть ее в основную программу.

```
{ $L addit.obj }
uses crt;
var plus:integer;
function addit(x:integer):longint;far;external;
begin
  plus := 6;
  writeln('SUM = ', addit(5));
end.
```

Функцию **addit**, как и в предыдущем примере, напишем на языке ассемблера. Программе на Паскале она становится известна посредством ключевого слова **external**. В той же строке декларации мы объявляем, что функции **addit** нужно передать один целочисленный параметр, она возвращает значение типа `longint` (4 байта) и вызывается как `FAR`. Не забудьте также написать директиву `$L`, указывающую имя объектного файла с откомпилированной функцией **addit**, который нужно подключить при компоновке.

Теперь напишем функцию `addit`, которую мы сохраним в файле `addit.asm`.

Borland Turbo Pascal использует не стандартный формат объектного файла `obj`, а свой собственный, что накладывает строгие ограничения на именование секций программы. Секция кода должна называться `CODE`, `CSEG` или именем, заканчивающимся на «`_TEXT`», секция данных — `CONST` или именем, заканчивающимся на «`_DATA`», а секция неинициализированных данных — `DATA`, или `DSEG`, или именем, заканчивающимся на «`_BSS`».

Если вы уже привыкли называть секции программы именами `.text`, `.data` и `.bss`, то в программах, предназначенных для компоновки с Паскаль-кодом, просто добавьте знак подчеркивания после точки, и компилятор будет доволен.

Функция `addit` будет вызываться в 16-битной среде, где размер типа данных `integer` равен 2 байтам. Возвращает функция значение типа `longint` (4 байта) и передает его через пару регистров `DX:AX`. Все эти вопросы, в том числе вопросы компоновки с Паскаль-кодом, решает макрос `c16.mac`. Код нашей подпрограммы `addit` представлен в листинге 13.2.

Листинг 13.2. Пример подпрограммы для паскалевской программы

```
SECTION .TEXT                ;начало секции кода
%define PASCAL                ;будем использовать FAR-вызовы
#include «misc/c16.mac»       ;подключаем макросы
extern plus                   ;получаем доступ к внешней
                              ;переменной plus

global addit                  ;экспортируем функцию addit
proc addit                    ;начало функции addit
                              ;макрос включает создание стек-фрейма
%$what arg                    ;объявляем единственный аргумент
                              ;по имени what
xor dx,dx                     ;сбрасываем DX
mov ax,[bp+%$what]            ;AX = what
add ax,[plus]                 ;AX = AX + plus
adc dx,0                       ;не забудьте флаг переноса
endproc                       ;удаляем полученные аргументы и
                              ;выходим из подпрограммы
```

Макрос `arg` принимает в качестве необязательного параметра размер передаваемого аргумента. Размер по умолчанию в 16-битной среде равен 2 байтам. Если бы мы передавали аргумент типа `longint` или указатель, нам пришлось бы явно задать его размер, то есть 4 байта.

Теперь посмотрим, во что превратятся макросы после подстановки:

```
cs:0076    55          push    bp          ;макрос proc
cs:0077    89E5        mov     bp,sp       ;макрос proc
cs:0079    31D2        xor     dx,dx       ;DX=0
cs:007B    8B4606      mov     ax,[bp+06]  ;получаем 1-й
                              ;аргумент
cs:007E    03065200    add     ax,[0052]   ;сложение аргумента
                              ;и переменной plus
cs:0082    81D20000    adc     dx,0000     ;сложение с переносом
cs:0086    89EC        mov     sp,bp       ;макрос endproc
cs:0088    5D          pop     bp          ;макрос endproc
cs:0089    CA0200      retf    0002        ;макрос endproc
                              ;все
```

После компиляции и запуска программы она выведет требуемую сумму на экран.

Глава 14 Заключение

Несмотря на то, что вы дочитали эту книгу до конца, ваше изучение языка ассемблера на этом не заканчивается, а, наоборот, только начинается. Вы заложили только фундамент дальнейшего изучения, познакомившись с основными понятиями и конструкциями языка ассемблера, которых должно хватить для написания простеньких программ. Если вы заинтересовались и хотите разрабатывать сложные ассемблерные программы, рекомендуем ознакомиться с расширенными наборами команд процессоров (MMX, SSE, 3DNow), описания которых вы найдете в Интернете. А также прочитать книгу, посвященную программированию на языке ассемблера под конкретную операционную систему — DOS, Windows или Linux.

Все программы и фрагменты программ, представленные в этой книге, распространяются свободно без лицензионных ограничений.

Все торговые марки и наименования программных продуктов являются зарегистрированными торговыми марками их владельцев.

Глава 15 **Часто используемые команды**

Группа команд	Назначение
Команды перемещения значений	MOV — скопировать
	XCHG — поменять местами значения
	PUSH — сохранить в стеке
	POP — извлечь из стека
Арифметические команды	ADD — сложение
	SUB — вычитание
	MUL — умножение
	DIV — деление
	INC — инкремент (увеличение на 1)
	DEC — декремент (уменьшение на 1)
Логические команды	AND — логическое И (логическое умножение)
	OR — логическое ИЛИ (логическая сумма)
	XOR — исключительное ИЛИ
	NOT — отрицание
Сравнение	CMP — сравнение
	TEST — поразрядное сравнение
Сдвиг и ротация	SHR — логический (поразрядный) сдвиг вправо
	SHL — логический (поразрядный) сдвиг влево
	RCR — ротация через флаг переноса вправо
	RCL — ротация через флаг переноса влево
Команды перехода	JMP — безусловный переход
	LOOP — переход, пока (E)CX не сравняется с 0
Условные переходы	JZ — если флаг нуля (ZF) установлен
	JC — если флаг переноса (CF) установлен
	JNZ — если флаг нуля (ZF) не установлен
	JNC — если флаг переноса (CF) не установлен
Подпрограммы	CALL — вызов подпрограммы
	RET — возврат из подпрограммы
	INT — вызов прерывания
Операции над строками	REP — повторять следующую команду, пока (E)CX не 0
	MOVSB — копирование строк
	CMPSB — сравнение строк
	SCASB — поиск в строке

000	00		043	2B	+	086	56	U	129	81	ü	172	AC	À	215	D7	Ï
001	01	0	044	2C	,	087	57	U	130	82	û	173	AD	Á	216	D8	Î
002	02	0	045	2D	-	088	58	X	131	83	ü	174	AE	Â	217	D9	Í
003	03	0	046	2E	.	089	59	Y	132	84	ü	175	AF	Ã	218	DA	Ì
004	04	0	047	2F	/	090	5A	Z	133	85	ü	176	B0	Ä	219	DB	Ï
005	05	0	048	30	0	091	5B	I	134	86	ü	177	B1	Å	220	DC	Î
006	06	0	049	31	1	092	5C	\	135	87	ü	178	B2	Æ	221	DD	É
007	07	0	050	32	2	093	5D]	136	88	ü	179	B3	Ç	222	DE	È
008	08	0	051	33	3	094	5E	^	137	89	ü	180	B4	È	223	DF	Ð
009	09	0	052	34	4	095	5F	~	138	8A	ü	181	B5	É	224	E0	Ñ
010	0A	0	053	35	5	096	60	^	139	8B	ü	182	B6	Ê	225	E1	Ò
011	0B	0	054	36	6	097	61	a	140	8C	ü	183	B7	Ë	226	E2	Ó
012	0C	0	055	37	7	098	62	b	141	8D	ü	184	B8	Ì	227	E3	Ô
013	0D	0	056	38	8	099	63	c	142	8E	ü	185	B9	Í	228	E4	Õ
014	0E	0	057	39	9	100	64	d	143	8F	ü	186	BA	Î	229	E5	Ö
015	0F	0	058	3A	:	101	65	e	144	90	ü	187	BB	Ï	230	E6	×
016	10	0	059	3B	;	102	66	f	145	91	ü	188	BC	Ï	231	E7	¸
017	11	0	060	3C	<	103	67	g	146	92	ü	189	BD	Ï	232	E8	¸
018	12	0	061	3D	=	104	68	h	147	93	ü	190	BE	Ï	233	E9	¸
019	13	0	062	3E	>	105	69	i	148	94	ü	191	BF	Ï	234	EA	¸
020	14	0	063	3F	?	106	6A	j	149	95	ü	192	C0	Ï	235	EB	¸
021	15	0	064	40	@	107	6B	k	150	96	ü	193	C1	Ï	236	EC	¸
022	16	0	065	41	A	108	6C	l	151	97	ü	194	C2	Ï	237	ED	¸
023	17	0	066	42	B	109	6D	m	152	98	ü	195	C3	Ï	238	EE	¸
024	18	0	067	43	C	110	6E	n	153	99	ü	196	C4	Ï	239	EF	¸
025	19	0	068	44	D	111	6F	o	154	9A	ü	197	C5	Ï	240	F0	¸
026	1A	0	069	45	E	112	70	p	155	9B	ü	198	C6	Ï	241	F1	¸
027	1B	0	070	46	F	113	71	q	156	9C	ü	199	C7	Ï	242	F2	¸
028	1C	0	071	47	G	114	72	r	157	9D	ü	200	C8	Ï	243	F3	¸
029	1D	0	072	48	H	115	73	s	158	9E	ü	201	C9	Ï	244	F4	¸
030	1E	0	073	49	I	116	74	t	159	9F	ü	202	CA	Ï	245	F5	¸
031	1F	0	074	4A	J	117	75	u	160	A0	ü	203	CB	Ï	246	F6	¸
032	20	0	075	4B	K	118	76	v	161	A1	ü	204	CC	Ï	247	F7	¸
033	21	0	076	4C	L	119	77	w	162	A2	ü	205	CD	Ï	248	F8	¸
034	22	0	077	4D	M	120	78	x	163	A3	ü	206	CE	Ï	249	F9	¸
035	23	0	078	4E	N	121	79	y	164	A4	ü	207	CF	Ï	250	FA	¸
036	24	0	079	4F	O	122	7A	z	165	A5	ü	208	D0	Ï	251	FD	¸
037	25	0	080	50	P	123	7B	<	166	A6	ü	209	D1	Ï	252	FE	¸
038	26	0	081	51	Q	124	7C	i	167	A7	ü	210	D2	Ï	253	FD	¸
039	27	0	082	52	R	125	7D	>	168	A8	ü	211	D3	Ï	254	FE	¸
040	28	0	083	53	S	126	7E	~	169	A9	ü	212	D4	Ï	255	FF	¸
041	29	0	084	54	T	127	7F	o	170	AA	ü	213	D5	Ï			
042	2A	0	085	55	U	128	80	ç	171	AB	ü	214	D6	Ï			

Рис. 15.1. Расширенная таблица ASCII (без символов национальных алфавитов)